

High Performance Data Management and  
Processing  
I590, Section 7462

Zdzisław Meglicki

April 29, 2004

Document version

`$Id: I590.tex,v 1.197 2004/04/29 21:57:35 gustav Exp $`

# Contents

<b>1</b>	<b>Introduction</b>	<b>7</b>
1.1	The Focus of the Course . . . . .	7
1.2	Required Level of Skills . . . . .	8
1.3	Bring Your Laptop . . . . .	9
1.4	Using NetMeeting . . . . .	9
<b>2</b>	<b>Supercomputers and Clusters</b>	<b>11</b>
2.1	The Almighty PC . . . . .	11
2.2	Supercomputers and Clusters . . . . .	12
2.3	HPC Facilities at Public Supercomputer Centers . . . . .	14
2.4	IU HPC Facilities . . . . .	15
2.5	Parallel Programming . . . . .	16
2.5.1	Data Parallel Paradigm . . . . .	16
2.5.2	Message Passing Paradigm . . . . .	18
2.5.3	Shared Memory Paradigm . . . . .	18
2.5.4	Mixing Parallel Programming Paradigms . . . . .	19
2.5.5	Scalability of Parallel Programming Paradigms . . . . .	20
2.6	HPC and The Grid . . . . .	20
2.7	To Program Or Not To Program . . . . .	23
<b>3</b>	<b>Working with the AVIDD Cluster</b>	<b>27</b>
3.1	Your AVIDD Account . . . . .	28
3.2	Connecting to the AVIDD Cluster . . . . .	29
3.3	Finding Help and Documentation . . . . .	34
3.4	Working with Data on AVIDD . . . . .	36
3.4.1	The AVIDD GPFS . . . . .	36
3.4.2	IU HPSS . . . . .	39
3.4.3	Moving Data Between HPSS and GPFS . . . . .	40
<b>4</b>	<b>Working with PBS</b>	<b>59</b>
4.1	PBS Configuration . . . . .	59
4.2	Submitting, Inspecting and Cancelling PBS Jobs . . . . .	62
4.3	Specification of PBS Jobs . . . . .	67
4.3.1	Interactive PBS Jobs . . . . .	67

4.3.2	Not Quite Interactive PBS Jobs . . . . .	69
4.3.3	Simple PBS Directives . . . . .	69
4.3.4	Jobs Dependent On Other Jobs . . . . .	73
4.4	Checkpointing and Resubmission . . . . .	86
4.4.1	Timing a Job . . . . .	87
4.4.2	Restoring and Saving the State of a Job . . . . .	91
4.4.3	Restoring, Timing and Saving a Job: the Complete Ap- plication . . . . .	96
4.4.4	Combining the Application with PBS: Automatic Resub- mission . . . . .	102
<b>5</b>	<b>MPI and MPI-IO</b>	<b>107</b>
5.1	Introduction . . . . .	107
5.1.1	The History of MPI . . . . .	107
5.1.2	MPI Documentation and Literature . . . . .	108
5.1.3	What is in MPI? . . . . .	109
5.1.4	Programming Examples . . . . .	110
5.1.5	Running MPI on the AVIDD Clusters . . . . .	111
5.2	Basic MPI . . . . .	117
5.2.1	Hello World . . . . .	117
5.2.2	Greetings, Master . . . . .	122
5.2.3	Dividing the Pie . . . . .	128
5.2.4	Bank Queue . . . . .	131
5.2.5	Diffusion . . . . .	143
5.2.6	Interacting Particles . . . . .	159
5.2.7	The Random Pie . . . . .	172
5.2.8	Error Handling . . . . .	180
5.3	MPI IO . . . . .	192
5.3.1	Writing on MPI Files . . . . .	193
5.3.2	Reading from MPI Files . . . . .	204
5.3.3	Writing Sequential Files with MPI-IO . . . . .	215
5.3.4	File Views . . . . .	224
5.3.5	File Hints . . . . .	246
5.3.6	Asynchronous IO . . . . .	252
5.4	MPI Graphics and Process Visualization . . . . .	253
5.4.1	Instrumenting MPI Programs with MPE . . . . .	254
5.4.2	MPE Graphics . . . . .	261
5.4.3	Exercises . . . . .	273
<b>6</b>	<b>The Assignment</b>	<b>275</b>
<b>7</b>	<b>HDF5</b>	<b>279</b>
7.1	Introduction . . . . .	279
7.2	Structured versus Flat Files . . . . .	280
7.3	Creating and Structuring HDF5 Files . . . . .	286
7.3.1	HDF5 File Creation . . . . .	286

7.3.2	HDF5 Datasets . . . . .	288
7.3.3	Writing On and Reading From HDF5 Datasets . . . . .	291
7.3.4	HDF5 Attributes . . . . .	294
7.3.5	HDF5 Groups . . . . .	297
7.3.6	HDF5 Groups and Datasets . . . . .	298
7.4	Property Lists . . . . .	302
7.4.1	Modifying Property Lists . . . . .	305
7.4.2	Create a File Family . . . . .	306
7.4.3	Create an MPI-IO File . . . . .	308
7.4.4	Create an Extendible Dataset . . . . .	310
7.4.5	Create a Compressed Dataset . . . . .	318
7.4.6	Activate Checksum for Error Detection . . . . .	322
7.5	Hyperslab Selection . . . . .	325
7.5.1	Sequential Example . . . . .	326
7.5.2	Partitioning MPI-IO/HDF5 Datasets . . . . .	333
7.6	Point Selection . . . . .	355
7.7	Compound Datatypes . . . . .	360
7.8	Iterating over HDF5 Groups . . . . .	367
7.9	What Else in HDF5? . . . . .	372
<b>8</b>	<b>Other Parallel Libraries</b>	<b>373</b>
<b>9</b>	<b>Databases: From Small to Huge</b>	<b>377</b>
9.1	PostgreSQL under Cygwin . . . . .	380
9.2	Talking to a PostgreSQL Data Base with <code>psql</code> . . . . .	384
9.3	Talking to a PostgreSQL Data Base from C . . . . .	391
9.4	Huge Data Bases . . . . .	395
	Index . . . . .	399



# Chapter 1

## Introduction

Welcome to I590, High Performance Data Management and Processing. This is our first experimental course that is going to make use of the Indiana I-Light infrastructure to deliver instruction to students at various IU campuses, most notably, IUN, IUPUI and IUB.

This course is a part of the AVIDD program, which has been financed to the tune of \$3.25 million by the National Science Foundation, which contributed \$1.8 million, IBM, which provided a Shared University Research (SUR) grant and by Indiana University, which paid the rest.

AVIDD stands for “Analysis and Visualization of Instrument Driven Data”, the analysis and visualization in question to be carried out on the four AVIDD clusters of IA32 and IA64 blades scattered amongst the three Indiana University campuses, IUN, IUPUI and IUB and coupled to the IU’s High Performance Storage System (HPSS).

### 1.1 The Focus of the Course

Using clusters for anything other than running lots of small jobs on the clusters’ individual nodes – this type of activity is nowadays called “capacity computing” – is highly non-trivial and calls for considerable computing skills. This non-trivial activity is often referred to as “supercomputing”, but it differs markedly from supercomputing *sensu stricte* and is more broadly applicable too. Whereas *real* supercomputing is concerned with number crunching first and foremost and is best done on highly specialized machines that are *not* based on common business CPUs, cluster “supercomputing” is often focused on manipulating very large amounts of data as in data mining, for example, or in managing and manipulating very large multi-tera-byte data bases.

A more general term, which is perhaps more appropriate, is “High Performance Computing” or HPC for short. HPC is not “super”. It is a notch below, but it is broader, both in terms of applicability and availability. Whereas supercomputers are rare, expensive (though not expensive in terms of \$/GFLOPS

– this is the main point about these highly specialized machines) and available to a handful of institutions only, HPC systems such as clusters of various sizes, including also clusters of SMPs, are very common. You will find them in laboratories, corporate computing rooms, and at the Internet providers’ shops, sometimes even in the hobbyists’ attics. Some vendors, e.g., IBM and Gateway, offer access to their own clusters to customers who need to use such facilities occasionally, but don’t want the bother of having to buy, configure and maintain them in their own machine rooms.

So, who is this course for? This course is for people who *need to* learn how to use clusters to manipulate very large data sets in the High Performance Computing style, not “chunk by chunk” but the whole lot, all, at once. I have emphasized the phrase “need to”. This is important, because such HPC data manipulation is not easy. There is a lot of difficult stuff you will have to learn. High Performance Computing on clusters is hard, tedious and wasteful of researchers’ time. If you can do your job in other ways, without having to use HPC techniques, do so. Resort to “capacity computing” if you can, if your particular problem type, data structures and data processing methodologies let you do this.

But if you’re interested in parallel data bases, data mining, or any other activity that is concerned with analyzing, processing and modifying a single gigantic data set, a data set that does not fit in the memory and on the disk of a single computer, then this is the course for you.

Now, if you are interested primarily in the “capacity computing”, I can hear you ask the question “Where am I going to learn how to do this?”. The point is that if you have the required level of skills for this course (see section 1.2 that talks about it) you already know how to do this. All you need here is to know how to edit, compile, link and run a UNIX program, and how to write simple UNIX shell scripts. You can then write a trivial shell script that distributes your program with arbitrary initial data to an arbitrary number of cluster nodes, or you can use the AVIDD batch queue system, PBS, to do the same in a way that is more considerate of other users (and less vulnerable to *skulker*). If you would like to learn how to do the latter, you can attend the course up to chapter 4 and then simply drop out.

## 1.2 Required Level of Skills

You may fear that given the difficulty of the subject you would be expected to have very highly developed computer skills in order to attend this course. Luckily this is not quite the case. The stuff we are going to talk about does not depend on object oriented programming, recursive functions, symbol manipulation, artificial intelligence, XML, Active Directory, Java and what not. This is not to say that you cannot combine these techniques and elements of IT with parallel programming in various ways. It’s up to you. But the basics of what we are going to talk about depend on plain C (or plain Fortran) only. And so plain C is the first pre-requisite for the course.



The other pre-requisite is UNIX. The AVIDD cluster is a Linux system. In order to work with it you will have to know how to compile link and execute programs under UNIX. You will have to know how UNIX is organized, how to connect to it, how to find information, how to manipulate UNIX files, etc.

The laboratories we are going to use for the classes are Windows/PC laboratories. This is because these are the most common computer laboratories at IU, because Windows/PCs are the most commonly used laptops and desktops and because we are going to use Microsoft NetMeeting a lot. So you need to know how to use Windows/PCs too. This is the last pre-requisite.

## 1.3 Bring Your Laptop

Although we will use UITS computer laboratories, such as may be available and suitable for this course, I do not assume that you are going to use these laboratories in your future research work. Rather I expect that you will use your office desktop, or your laptop, or your research laboratory server, or your departmental server. Furthermore, we cannot expect every software component we may need in the course to be available on these common use UITS PCs. Consequently, if you have your own laptop and would like to bring it with you and use in the course, you are welcome to do so.

I have remarked in the preamble that we may even consider delivering the course directly to your home or office – this especially if we don't have many students enrolled. In this case, obviously, you will *have to* provide your own PC (or a Mac).

## 1.4 Using NetMeeting

We are going to use Microsoft NetMeeting to deliver much of the course, especially to the remote audiences in Gary, Lafayette and, possibly, in Indianapolis too. NetMeeting is available with Windows 2000 and XP, but it is a little hard to find under XP, because it is no longer placed in the XP Programs menu.

To invoke NetMeeting under Windows 2000 bring up the following menu cascade:

```
Start
  Programs
    Accessories
      Communications
```

There in this last menu you will find NetMeeting. Simply select it and the application should pop up.

Under Windows XP you will have to go to

```
My Computer
  Fixed Disk (C:)
    Program Files
```

### NetMeeting

There you will find the icon called "conf". Double click on the icon to bring the application up.

If this is the first time you use NetMeeting on this computer, you will be asked some questions about who you are and which is going to be your NetMeeting server. Ignore the question about the server and answer the remaining questions as well as you can.

In the end you should get the application window up. There are various push-buttons in this window and a small video screen. If your computer has a camera attached to it and the camera is configured correctly, you should be able to click on the button with a little blue triangle in it. It is on the left hand side just under the screen. If you hold the pointer over this button for a moment, with the NetMeeting window selected, a little help balloon should appear with the words "Start Video" in it. If you click on this button you should get to see whatever the camera is looking at in the video screen.

To make a connection pull down the "Call" menu in the top left corner of the NetMeeting window and select "New Call". In the "To:" field enter either the full DNS name of the PC you want to connect to, e.g., woodlands.tqc.iu.edu, or, if the PC doesn't have a name, just its IP number, e.g., 129.79.15.18. In the "Using:" field choose "Network". Then press "Call". If the user on, in this case, woodlands.tqc.iu.edu, accepts your call the connection should be established.

In the bottom row of the NetMeeting window you will find push buttons, which activate various functions, e.g., "Chat" or "Share Program". We will use the latter to show you how to do various things on the AVIDD cluster. We will use the former to communicate in addition to the teleconference link.

You should take some time to play with NetMeeting and become familiar with it.

NetMeeting is a very useful tool. It is not a Microsoft's original invention. It builds on earlier freeware experiments such as "mbone", but it wraps it all up in a neat package and is altogether more seamless and easier to use. I used it on a number of occasions to deliver remote presentations from Bloomington to various business meetings in Houston, Chicago, Urbana-Champaign, Washington DC, and other remote locations. Although you can use it to transmit sound, in principle, it is not very suitable for this. NetMeeting sound is too choppy. Consequently, the best way to use NetMeeting is to complement it with a teleconference.

## Chapter 2

# Supercomputers and Clusters

### 2.1 The Almighty PC

Present day PCs (and Macs) can be amazingly powerful and *relatively* inexpensive.

I have emphasized the word *relatively* because you should not expect computer vendors not to be after money in your pocket. And so, they keep prices of their PCs always at the higher rather than lower side by a variety of means, which include closing down older model lines and rolling out machines with faster CPUs, more memory, more disk space, and larger displays, whether you, dear user, need it or not. They have to keep their prices sufficiently low to lure you into their shops, but not so high that you would find their equipment unaffordable. There is some competition in this business that helps keep prices down, but not enough. Nowadays we are down to two major US PC vendors, Dell and HP, with Gateway, IBM and Apple mopping up crumbs that fall off the table, two Japanese vendors that count, Toshiba and Sony, and Taiwanese Acer.

Compare this to the glorious world of automobiles, where you have three major US vendors, six Japanese vendors, three Koreans, three great German companies, three French, two Swedish, four Italian and a plethora of British exotica. Even though some of these have been bought by others, their distinct production lines survive and compete offering a great variety of products to people around the world.

But let's get back to computers. Whatever competition there still is in this business, is enough to keep pushing computer performance up and their prices down enough to provide us with laptops with 3 GHz CPUs. I saw some people write that their laptops had as much computing power today as supercomputers had only ten years ago. Such statements make great advertising lines, but they are quite incorrect. They are incorrect, because it is only the CPUs themselves

that have computational power matching that of ten years old supercomputers. Other PC components, like memory, system bus, IO interfaces, etc., lag well behind those of even ten years old supercomputers, and so these laptops, although quite useful, are not really in the supercomputer league. But they can perform certain operations very quickly, especially the ones for which they are highly optimized: fast screen redraw, fast searches on cached data, fast handling of interrupts – the stuff that is important to PC users.

## 2.2 Supercomputers and Clusters

PCs can be delivered in various packages: desktops, laptops, desksides, blades (for rack mounting), raw motherboards for embedded systems, headless boxes for network management, and even wearable computers. A substantial number of PC blades can be packed into a rack (up to 48 per rack) and linked with a high bandwidth communication network to allow for a single very large computation to be laid out on such a distributed system.

About seven or eight years ago some people came to the conclusion that this would be the future of supercomputing, although initially they thought more in terms of commodity processors other than IA32, e.g., POWER, Alpha, SPARC or PA-RISC. Because important decision makers were amongst them, that view had dire ramifications for the US supercomputer industry, which in effect all but died overnight.

This made PC and UNIX workstation vendors deliriously happy, because now they had this whole end of the market to themselves (pity they knew so little about it) and they could sell a lot of rack mounted PCs and workstations to universities, defense laboratories, exploration companies and design bureaus.

This made programmers and computer scientists very happy too, because they knew better than anybody else how difficult such systems would be to assemble, manage, use and program and so they sniffed many lucrative life-long sine-cure jobs for themselves.

The only people who were unhappy about this turn of events were supercomputer users, who were suddenly faced with the demise of highly efficient and easy to use machines, of various architectures, which they relied on to do first class science without having to waste a lot of their own time on programming equilibristics.

Three Japanese companies, Fujitsu, NEC and Hitachi, continued to build real supercomputers, some based on vector other on scalar CPUs, but all quite nicely balanced and with excellent IO and memory bandwidth, but they were barred from the US market by legal means. They have been selling their machines in other countries with considerable success though.

And so things progressed at their own pace. Some applications were ported to this new distributed programming model, some were abandoned, and some new applications were developed too. Many people made good living out of it, but, by and large, a lot of tax payers money and scientists time got wasted in the process. Some of this money fed the dot-com boom of late 90s.

And then the Japanese commissioned the Earth Simulator, a system that outperformed ten most powerful US clusters put together, was very efficient and childishly easy to program, at least in comparison with American PC clusters. This caused a great stir and uneasiness amongst the aforementioned important decision makers as they were finally forced to see for themselves, what every computational scientist was telling them all along: that PC clusters were no match for real well designed supercomputers.

They responded in their usual way: they assembled numerous committees to scrutinize the current context, reassess the US position in supercomputing and to outline future directions. And so even more tax payers' money got wasted. But eventually DARPA announced the High Productivity Computing Systems Program and awarded \$50 millions each to Cray, IBM and Sun for the development of novel supercomputer architectures that would scale to PFLOPS.

The systems proposed by these three companies are *not* PC clusters. They are very, very different.

The Cray's Cascade system derives from the earlier PFLOPS architecture developed for Defense some five years ago as well as from the Cray MTA multi-threaded architecture machine, previously known as "Tera". Cascade is being developed jointly with researchers from JPL, Caltech, Stanford and Notre Dame. The Cascade machine will combine a lot of earlier ideas such as UMA and NUMA SMPs, hybrid electrical and optical interconnect, lightweight threads in memory and aggressively parallelizing compiler technology. But perhaps the greatest innovation is going to be the use of processor in memory (PIM) chips. Amongst the greatest handicaps of present day systems is movement of data from memory to CPUs. This problem can be overcome by bringing CPUs to data and computing directly in the memory. This is how PIM chips work.

IBM's proposed machine is called PERCS, which stands for Productive, Easy-to-use, Reliable Computing System. The system will reconfigure its hardware dynamically, on the flight, to suit a problem at hand. IBM has the ambition to make it into a new commercial computing architecture that may deal a blow to the PC and PC clusters. This system will use some off-the-shelf components, most notably IBM's POWER5 (or later) processors, but much of the rest is going to be quite new. They may use PIMs too.

Sun plans to base its machine on a quite revolutionary concept of clockless computing. Computation in clockless computing unfolds at almost analog speed, though it is still digital, but it is not halted by constant references to the system clock. Every computational process runs as fast as it possibly can and stops only when it needs to communicate with other processes.

But none of these three systems is going to be delivered any time soon, and when they eventually do get to see the light of day, they'll be restricted to defense facilities, national laboratories and, perhaps, national supercomputer centers. It is unlikely that you'll see any of these machines (perhaps with the notable exception of the IBM system) in your laboratory or on your desktop.

In the meantime the Earth Simulator remains the most powerful, the most computationally efficient and the easiest to program of all supercomputers on the planet. But, guess what... it is a cluster too. It is a cluster of 640 super-

computers (NEC SX6), each of which is a 64 GFLOPS system. Because the nodes of Earth Simulator are real supercomputers in their own right, not PCs, the system delivers very high computational efficiency of between 40% and 60% on production applications. It is this very high efficiency, more than its peak performance, that made the US computational scientists stop and think.

Yet, the fact that the Earth Simulator is a cluster too is actually good news. It means that whatever you are going to learn in this course is going to be applicable not only to PC clusters, but also to systems such as the Earth Simulator. It is going to be applicable to Cray X1 (and its successor the Black Widow), to Cray Red Storm, to IBM SP, to clusters of IA64s and Opterons, and even to IBM PERCS discussed above.

The computational methods and paradigms we are going to discuss in this course are universal enough to be applicable to a very broad range of systems. For this same reason we will stay away from any IA32 specifics.

### 2.3 HPC Facilities at Public Supercomputer Centers

Another result of decisions made in mid-1990s, apart from the near-death experience of the US supercomputer industry, was that American researchers ended up with very limited access to well engineered supercomputers in their own country.

Perhaps the only public computing facility that has been providing access to such systems over the years is the Arctic Region Supercomputer Center (ARSC) in Fairbanks, Alaska. Today the center operates a 272-processor Cray T3E, a 32-processor Cray SV1ex, an 8-processor NEC SX6 (which is re-badged for the US market as Cray SX-6), and, the most recent acquisition, a 128-multi-streaming-processor Cray X1. The latter is a highly efficient massively parallel vector system, with 1.6 TFLOPS peak performance. This system should deliver between 40% and 60% efficiency on production codes, which would make it equivalent to, say, a 20 TFLOPS IA32 cluster. (They are going to install just such a 20 TFLOPS IA32 cluster at NCSA, see below.) The Cray X1 at ARSC is equipped in about 1 Peta Byte storage in the form of Sun disk array servers and Storagetek tape silos (we have a similar storage system here at Indiana University).

There are also some US Army computer centers that have well engineered machines, but they are largely inaccessible to US researchers on account of being dedicated to military work, so we won't talk about those.

The three National Supercomputer Centers, Pittsburgh, Urbana-Champaign, and San Diego, offer various clusters of scalar CPUs only.

Pittsburgh has a 512-processor Cray T3E and a cluster comprising 750 4-way Alpha SMPs.

NCSA in Urbana-Champaign has (or is going to have soon) two IA32 clusters, one with 484 2-way SMP nodes dedicated to computation and 32 2-way

SMP nodes dedicated to storage, and another one with 1,450 2-way nodes. The latter is going to yield (when it is finally installed) 17.7 TFLOPS peak, which may translate to perhaps about 1 TFLOPS deliverable on production codes – close enough to the ARSC’s Cray X1.

You can think of our AVIDD cluster as being a smaller version of this large NCSA IA32 cluster. Codes developed for AVIDD should run almost without change on the NCSA 17.7 TFLOPS system.

NCSA also has two IA64 clusters, one with 128 2-way nodes dedicated to computation and 4 2-way nodes dedicated to storage, and another one with 256 2-way nodes.

Finally SDSC (San Diego) has an aging IBM SP (it is a cluster too) with 1,152 POWER3 CPUs. But SDSC is in alliance with the University of Texas and the University of Michigan, which contribute some fairly sizeable systems of their own to the pool. For example, the University of Texas contributes an IBM SP with 224 POWER4 CPUs, which is only a little less powerful than the SDSC system, and the University of Michigan contributes four clusters, which altogether add up to almost as much as the SDSC SP.

All in all these facilities are somewhat disappointing. Developing parallel programs for the clusters of scalar CPUs is very tedious, debugging is cumbersome and takes ages, and, worst of all, the codes more often than not end up running very slowly and generating more heat than good science.

Because the fuddy daddies who got us into this morass are still at the helm of the sinking boat, you should not expect things to change any time soon. It’s going to be more of “steady as she goes”, until the DARPA project bears fruit and we get some new toys to play with.

This situation gave rise to some concern and pointed questions in the Congress, where Vincent Scarafino, the manager of numerically intensive computing at Ford Motor, commented: “The federal government cannot rely on fundamental economic forces to advance high-performance computing capability. The federal government should help with the advancement of high-end processor design and other fundamental components necessary to develop well-balanced, highly capable machines. U.S. leadership is currently at risk.”

The chairman of the congressional committee that looked into these matters, Sherwood Boehlert, a representative from New York, stated that “Lethargy is setting in [at the NSF] and I’m getting concerned. I don’t want to be second to anybody.”

## 2.4 IU HPC Facilities

Indiana University is remarkably well equipped in HPC systems. You can view a list of what is available at <http://www.indiana.edu/~rats/>.

Our main research system is the IBM SP. It is a cluster system connected by a special high bandwidth switch network, which provides point-to-point connectivity at 150 MB/s between a variety of nodes, beginning with 136 4-way POWER3+ SMPs, 4 16-way POWER3+ SMPs, and one 16-way POWER4

SMP. This system also has a small number of other nodes, which “don’t quite fit” and are used for auxiliary tasks. The General Parallel File System installed on our SP provides nearly 3 TBs of storage. All CPUs mentioned here are 64-bit CPUs, which can perform up to 4 double precision floating point operations in a single clock cycle, albeit on special operations (add-multiply) only.

The whole system is gauged at about 1 TFLOPS peak. Because its nodes are quite powerful SMPs (and the 16-way nodes are *very* powerful SMPs), the system can be used to run a large number of shared memory jobs and this makes it extremely useful in our environment.

The next system is made of the four AVIDD clusters. These are the clusters that we are going to use in this course. There is a lot of spare computational capacity on this system at present and, because it is so similar to the NCSA clusters, it can be also used as a training and testing ground for the latter. Its primary purpose is “Analysis and Visualization of Instrument Driven Data”, hence its name, AVIDD, and the rationale for this course, “High Performance Data Management and Processing”.

We are going to dedicate the whole chapter 3 to this cluster, so we are not going to spend more time on it here.

Although we have HPC clusters at IU, we don’t have “real supercomputers”, like Cray X1, NEC SX6, Cray T3E or Hitachi (University of Tokyo has one of these that can run LINPACK at 1.7 TFLOPS) for the following reason. They are very good for certain types of *typical* supercomputer jobs, but no good for a very broad range of jobs. At our university we might perhaps have two or four users who would benefit from such systems, but they would be pretty useless to the vast majority of our users, who are *not* typical supercomputer users. On the other hand clusters, although not efficient as supercomputers, can be used to run just about anything, beginning with little sequential scalar jobs, or a lot of little sequential jobs, and ending on parallel jobs of various sizes: some requesting 16 CPUs only, some running on 64 2-way nodes.

## 2.5 Parallel Programming

### 2.5.1 Data Parallel Paradigm

The simplest parallel programming paradigm is data parallelism.

If you have, say, two one-dimensional arrays  $\mathbf{A}$  and  $\mathbf{B}$ , of equal size, say,  $N$  double precision floating point numbers each, the traditional way to add such arrays in the C-language would be

```
for (i = 0; i < n; i++) c[i] = a[i] + b[i];
```

You’d do it similarly in Fortran-77 or Pascal or any other *sequential* programming language. But in Fortran-90 you can simply state:

```
C = A + B
```



which corresponds closely to how you would write it in mathematics:

$$C = A + B$$

The operation of adding two arrays is explicitly parallel, i.e., you can perform each addition  $c[i] = a[i] + b[i]$  independently. If you run a Fortran 90 compiler on a Cray X1 or NEC SX6, the compiler will automatically convert program lines such as  $C = A + B$  into parallel operations that will execute  $c[i] = a[i] + b[i]$  simultaneously (or almost simultaneously – depending on the architecture of the machine) for all values of  $i$ .

A lot of scientific and engineering computing can be expressed efficiently in this data-parallel paradigm. All field problems like electrodynamics, fluid dynamics and chromodynamics fit in this category. Weather and climate modeling codes do too. Gene matching and searching codes can be expressed in terms of data parallelism as well. Basically anything that operates on very large arrays of data submits to data parallelization easily.

Data parallel languages provide various constructs for conditional differentiation of some operations on various parts of arrays, e.g., masking, and for shifting the content of the arrays in a way that is similar to shifting the content of the register, i.e., you can shift arrays left and right, you can rotate them, or shift with padding.

Data parallel programs are very easy to understand and to debug because their logic is essentially sequential. But it is also this sequential logic that results in some inefficiencies. Suppose you want to set values in the  $B$  array to zero for all such  $i$ s for which  $c[i] == 1$ . The Fortran-90 statement to do this is

```
where (c .eq. 1)
  b = 0
end where
```

If  $c[i] == 1$  for just a handful of  $i$ s, this operation will be performed on only a handful of processors that are responsible for the  $b[i]$  involved, whereas all the other processors will idle for the duration of this operation. If the operation is very involved, e.g., there may be some very complex and long computation going on within the confines of the `where` statement, the idling processors may end up doing nothing for a long time.

Nevertheless, there is a lot to be said in favor of data parallelism. It is much more universal than many computer scientists are prepared to admit, and because it is so easy to use, it should be provided to supercomputer users, especially the ones writing their own codes. Data parallelism is very highly scalable too. We will probably see the return of data parallel computing in context of petaflops systems. The PIM (Processor in Memory) architecture is very similar to the Connection Machine (see below).

Data parallel programs run best on vectors and massively parallel machines, like the the Connection Machine. They can run, although not as efficiently, on SMPs and clusters too, but then, not much will run on clusters efficiently anyway. There is a special variant of data parallel Fortran for clusters, called

High Performance Fortran (HPF). The best HPF compilers can be bought from The Portland Group Compiler Technology.

For more information on HPF see, e.g., the High Performance Fortran page at the Rice University. Also see a brief tutorial about HPF, which was included in the P573 course.

### 2.5.2 Message Passing Paradigm

Message passing parallel programming paradigm is very flexible, universal, can be highly efficient, and is absolutely ghastly to use. But since the former three outweigh the latter, message passing wins and is currently used by most parallel production codes.

As was the case with data parallel programming, message passing programming is a paradigm and can be implemented on systems with various architectures. You can use message passing on clusters, on SMPs, even on single CPU machines and on fancy supercomputers like Cray T3E and Cray X1.

In message passing paradigm your computer program can be logically split into as many different processes as you need. You can even have more processes than you have CPUs, but usually you try to match these two numbers together. The processes can all run quite different codes and they can run on CPUs that are geographically distant. For example you can have one process run in Bloomington and another one run in Gary.

The processes exchange data with one another by sending messages. A process can send a message in such a way that it doesn't care if the message has been received by anybody. So the messages that have been sent, don't need to be received. But usually they are and there are special function calls for receiving messages too.

The most commonly used library for message passing is called the Message Passing Interface. Two popular freeware implementations of it exist, one developed by the Argonne National Laboratory, called MPICH, and another one developed by the Ohio Supercomputer Center, but currently maintained by the LAM organization and the LAM team at . . . Indiana University.

MPI is huge. There are hundreds of functions in it for doing various things. We are going to study and use some of them in this course, with special emphasis on functions for doing parallel IO.

Because MPI is so flexible and universal, data parallel languages for clusters are often implemented on top of MPI, i.e., the compiler automatically converts a data parallel code to an MPI code. This is how the Portland Group HPF compiler works and this is how the IBM HPF compiler for SP works too.

### 2.5.3 Shared Memory Paradigm

The shared memory paradigm is really restricted to shared memory systems, SMPs, although it is possible to simulate it on distributed memory systems too, albeit not without some cost. The basic idea here is that you can have a normal UNIX program, which spawns threads. The threads are each given separate

CPUs to run on, if such are available. They can communicate with each other by writing data to shared memory and then reading data from it.

Recall that when a traditional `fork` is called under UNIX, it copies the whole process into separate memory location and the processes, in principle, don't share memory. But threads do.

Some operating systems, e.g., Dynix, used to let forked processes share memory too.

Communicating through shared memory can be very fast, but it can be also quite troublesome. To begin with, what if two processes both try to write on the same memory location at the same time? The result of such operation would be unpredictable. So the shared memory paradigm introduces a concept of a memory lock. A process that wants to write on a shared memory location or that wants to read from it, can lock it, so that no other process can access it for writing at the same time.

But this solution causes other problems, of which the possibility of a deadlock is most severe. A deadlock can occur if process *A* locks a memory location, forgets to unlock it or perhaps just postpones to unlock it for some reason and goes to write on another memory location, which just has been locked in the same way by process *B*, which is currently waiting for the memory location on which *A* has just written to get unlocked before it will unlock the one *A* is waiting for. Both processes will end up waiting forever. This is a deadlock.

Deadlocks can be avoided if certain rules are followed, but they still happen nevertheless.

Message passing programming can be implemented very efficiently on top of shared memory. In this case chunks of shared memory are used to store messages. Message sending is implemented as writing to the chunk, and message reading is implemented as reading from it. If this is done right, deadlocks can be avoided.

Data parallel programming can be implemented on top of shared memory too, with multiple processors of an SMP taking care of separate chunks of arrays that reside in shared memory. So SMPs are very good for Fortran 90 programs.

Various utilities exist that simplify shared memory programming, so that you don't have to spawn threads explicitly. For example, OpenMP lets programmers parallelize loop execution by inserting compiler directives in front of the loops. This helps programmers parallelize legacy applications that were written in sequential languages like C and Fortran 77.

#### 2.5.4 Mixing Parallel Programming Paradigms

You can mix various programming paradigms in the same program. For example, you can use Fortran 90 in order to express data parallelism on an SMP node, and then mix it with MPI library calls in order to make processes running on SMP nodes communicate with each other. If the problem-on-the-node is not data parallel, but can be parallelized by some other means, you can resort to multi-threading on the node and MPI communication between the nodes, or you

can multi-thread implicitly by inserting OpenMP directives in your on-the-node program.

Programs written by mixing explicit multi-threading and MPI are perhaps the most horrible to analyze and write. Yet they are the ones that are going to run most efficiently on clusters of SMPs.

### 2.5.5 Scalability of Parallel Programming Paradigms

Parallel programming and parallel program execution can be often very disappointing. You may invest a lot of very hard work in a project, only to discover that your parallel program doesn't run any faster than its sequential sibling. The reason for this is that the cost of communication, especially the cost of communication in a cluster, is extremely high. So you always have to balance amount of work to be carried out on a single node, versus amount of data that has to be exchanged between the nodes. The more work on the node and the less data to exchange, the better.

Communication problems are also at the root of poor scalability of parallel programs. Even if your program does perform up to the expectations, you may discover that your speed-up will no longer be satisfactory when the number of nodes, the program runs on, gets too large. Most production codes are written to run on 16 or 32 nodes maximum. Some may still scale to 64. But I haven't seen many production codes that would run well on, say, 1024 cluster nodes. The cost of communication and synchronization for this many nodes would be prohibitive.

The present day parallel programming paradigms, perhaps with the notable exception of the data parallel paradigm, are not scalable beyond tens, or at best low hundreds, of processes.

But the data parallel paradigm can be laid out on machines with tens of thousands of CPUs. For example, the Connection Machine CM2 used to have up to 64,000 CPUs. This paradigm, as I have remarked above, should map on the PIM systems too. We will almost certainly see some form of it on the petaflops systems that are currently being developed.

## 2.6 HPC and The Grid

The latest fashion in some academic IT circles is "The Grid", and many people have a quite incorrect view that "The Grid" in some way is deeply connected to high performance computing, or even that it *is* high performance computing in its latest guise.

"The Grid" is *not* related to high performance computing.

High performance computing and supercomputing have been around for tens of years without "The Grid" and they will continue to be around for tens of years without it.

The other view, which is also quite incorrect, is that there is just one "Grid", which is also "The Grid", and that this grid is based on "Globus", which is a

collection of utilities and libraries developed by various folks, but mostly by folks from the University of Chicago and the Argonne National Laboratory.

Many people, who actually know something about distributed computing, pointed out that what is called “The Grid” nowadays, was called “Distributed Computing” only a decade ago. It is often the case in Information Technology, especially in academia, that old washed out ideas are being given new names and flogged off yet again by the same people who failed to sell them under the old names.

There are some successful examples of grids in place today. The most successful one, and probably the only one that will truly flourish in years to come, is the Microsoft “.NET Password” program. It works like this: when you start up your PC running windows, “MSN Messenger” logs you in with the “.NET Password”. This way you acquire credentials, which are then passed to all other WWW sites that participate in the “.NET Password” program. For example, once I have been authenticated to “.NET Password”, I can connect to Amazon.com, Nature, Science, Monster, The New York Times, and various other well known sites, which recognize me instantaneously and provide me with customized services.

Another example of a grid is AFS, the Andrew File System. AFS is a world-wide file system, which, when mounted on a client machine, provides its user with transparent access to file systems at various institutions. It can be compared to the World Wide Web, but unlike WWW, AFS provides access to files on the kernel and file system level. You don’t need to use a special tool such as a WWW browser. If you have AFS mounted on your computer, you can use native OS methods in order to access, view, modify, and execute files that live at other institutions. User authentication and verification is based on MIT Kerberos and user authorization is based on AFS Access Control Lists (ACLs).

Another example of a grid is the grid that is currently being built by CERN and that is going to be used by Europe’s high energy physicists and their collaborators. They have their own highly specialized protocols, libraries and utilities that are built on top of “Globus”, but the latter is used as a low-level library only. Recall that it was CERN where WWW was invented in the first place.

Another example of non-Globus Grid software developed in Europe is Unicore. It was developed by Forschungszentrum Jülich, GmbH in cooperation with other German software companies, the supercomputer centers in Stuttgart, Munich, Karlsruhe, and Berlin, the European Centre for Medium-Range Weather Forecast in Reading, UK, and various hardware companies, such as Fujitsu, Hitachi, NEC, Siemens, as well as some American partners, HP, IBM and Sun Microsystems.

Europeans want to do things their own way and for various reasons eschew being dependent on American technology, including Information Technology.

A yet another example of Grid software is Legion, developed by researchers from the University of Virginia under various contracts with DoE, DoD and DARPA. Legion is much more usable and more functional than Globus and provides numerous higher level utilities and abstractions. An insightful comparison between Legion and Globus can be found in “A philosophical and technical

comparison of Legion and Globus” by Grimshaw, Humphrey and Natrajan, IBM Journal of Research and Development, vol. 47, no. 2.

Why grid and high performance computing are not the same thing?

The purpose of the grid is to provide users with connection to various computing and storage resources usually at other institutions.

There is no need for grid protocols within an institution, because other site-wide authentication and verification methods such as Kerberos or Active Directory work better in this context. So grid is for long-haul connections.

Long-haul connections *always* have very high latencies. This is caused, first, by the speed of light, which by supercomputer standards is very low, and, second, by the fact that you have to pass through numerous routers, switches and sometimes even firewalls on your way between the institutions and they add even more latency to the connection, while in some cases restricting the available bandwidth severely (e.g., the firewalls). High latency kills high performance computing. The only jobs that are relatively immune to it are ”high capacity computing” jobs, i.e., trivially parallelizable jobs, which run as numerous small programs on a large number of computers and which don’t communicate with each other. But ”high capacity computing” is not high performance computing, neither is it supercomputing, where intense communication between processes running on various parts of the system is frequent and often synchronized.

The I-light link between Bloomington and Indianapolis is not ”as crow flies”. It goes around quite a lot and its total length is about 80 miles. Were it not for switches and routers, it would take about 0.5 ms for the light signal to traverse this distance. In reality it takes much longer. But let’s stay with this ideal number of 0.5 ms. In this time a 1 TFLOPS supercomputer can perform 500,000,000 floating point operations. If a program running on an IA32 cluster at IUB and IUPUI has to synchronize operations frequently, every time it needs to do so we’ll lose billions of floating point operations waiting for the synchronization to complete.

The other problem with long-haul connections is that effective bandwidths on such connections for large data transfers are usually very low, even if the lines are advertised as high bandwidth ones and even if there are no firewalls. It is enough that a very small fraction of packets get dropped on some routers to bring the effective transfer rates down from the nominal hundreds of MB/s to mere five or so. We have seen some long-haul transcontinental transfer rate records established recently. For example, between SLAC and the Edinburgh University (40 MB/s) and between Caltech and CERN (80 MB/s). But 40 MB/s or 80 MB/s is very little by high performance computing standards. Here we need transfer rates of GB/s or better.

But let’s get back to ”The Grid”. Assume that we have it in place. ”The Grid” will provide us with tools to access, say, NCSA, SDSC, PSC, ARSC and some other supercomputer centers. The tools in question may even be quite nice given another 20 years of development. You’ll click some push buttons, turn some dials and... you’ll be there. Now what? You got connected to, say, NCSA, and you still have to write a supercomputer program to run on the NCSA cluster utilizing all its nodes in parallel and communicating frequently

between various processes. “The Grid” is of no help here. It doesn’t tell you how to do this.

This course, however, will. Think of this in the following terms: this course is about what you need to do *once* you have connected to a supercomputer resource. You may have connected using “The Grid”, or using ssh, or using Kerberized telnet, or using simply a telephone line. But now you are there and you have to reach for tools other than ssh, “The Grid”, or the telephone line, in order to construct, submit and run a supercomputer job.

## 2.7 To Program Or Not To Program

To program or not to program?

The answer is, *not* to program, if you don’t have to. Programming is a tedious activity and, unless you are a professional programmer and make your living doing this, there is nothing for you to gain from such an exercise. Because of the tedium involved and because of the enormous amount of labour required to write even simple applications, programming jobs, like blue-collar jobs, end up going overseas to cheap labour countries, most often to India nowadays. So, even if you have been thinking about becoming a professional programmer, think again and revise your options.

If you can solve your research problems by running an off-the-shelf (commercial or free) application on your laptop or desktop PC, do so. Focus on your research mission, not on computing. The latter is only a tool and it is only one of many you will have to use in your research.

If your laptop or your desktop PC don’t have enough power to solve your computational problem, use your laboratory or your departmental server. A recent review of how US researchers use computers showed that their laboratory and departmental servers are by far the most important systems in their work.

If your laboratory server is not powerful enough to solve your computational problem, use central computing facilities provided by your university, or, if your university doesn’t provide such, use systems provided by the national supercomputer centers. This is what they are for.

In all cases try to use off-the-shelf software if available. Only if your problem is so exotic that there is nothing out there in the software world to help you out, you may really have to sit down and write your own program. But before you do, think again. Can your problem be solved by laboratory experimentation? Think of Nature and your laboratory bench as a computer that is much faster and much more accurate than any man-made computer. Can you solve the problem analytically, without having to resort to numerics? Laboratory and analytical results are always going to be more convincing than numerical simulations.

Most research domains have developed various codes for reduction of experimental data in their related disciplines. For example, astronomers have the IRAF package that they can all use to process astronomical images. High energy physicists have numerous applications for data processing developed at and distributed by CERN. Geneticists have developed numerous codes related to their

work too. There are commercial engineering codes for just about everything that engineers work on nowadays. There are codes for car crash simulations, codes for designing and testing designs of integrated circuits, codes for structural engineering, water engineering and what not.

But let us go back to the case mentioned above. You're stuck, you have to carry out some complex numerical computation, there is no off-the-shelf code that you can use (you have checked, haven't you?), there is no laboratory procedure you can resort to in order to attack the problem and the problem is analytically intractable (you asked your friend, who is a mathematician and she told you so; never mind you weren't nice to her when she was your girlfriend. . .). So, what are you to do?

The first thing you should try is to use one of those so-called problem solving environments like Matlab or Mathematica. They are designed to minimize programming effort and to maximize problem solving efficiency. With these environments you can probably attack any problem that can be solved on your own laptop, desktop or laboratory server.

If this doesn't do, if the computation is going to be truly massive, or if the computation is of the data base variety (Matlab and Mathematica don't do data bases), only then look towards SMPs or clusters. But look towards SMPs first. They are easier to use. They are fabulous data base servers. You can run data parallel programs on them.

Alternatively if you can attack your problem by "capacity computing", i.e., by running a lot of relatively small jobs on separate machines, use a cluster. A lot of stuff is tractable by "capacity computing". High energy physics resorts to "capacity computing" for most of its data reduction procedures. SETI, Search for Extra Terrestrial Intelligence, is another example. Car crash analysis: if you have to simulate car crashes under every possible angle, distribute the jobs over a cluster.

If neither of the above applies, an SMP is not powerful enough, your problem cannot be attacked by "capacity computing", well, how about getting an account on the ARSC's Cray X1 and trying to solve the problem by data parallel means on that machine?

Do I hear it right? Are you saying that your problem is huge and that it is not data parallel and that it cannot be solved by laboratory experimentation and that it is analytically intractable and that there is no SMP around powerful enough, and that it cannot be tackled by "capacity computing" either?

Let me ask this: "Is there enough money in it to bother?" And if there is, are you going to get any of it?

To be frank, the only problem of this type that I know of is data mining and parallel data bases, i.e., operations on gigantic data bases that are so large that they just can't be squeezed into a single SMP, however large.

Most computational problems that currently run on clusters deployed at the National Supercomputer Centers would probably run better on systems such as Cray X1 or the Earth Simulator. But parallel data base problems and data mining problems would not.



So, this is a yet another rationale for this course, “High Performance Data Management and Processing”.



## Chapter 3

# Working with the AVIDD Cluster

*AVIDD est omnis divisus in partes quartas quarum una in Bloomington, alia in Indianapolis, tertia in Gary locatur. Pars quarta quoque in Indianapolis locatur, sed restricta est.*

This is how Julius Caesar might have described the AVIDD cluster before vanquishing it, burning it to cinder and enslaving its administrators: it is made of four parts, of which the first resides in Bloomington, another in Indianapolis and the third one in Gary. The fourth part lives in Indianapolis too, but is accessible to IU Computer Science researchers only.

The IUB component of the cluster has 97 computational nodes, 4 file serving nodes (too few for a cluster of this size and, especially, for a cluster dedicated to IO, and 3 head nodes. The IUPUI component is similarly configured, i.e., it has 97 computational nodes, 4 file serving nodes and 3 head nodes.

The IUN component has only 8 nodes. It is supposed to be used for teaching, e.g., for a course like I590, but I cannot reach it from Bloomington (it may be fire-walled) and so I didn't have a chance to look at it yet.

All AVIDD nodes we are going to work with are 2-way IA32 (Pentium 4) SMPs and run Linux 2.4. So, in effect, we have 194 CPUs dedicated to computations on each of the two major IA32 components and you can submit jobs that span both components, utilizing all 388 CPUs. These CPUs run at 2.4 GHz, and their floating-point unit registers are 128-bit long. They support streaming SIMD (Single Instruction Multiple Data) instructions. This means that you can perform, in theory, up to 931.2 quadruple precision floating point operations per second on the whole 2-component cluster, or double that in double precision, if you pack your instructions and data movement within the floating point processor very cleverly.

This performance will collapse dramatically if you have to feed your data into the floating-point unit from the memory of the computer, because, first, the memory bus is only 32-bits wide, and second, it's going to take great many

cycles to move a 32-bit wide word from memory to the CPU. Nevertheless, our very clever computer scientists managed to wrench more than a TFLOPS performance from the combined 4-component cluster on a LINPACK benchmark. On a normal off-the-shelf parallel application, running on the whole cluster, you should expect about 5% of this, i.e., about 50 GFLOPS. With a lot of optimization you may be able to get it up to about 80 GFLOPS. The trick is to load as much data as possible into these 128-bit long registers of the floating point unit, and then keep the data there as you compute on it, while moving data in the background between memory and higher level caches.

We will not go into any of this, because it is a complete waste of time. . . unless you are a well paid professional programmer optimizing a commercial application for a commercial customer.

Always read the small print at the bottom of the page!

This is where you may find some really interesting stuff, like the fact that I am an alien in a human shape, with a body made of organic stuff that is quite worn out by now so that it is discolored, wrinkled and falls off in places. We arrived here in the Solar System shortly after what you call World War II and have been here ever since. My real body looks a little like a large octopus and was engineered to be resistant to vacuum, high radiation doses, temperature extremes and weightlessness, and right at this moment it is floating in front of a servo-robot manipulator in a spaceship hidden at the Lagrange point behind your moon. The manipulator is tele-linked to the body you see in front of you in the class. There is a slight delay between my instructions and the body's responses that may seem to you like your professor has slow reactions and is absent minded. Since this is typical of your faculty in general, we never had to do much about it.

### 3.1 Your AVIDD Account

There are two ways to obtain an account on the AVIDD cluster.

If you are an IU faculty or staff, connect to <https://itaccounts.iu.edu/>. Then click on “Faculty or Staff”, click on “Create more IU computing accounts. . .” too and press “Continue” at the bottom of the page. The system will bring up a login page. You will have to type your IU-wide user name and your IU-wide password and press the “Continue” button again.

If you do not have an IU-wide user name and password, go back to <https://itaccounts.iu.edu/> and request an IU-wide account for yourself. You won't be able to do much, in terms of IT at IU, without it. Of course, depending on the nature of your work at IU, you may or may not need access to IU IT facilities. It is quite possible to function without IT altogether, or to use external IT facilities, e.g., Microsoft's Hotmail or AOL, and your own private laptop. If you are in this category you can still obtain an AVIDD account. But in this case you will have to contact AVIDD administrators directly. To do so send a courriel (“courriel” is an official French term for e-mail – Pretentious? Moi?) to George W. M. Turner, who is really Santa Claus, but he moonlights as the AVIDD administrator.

After you've been logged on, you'll be transferred to the “Account Management Service” page, where you'll find a couple of choices. Choose “create more

accounts”. This will transfer you to a yet another page titled “Select account to request”. There, near the top, you’ll find the “AVIDD Linux cluster” entry in the “Research only Systems Group” . Click on it, type your IU-wide network password again in the field at the bottom of the page and press the “Create Account” button.

This procedure will *not* work, if you are an undergraduate student, because the AVIDD cluster will not be listed for you. In this case you need to go to <http://www.indiana.edu/~rats/application.shtml> and fill the form there. In the “Additional comments for processing” section specify the name of the course, 1590. The account will be valid for the duration of the Fall Semester 2003 only. The account creation process will take one business day to complete if everything is in order.

## 3.2 Connecting to the AVIDD Cluster

The only way to connect to the AVIDD cluster from your desktop or laptop is to use the so called “secure shell” and related utilities.

Standard telnet and Kerberized telnet connections to the cluster are disabled, as are the Berkeley r-tools, i.e., rsh, rlogin and the like. FTP and Kerberized FTP do not work on the AVIDD cluster either, so you cannot transfer your data to it this way. You will have to use scp instead.

So in this section we’re going to learn about using slogin and scp.

Slogin and scp are commonly distributed with Linux. But they are not distributed with Windows and proprietary UNIXes. I don’t know about MacOS-X. Most researchers and students working with AVIDD will be either Linux or Windows users anyway. If you’re a Linux user, you probably know all about slogin and scp already, so you can skip most of this section and go to chapter 4 right now. The only thing you need to know is that the front-end node on the AVIDD cluster is called `bh1.avidd.iu.edu`.

If you don’t know all about slogin, or if you’re a Windows user and do not know anything about it at all, read on.

The best way to get hold of slogin and scp under Windows is to install Cygwin. Cygwin is a full Linux emulator that runs under Windows. In some ways it is even more than Linux, because it provides its users with seamless access to all Windows facilities and then gives them almost all of Linux on top. Cygwin even provides access to NTFS ACLs (the Cygwin commands that manipulate NTFS ACLs are `setfacl` and `getfacl`) and to Microsoft Dfs. For example, the commands

```
$ cd //tqc.iu.edu/dfs/home
$ getfacl gustav
# file: gustav
# owner: gustav
# group: ovpit
user::rwx
group::r-x
other:r-x
```

```
mask:rwx
$ cd /cygdrive/c/winnt
```

put me in the Microsoft Dfs directory `\\tqc.iu.edu\dfs\home` first, then I list NTFS/Dfs ACLs on the sub-directory “gustav” and finally I go to `C:\WINNT`.

Cygwin comes with the secure shell and related utilities, X11 server and applications, GNU compilers, emacs, make, TeX, inetd utilities (you can run your own telnet and ftp servers), PostgreSQL data base, exim mail server and Apache WWW server. It’s all there and it’ll cost you nothing. If you wanted to *buy* all this functionality, you’d end up spending thousands of dollars on software.

How do you go about installing Cygwin? It’s easy. Go to

<http://www.redhat.com/download/cygwin.html>

and press “Download Now!”. This will download a small file called “setup.exe”. When you execute this file, you will be guided through various steps, including the download of binaries and documentation, and then unpacking and installation of the whole package on your PC. You will use the same binary to install Cygwin upgrades and patches down the road.

You don’t have to download and install the whole lot. It’s up to you how much of the package you really need and want to have. But I have installed all of Cygwin on my Windows 2000 PC and use pretty much most of it too. The whole Cygwin will take about 900MB of disk space on your C drive. If you intend to work with IU clusters a lot, if you plan on doing a lot of scientific work and if you want to do it all from your Windows box but without having to go through a double-boot, I recommend that you install Cygwin.

I am one of these weird and perverted individuals who are not very fanatical about UNIX, Linux, Windows, Macs and the rest. To me it’s all overpriced trash, unless it’s free (like Linux or Cygwin), in which case it is free trash. Quite ruthlessly I’ll reach for anything I can get my hands on, to get the job done as quickly and with as little fuss as possible. This usually implies a mixture of commercial software and freeware. Not all commercial software is worth spending money on, but then not all freeware is going to do the job either. You just have to mix and match, weighing your money, skills and requirements in the process.

So, let us assume that you already have Cygwin installed and there is going to be this little icon on your Windows desktop that looks like a black C with a green something inside it. Press on this icon and, assuming that you have configured everything correctly (and this is going to take some tinkering) you’ll get a window that looks rather slyly like a Linux window (even though it runs under Windows) with the `bash` prompt, e.g.,

```
gustav@WOODLANDS:../gustav 14:17:11 !516 $
```

My prompt tells me the name of the machine I’m on, the directory I’m in (but not the full path name, just the last segment in the path), the time of the day and the number of the command I am about to issue. But for the sake of brevity

I'm going to truncate it just to the name of the machine and the dollar in the examples that follow.

In this course we are going to use IUB and IUPUI clusters. To connect to one or the other, you have to login to either `avidd-b.iu.edu` (this is the Bloomington cluster) or to `avidd-i.iu.edu` (this is the IUPUI cluster). In both cases, you'll end up in the same home directory, which is going to be mounted on both clusters.

Let us begin with the Bloomington cluster. If this is your first connection, here is what it is going to look like:

```
WOODLANDS $ slogin avidd-b.iu.edu
gustav@avidd-b.iu.edu's password:
generating ssh file /N/B/gustav/.ssh/id_rsa ...
Generating public/private rsa key pair.
Created directory '/N/B/gustav/.ssh'.
Your identification has been saved in /N/B/gustav/.ssh/id_rsa.
Your public key has been saved in /N/B/gustav/.ssh/id_rsa.pub.
The key fingerprint is:
ed:84:29:8d:22:70:7d:5f:09:eb:c5:3b:ff:54:61:7b gustav@bh1
adding id to ssh file /N/B/gustav/.ssh/authorized_keys
[gustav@bh1 gustav]$
```

Observe that once you have made the connection you end up on the host called `bh1`. To be more precise, `avidd-b.iu.edu` evaluates to `bh1.uits.indiana.edu` and `avidd-i.iu.edu` evaluates to `ih1.uits.iupui.edu`. These two are called the *head* nodes of the two respective clusters. They are the nodes to which you connect from the *outside* in order to submit your AVIDD jobs. Such head nodes are also called *front-end* nodes.

The head nodes have more than just one network interface. The interfaces that correspond to `avidd-i.iu.edu` and `avidd-b.iu.edu` are on the public campus network. They are Gigi interfaces. These nodes can be seen on other networks too. There is a “cluster” network there, and a Myrinet network. The latter is used to run MPI jobs and to support GPFS, the General Parallel File System.

In order to make this first connection I had to type my AVIDD password explicitly. This is tedious, especially if you need to make new connections frequently and if you have a complicated password, and it is not very secure either, because the password travels over the network albeit in an encrypted form.

There is a simple way to change this by reconfiguring `ssh` on your PC and on the AVIDD cluster to work with DSA or RSA keys instead of passwords. This is more secure and more convenient too.

The procedure is as follows.

First you have to generate your own private/public key pair. You do this by calling a program `ssh-keygen`. This program will generate the keys and it will place them in the `.ssh` directory in your Cygwin home (which is like Linux home; on my PC I have simply linked it to the Windows’ “My Documents”).

You can generate RSA or DSA keys with `ssh-keygen`. DSA keys are more secure, so I recommend the latter. Issue the command:

```
WOODLANDS $ ssh-keygen -t dsa
```

The command will ask you for the DSA passphrase. The passphrase can be as long as you wish. Here is the first feature that makes the DSA system more secure than UNIX or Linux passwords, which are usually limited to just eight characters or so (if you have more than eight characters in your Linux password, the characters beyond eight may be ignored). Every character in your DSA passphrase matters, e.g., I have 43 characters in one of my favourite passphrases.

The private key will be stored on `~/.ssh/id_dsa` and the public key will be stored on `~/.ssh/id_dsa.pub`. You can show your public key to the world safely. Without its private partner it's useless. In particular you can transfer it to the AVIDD cluster and append it to the file `~/.ssh/authorized_keys`. The easiest way to do this is to copy it from your local laptop window and then paste it into the AVIDD window, in which you are editing `authorized_keys`. Make sure the pasted text is a single line. It may happen that the copy/paste process inserts newlines in the string. If it does, simply remove them. This is what my public DSA key looks like:

```
ssh-dss AAAAB3NzaC1kc3MAAACBAKykdA8AG7Vazhia9fI+uKgsyQzQSCK5LhaQy9XwmEk80hJ/Pg3T
4m+yZ1CS93GM2Z2HXEibCe39piNgg5d+0mhxaRHP48TUZhqX8pgU4vG89o/LqWmUSDAE1bnyjL7VHfI1
LCZ465dTJezZpAYLz1B+JU20CKjN4y46rzJsMMznAAAAFQDv1+pusBscm1hq0/Gxiz8E7o+eGQAAAIbc
6fDEDraImCtSty124Wi7rEamNDIabswc0bhMcm93Hr09V0No097A7c7shvs0bbdfwUDCMtYSwkaeHB4o
VRR/ULL9FWcxzbv3HFw81PTx1CFcyL2+u8e/1d2itpAruTzcs0QZNQ1dBRjpmPuz52TSD89WVOZE1Lox
58LKRy4ixAAAAIATq0aL2bbrXu2tK1QuMXYqFHSQUIXMWiQTW7ARJ8mu/EZ92MXvhRYBQYaSXkccq7HHq
qhpV3//sGhv28G5gxxFAIynD9xB7UxH44K0F8v1/KmF3H1dn74m3WWhn6+Xz6J0ttRRZa7ZGAzrNkwu4
TD3k3y4hyw4M7p/fhftJ8/oORA== gustav@WOODLANDS
```

Although the text printed above is spread over eight lines, it is, in fact, a single line.

Once you have done all this, you can carry out various transactions with the AVIDD cluster as follows.

```
WOODLANDS $ ssh-agent bash
WOODLANDS $ ssh-add
Enter passphrase for /home/gustav/.ssh/id_dsa:
Identity added: /home/gustav/.ssh/id_dsa (/home/gustav/.ssh/id_dsa)
WOODLANDS $ slogin avidd-b.iu.edu
[gustav@bh1 gustav]$ ^D
Connection to avidd-b.iu.edu closed.
WOODLANDS $ ssh avidd-b.iu.edu date
Thu Aug 7 15:10:46 EST 2003
WOODLANDS $ ssh avidd-b.iu.edu ls
src
WOODLANDS $ scp avidd-b.iu.edu:.bashrc bashrc-avidd
.bashrc                               100% 124      2.3KB/s   00:00
WOODLANDS $
```

Let me explain what happens here. First I have invoked the program `ssh-agent` and asked it to execute my login shell, `bash`. The agent is going to hoard my keys



and pass them on to any secure shell transactions *transparently*, i.e, without me having to type them in explicitly over and over. Once the `ssh-agent` has forked a new agent-supervised shell for me, I have invoked the command `ssh-add` in order to add my keys to the agent's cache. This is the only time I actually have to type the passphrase. Now I have issued the `slogin` command and got right to `avidd-b.iu.edu` without having to type my passphrase. I can also issue `ssh` commands without having to type the passphrase and check the date or the content of my home directory on the AVIDD cluster. The last command, `scp`, transfers the content of my `.bashrc` file on the AVIDD cluster to `bashrc-avidd` on my local machine. Again, I am not asked for the password or for the passphrase. The `ssh` agent takes care of this behind my back.

A useful command to execute in the `ssh` agent supervised shell is

```
WOODLANDS $ xterm -sb -sl 300 -n avidd -T avidd -e slogin avidd-b.iu.edu &
```

This command brings up an X11 window on your display with a shell running on the AVIDD cluster. Observe that the `xterm` program runs locally on your PC, not on the cluster. Exiting the AVIDD shell closes the window automatically.

For this to work, you must have X11 server running on your PC. You can use the one that comes with Cygwin. Its latest version is very good, almost as good as quite expensive commercial offerings, although it still has a couple of glitches here and there and may even hang on you after a day-full of activity.

Here is how I use it on my home computer. First I have copied a file `/usr/X11R6/bin/startxwin.sh` to my own private `~/bin` and modified it to look as follows:

```
#!/bin/sh
export DISPLAY=127.0.0.1:0.0
PATH=/usr/X11R6/bin:$PATH
rm -f /tmp/.X11-unix/X0
XWin -multiwindow -clipboard &
emacs &
xclock &
exit
```

I have stripped *numerous* comment lines off this script to make it shorter. The script cleans the X11 socket from the `/tmp/.X11-unix` directory (the original script tries to remove the whole directory, but this can fail sometimes, especially if other users run X11 and Cygwin programs on the same system too), then invokes the X11 Cygwin server, called `XWin`, with `-multiwindow` and `-clipboard` options. The first option combines X11 and native Windows displays into one, so that X11 applications can be managed the same way Windows applications are managed and so that you don't get a separate X11 root window in the background. The second option lets you copy and paste between Windows and X11 applications. Then I call `emacs` and `xclock` and the script exits. If everything works just fine, you should see the Emacs and the Clock windows pop up and you should also see a large X appear in the right corner of the task bar.

I usually run `startxwin.sh` under the `ssh-agent`. This way every X11 application carries my DSA keys with it. I can invoke these applications from my Emacs shell.

I connect to the AVIDD cluster using the following command:

```
$ xterm -sb -sl 300 -e slogin -X avidd-b.iu.edu &
```

The `-X` option will set up X11 environment for me automatically on the other side. The `DISPLAY` over there will be defined in terms of a socket local to the AVIDD head node. This file is readable to you only and it lives in the `/tmp` directory. Data sent to the socket will be encrypted and transmitted to your Cygwin X11 server, to be displayed on your screen. This is probably the safest way to use X11, because the data streams are going to be encrypted and because you don't enable access to your X11 display to all users on the AVIDD head node.

**Note** Secure shell installed on the AVIDD cluster is Open SSH version 3.6.1 patch 1. You get exactly the same secure shell with Cygwin, which should not be surprising, because Cygwin is Linux for Windows. But there are various other Secure Shells around, some free some commercial. I have Secure Shell 2 (`ssh2`) installed on one of my systems, and it has a different format for its authorization file, public keys, etc. `ssh2` format, copied directly from an `ssh2` public key file and pasted onto the Open SSH authorization file will not work. You may even end up locking yourself off the AVIDD head node altogether.

The `-X` option will work only between OpenSSH on your workstation and on the AVIDD cluster. It does not work on connecting to, e.g., SSH2 servers on other machines.

### 3.3 Finding Help and Documentation

Once you have connected to the AVIDD head node successfully, you may need to look up system documentation at times.

AVIDD is a very standard Linux and so documentation can be found in all the standard places. First there is a `man` directory in `/usr`. This is one of the traditional UNIX locations for this directory and it is frequently linked to `/usr/share/man` on other systems, but not on the AVIDD cluster. You will find manual entries in this directory that refer to GPFS the General Parallel File System. The most important ones talk about manipulating ACLs, Access Control Lists, on GPFS.

Most standard Linux commands, including GNU compilers, have their manual entries in `/usr/share/man`. However, X11 applications and libraries live in a separate directory tree of their own in `/usr/X11R6` and there they have their own manual directory in `/usr/X11R6/man`.

There is also a directory tree in `/usr` called `local` and some system applications are installed there together with their documentation. You will find their manual entries in `/usr/local/share/man`. There is also a directory `/usr/local/man`, which contains documentation for the PBS system, installed in `/usr/pbs`.

Program `man` reads file `/etc/man.config` (you can read it too) in order to find about locations of various manuals. If a directory that contains some

manuals, e.g., `/usr/local/share/man`, is not mentioned on that file, you will have to invoke the command with the `-M` switch, e.g.,

```
$ man -M /usr/local/share/man root
```

Apart from manual entries there are also additional documents and sometimes even tutorials in various `doc` directories. There is a `doc` directory in `/usr/share` and another one in `/usr/local/share`.

The GNU project distributes its documentation in the `info` format for reading with `emacs` and with a stand-alone program `info`. There may be more than one `info` directories on a system, in which case they should be declared on a file called `default.el` in the main Emacs Lisp directory. The main Emacs Lisp directory on the AVIDD head nodes is `/usr/share/emacs/21.2/lisp`, but there is no `default.el` there (at the time I'm writing this section). If you need to add more `info` directories, you can redefine the `emacs` `info` directory list in your `.emacs` file as follows:

```
(setq Info-default-directory-list
      '( "/usr/share/info/"
        "/usr/share/texmf/info/" ))
```

or you can append a directory to the existing path like this:

```
(setq Info-default-directory-list
      (nconc Info-default-directory-list
            (list "/usr/share/maxima/info"))) )
```

A source of great wisdom is always the `/etc/motd` file. This file will flash on every connection to the cluster unless you create a file `.hushlogin` in your AVIDD home directory.

There is also a WWW page that describes the AVIDD cluster and you will find it at <http://www.indiana.edu/~rats/research/avidd/index.shtml>. There is a nice photograph of one of the blade racks on this page too.

I asked the AVIDD administrators to mount this page at a more obvious WWW location, e.g., `www.avidd.iu.edu` or just `avidd.iu.edu`, but I was told by a gentleman named Craig Stewart that “doing something with web addresses as you suggest becomes entirely beyond our capabilities.” So there.

Other very important systems, like MPI, have their own directory trees. For example, the Argonne MPI system, MPICH, lives in `/usr/local/mpich` and extensive MPI documentation can be found in `/usr/local/mpich/gcc/doc` or in `/usr/local/mpich/intel70/doc` (depending on whether you want to use GNU or Intel compiler for your MPI applications). Corresponding manual pages live in `/usr/local/mpich/gcc/man` and `/usr/local/mpich/intel70/man` respectively.

LAM MPI, a system developed at the Ohio Supercomputer Center, which is currently maintained by Indiana University, lives in `/usr/local/lam-7.0` and you will find its `man` pages there too.

HDF-4.1 with its `man` pages can be found in `/usr/local/hdf`.

HDF5, ROMIO (a version of MPI IO that works with MPICH) and parallel DB2 are not installed on the AVIDD cluster yet. But I'll put a note here as soon we have the software.

## 3.4 Working with Data on AVIDD

When you login on a head node of the AVIDD cluster, you land in your HOME directory, which is most likely to live on the NFS, which is currently served from bf4 (in Bloomington) and if4 (in Indianapolis).

NFS is a convenience file system, but NFS IO is notoriously poor. You will not notice it on simple file editing operations, but you will notice it if you try to run any IO intensive jobs against it. Another thing you'll notice is that there isn't all that much space there: a mere 100 GBs on each server, which is peanuts by HPC standards.

It is not much even by PC standards. You can buy a desktop with a 160 GBs drive nowadays, and larger drives are in the works. They don't cost much and, incidentally, if you don't have enough space for your every-day data on your desktop or on your laboratory machine, simply go to Circuit City and buy as much as you need. We're talking about a few hundred dollars at most. A few years down the road, we're going to have TBs of disk space in our desktops and hundreds of GBs in our laptops.

OK, but let's get back to *today*. If you have a very large data set and want to analyze it on the AVIDD cluster, the place to put the files is on GPFS. GPFS is mounted on `/N/gpfsb` in Bloomington and on `/N/gpfsi` in Indianapolis and there is about 1.7 TBs on each.

If you have so much data that only a portion of it is going to fit on GPFS, you should keep the rest on HPSS, the High Performance Storage System. HPSS provides several peta-bytes (PBs) of space on tape cartridges, which live inside two robotic Storagetek silos: one in Indianapolis and another one in Bloomington.

In the next three sections we are going to discuss first the GPFS and then the HPSS and finally the way to move data between one and the other.

### 3.4.1 The AVIDD GPFS

GPFS is a *truly parallel* file system. In case of the AVIDD cluster the Bloomington component of GPFS is served by bf1, bf2, bf3, and bf4 and the Indianapolis component is served by if1, if2, if3, and if4.

What does *truly parallel* mean in this context? It means that every file that's written on GPFS ends up being *striped* over four different machines, in our case. The striping is much like striping of data on a disk array, but here the difference is that the GPFS "disk array" is assembled from disk arrays attached to several machines. This has the advantage of overcoming IO limitations of a single server.

Our experience with PC blades tells us that an IA32 server with a disk array attached to it can write at something about 15 MB/s on the array and it can read at up to 40 MB/s *from* the array. So if you have a very large file that is striped over four such servers, you should expect four times the above performance, i.e., 60 MB/s on the writes and 160 MB/s on the reads. And this is indeed, more or less, what we see on the AVIDD cluster (actually this is more rather than less, but we're within the right order of magnitude here).

60 MB/s on the writes does not amount to much. This is not high performance computing. For high performance computing we would need at least ten times more, which means 40 GPFS servers at IUB and another 40 at IUPUI. But for the time being we don't have projects in place that need this level of performance and so we have only four servers at each site. Still, this is enough for us to learn about this technology, and if you ever need much higher levels of performance, you can always get an account at the PSC, or NCSA, or SDSC.

SDSC is especially well equipped for data intensive work and they have demonstrated transfers from their tape silos at 828 MB/s. NCSA engineers tested a system with 40 GPFS servers on their IA32 cluster, getting expected performance, but they reduced the number of GPFS server nodes to four at present, and in future they're going to use a different solution, based on disks attached to a Storage Area Network (SAN) and PVFS, which is a freeware parallel file system from Clemson University in South Carolina.

Note that in order to have very high IO transfer rates to parallel processes running on clusters you *must* have a parallel file system. Whether this file system is served from disk arrays on SAN or from disk arrays directly attached to some server nodes is another issue.

But let's get back to AVIDD. The directory /N/gpfsb is writable to all. So the first thing you need to do is to create your own subdirectory there:

```
[gustav@bh1 gustav]$ cd /N/gpfsb
[gustav@bh1 gpfsb]$ mkdir gustav
[gustav@bh1 gpfsb]$
```

From this point on you can use this directory like a normal UNIX file system. This, in particular, is where you should run your parallel jobs from.

GPFS files are protected by the usual UNIX permissions, i.e., read (r), write (w) and execute (x) for the user, group and others. GPFS on the SP also supports additional controls in the form of Access Control Lists, ACLs. These can be manipulated with the "mm" commands, like mmeditacl, mmgetacl and mmputacl, which are described in /usr/man/man1, but these commands don't seem to work on the AVIDD GPFS. The IA32 version of GPFS is quite restricted compared to the fully functional GPFS you get on the SP.

Although AVIDD GPFS delivers only 60/160 MB/s on writes/reads, there are situations when you may notice much higher transfer rates, especially on writes. How can this be? The answer is memory caching. UNIX never writes data on the media directly. When you open a file, write on it, close it, even when you flush (UNIX programmers should know what this means), the data does not go the disks. Even when UNIX *thinks* that it has pushed the data to

the disk, the data may be still stuck in the disk's own memory cache. These multiple levels of memory caches, both on UNIX, on GPFS, and on the physical disk arrays themselves serve to mask the slowness with which data is written on the actual physical media compared to the speed with which it can be handled internally within the computer. Because GPFS server nodes and computational nodes all have a lot of memory (I think they have at least 2 GB each), they can cache a lot of IO, perhaps even hundreds of MBs. So if you write just a 100 MBs of data to GPFS, you may discover that the write has occurred at memory speeds, not disk speeds. The only time you actually get to see the real IO transfer rate is when you write a very, very large file. Then all the memory caches overflow and data eventually has to be written on the physical media. As you keep pushing the data from your program you will see the transfer rate drop and drop until it reaches the real disk write speed eventually.

Another trick that hardware vendors employ is automatic data compression on disk arrays. This is often done by a special chip that is embedded in the array controller. The effect of this is that if you write a file of, say, zeros on the drive, the transfer rate is going to be phenomenal, even with all the memory caching out of the way. So, if you want to test real IO in this context, you need to write strings of random numbers or characters on the drive. Such strings cannot be compressed.

All this memory caching and automatic data compression are good things and there is a way to make use of them even when you work with very large files. The way is to process a file on a very large number of GPFS clients. If you write a file in parallel from, say, 40 GPFS clients, even if the file is 10 GBs long, you'll end up writing only 250 MBs per client. This amount of data is probably going to be cached in the client's memory while your program executes and closes the file. The data will take much longer to flow to the media, of course, but all this is going to take place in the background and you will not have to wait for it. As far as you are concerned, you will probably have written this file at several hundreds of MBs per second.

Physical writes are always slower than physical reads. There are physical reasons for this. When you read data from a disk, you need to find where the data is. This is usually a pretty fast process, based on hashed tables or something similar. Eventually you get to the data, the head lowers and reads the data from the disk surface. The data then flows to the user. When you write data on the disk, the process is slower because, first, you have to locate where the best free space is, then you have to add new records to the hashed table and this is a slower process than just locating existing data, and finally you have to write data on the media. The writing process is more involved: stronger magnetic fields have to be applied, there is a lot of checking, if there are some errors, then the writes are repeated and portions of the disk may have to be marked as bad, and so on.

Virtual writes are always faster than virtual reads. The reason for this is that virtual writes only write data on memory and so they can be very fast. But virtual reads almost always have to read data from the disk physically, *unless* the data has been read very recently, and is still cached in the disk's controller.

### 3.4.2 IU HPSS

The High Performance Storage System, HPSS, is a hierarchical massive data storage system, which can store many PBs of data on tape cartridges mounted inside automated silos, and which can transfer the data at more than one GB/s, if appropriately configured. HPSS is designed *specifically* to work with very large data objects and to serve clusters with parallel file systems such as GPFS (see section 3.4.1).

HPSS is used by Trilab, i.e., LANL, LLNL and Sandia, BAE Systems (they have more than twenty HPSS installations), SDSC NASA, Oak Ridge, Argonne (ANL), National Climatic Data Center (NCDC), National Centers for Environmental Prediction (NCEP), Brookhaven, JPL, SLAC, three research institutes in Japan, KEK, RIKEN, and ICRR), one research institute in Korea, KISTI, European Centre for Medium Range Weather Forecast (ECMWF), French Atomic Energy Commission (CEA) and Institut National De Physique Nucleaire Et De Physique Des Particules (IN2P3), The University of Stuttgart in Germany, Indiana University, of course, and some other large customers. Although the number of HPSS users is not very large, the amount of data these users keep on HPSS is more than 50% of all world's data, sic! HPSS is a very serious system for very serious Men in Black. It is not your average off-the-shelf Legato.

HPSS has been a remarkable success at Indiana University, even though we have not made much use of it in the high performance computing context yet. But HPSS is very flexible and it can be used for a lot of things.

Yet, as with any other system of this type, and there aren't that many, you must always remember that HPSS is a tape storage system when you work with it, even though it presents you with a file system interface when you make a connection to it with ftp, or hsi or pftp. This has some important ramifications.

As GPFS is a *truly parallel* file system, HPSS is a *truly parallel* massive data storage system. This is why the two couple so well.

HPSS files can be striped over devices connected to multiple HPSS servers. It is possible to establish data transfer configuration between HPSS and GPFS in such a way that the file is moved in parallel between HPSS servers and GPFS servers. This operation is highly scalable, i.e., you can stripe an HPSS file and its GPFS image over more and more servers and the data transfer rate will scale linearly with the number of servers added. But such scalability is costly, since every new server and disk array you add costs at least a few thousand dollars. Still, a few thousand dollars for a GPFS or an HPSS server is very little compared to what such systems used to cost in the past. For example a Convex machine that used to be amongst the best servers for the UniTree massive data storage system used to cost several hundred thousand dollars, it had to be connected to other supercomputers by a HIPPI bus, and data transfer rates would peak at about 40 MB/s. With 16 well tuned and well configured HPSS PC servers and 16 equally well tuned and well configured GPFS servers you should be able to move data at 360 MB/s in each direction. Note that whichever direction you move the data in, you're always slowed down to the write speed on *the other side*. I have seen inexpensive PC attached IDA disk

arrays that supported writes at 20 MB/s. So,  $20 \times 16 = 360$ .

In our case the situation is somewhat unbalanced. We have a somewhat better IO at the HPSS side and a somewhat worse IO at the AVIDD side, and only 4 servers at each side and so we end up with about 40 MB/s on writes to GPFS and 80 MB/s on writes to HPSS.

But first things first.

### Your HPSS Account

To get an HPSS account proceed the same way you did with the AVIDD account, i.e., go to <http://itaccounts.iu.edu/>, click on who you are (e.g., Student or Faculty), click on “Create more IU computing accounts”, and press “Continue”.

A new page will appear, where you will have to log in. Type your IU network ID, your IU network password and press “Continue” again.

You get to another page. Press on “create more accounts”, and now various options appear, amongst them should be “MDSS”, which stands for “Massive Data Storage System”, which is how we refer to our IU HPSS (because this is what it is, whereas HPSS is a particular system that is used to provide this service).

This will work only if you are a staff, a faculty or a postgraduate student.

If you are an undergraduate student, you will have to ask your academic supervisor to request the MDSS account for you for the duration of this course.

In case of any problems with HPSS, send e-mail to Storage Administrators.

### Finding Information about HPSS

Information about IU HPSS and other storage services at Indiana University is provided on <http://storage.iu.edu>.

Connect to this WWW site to read the latest announcements, to learn about services available, to get access to available documentation, tutorials and workshops. In particular have a look at the IU MDSS and CFS Tutorial penned by Anurag Shankar and myself. The tutorial covers quite a lot of stuff and provides some useful hints and advice about using the system, all on just a few easy-to-read pages. You can download the PDF version of the tutorial for printing with Adobe Acrobat too.

The stuff that is going to be of particular importance to us is Parallel FTP and Multinoded Transfers. You should study it before moving on to the next section.

The tutorial has been written for the IU SP and things may work a little differently between HPSS and AVIDD. We are going to discuss this and illustrate with hands-on examples next.

### 3.4.3 Moving Data Between HPSS and GPFS

I am now going to explain to you how to move data between AVIDD GPFS and HPSS.



But before we get to this topic we are going to have a simple IO programming exercise first. We are going to concoct a program that writes a file full of random numbers. The program works similarly to a popular UNIX tool called `mkfile`, but whereas `mkfile` writes a file full of nulls, which can be misleading if you want to test IO, because of hardware implemented data compression on disk arrays, our program randomizes the output.

### Program `mkrandfile`

Here is what the program looks like

```

/*
 * Create a file containing random pattern of specified length.
 *
 * %Id: mkrandfile.c,v 1.1 2003/09/11 19:32:16 gustav Exp %
 *
 * %Log: mkrandfile.c,v %
 * Revision 1.1 2003/09/11 19:32:16 gustav
 * Initial revision
 *
 */

#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<errno.h>
#define ARRAY_LENGTH 1048576

main(argc, argv)
    int argc;
    char *argv[];
{
    FILE *fp;
    int junk[ARRAY_LENGTH], block, i, number_of_blocks = 0;

    /* variables for reading the command line */

    extern char *optarg;
    char *name = NULL;
    int c;

    /* error handling */

    extern int errno;

    while ((c = getopt(argc, argv, "f:l:h")) != EOF)
        switch(c) {
        case 'f':
            name = optarg;
            (void)printf("writing on %s\n", name);

```

```

        break;
    case 'l':
        sscanf (optarg, "%d", &number_of_blocks);
        (void)printf("writing %d blocks of %d random integers\n",
number_of_blocks, ARRAY_LENGTH);
        break;
    case 'h':
        printf ("synopsis: %s -f <file> -l <length>\n", argv[0]);
        exit(0);
    case '?':
        printf ("synopsis: %s -f <file> -l <length>\n", argv[0]);
        exit(1);
    }

    if (number_of_blocks < 1) {
        printf ("initialize number of blocks with -l option\n");
        printf ("use -h for help\n");
        exit(2);
    }

    if (name == NULL) {
        printf ("initialize file name with -f option\n");
        printf ("use -h for help\n");
        exit(2);
    }

    if (! (fp = fopen(name, "w"))) {
        perror (name);
        exit(3);
    }

    srand(28);
    for (block = 0; block < number_of_blocks; block++){
        for (i = 0; i < ARRAY_LENGTH; junk[i++] = rand());
        if (fwrite(junk, sizeof(int), ARRAY_LENGTH, fp) != ARRAY_LENGTH) {
            perror (name);
            exit (4);
        }
    }
    if (fclose(fp) != 0) {
        perror (name);
        exit (5);
    }

    exit (0);
}

```

Let me explain what the program does and how. This is also going to brush up on your C and UNIX programming skills.

The program begins by calling a standard UNIX function `getopt`, which reads the command line. The string `"f:l:h"` tells `getopt` what option switches we expect to see on the command line. And so, we expect to see `-f` followed by an argument, then `-l` followed by an argument (the presence of the argument is indicated by `:"`), and `-h`, which doesn't have a following argument.

The options can appear on the command line in any order. This is a nice thing about `getopt`. And it is not necessary to use all the options either.

The `switch` statement that is inside the `while` loop processes various cases. And so, if the option character is `"f"` (the minus sign in front of the option character is always implied by `getopt`), then the argument that follows `-f`, and this argument is *always* going to be a string, is going to be the name of the file on which we are going to write our randomized integers. Observe that `name` is just a pointer to a character, and it is initialized to `NULL`. We don't have to copy the content of the string pointed to by `optarg`, the latter being a pointer to a character too. Instead we simply request that `name` should point at the same memory location as `optarg`. This way we can use the name of the file passed to the program on the command line, by referring to it as `name`.

If the option character is `"l"`, the option (`-l`) should be followed by an integer. This integer is going to be interpreted as the number of blocks of one million integers each to be written on the file. But recall that `optarg` is always a string. So here we have to convert this string to an integer. This can be done in various ways, most of them quite cumbersome, but there is one easy and relatively safe way to do this. I use function `sscanf`, which scans a string the same way that `scanf` scans an input line. The string in question is pointed to by `optarg`, and we expect the format to be `"%d"`, which means "an integer". The last parameter in the call to `sscanf` is *the address* of `number_of_blocks`. Function `sscanf` will go there and will write the value of the integer in that location. If `sscanf` does not find an integer in `optarg` it won't write anything there, in which case `number_of_blocks` will retain its original value of zero.

The last option defined by `"f:l:h"` is `-h`. This option does not take an argument. If it is encountered on the command line, the program will print the brief synopsis on standard output and it will exit the program cleanly, i.e., with the exit status zero, right there and then.

If function `getopt` encounters any other option on the command line, it is going to return `"?"` and the last `case` of the `switch` statement, will print the synopsis of the program on standard output and will exit the program raising an error flag, i.e., the exit status will be set to one.

Having collected the information from the command line, we are now going to check two things, just in case. The first one is the value of `number_of_blocks`. This value may be incorrect if, e.g., it has not been specified on the command line, then it would be zero, or if it has been specified incorrectly, e.g., made negative. If any of the two conditions holds, we're going to exit with the exit status set to 2. The second thing we're going to check is if we have the name of the file. If the user has not specified the name of the file on the command line, `name` will stay set to `NULL`. If we detect this condition, we're going to exit the program with the exit status set to 2 too.

These two simple checks do not exhaust countless possibilities that the user may get something wrong. For example the user may type a weird file name, which may be either too long or it may contain some control codes, or the number of blocks requested may be too large. Some of the problems may be captured by checks in the remainder of the program, but some may remain undetected until it is too late and something horrible happens. This is how security bugs are born. The command line reading procedure in our program is somewhat fragile, but it will do for now.

Having obtained all the information from the command line and having checked that it is sensible, we are now going to open the file for writing. The statement that does it is

```
if (! (fp = fopen(name, "w"))) {
    perror (name);
    exit(3);
}
```

Observe that we don't just open the file. We *attempt* to open it and then immediately check if we've been successful or not. If the attempt is unsuccessful, function `fopen` is going to return `NIL`, which becomes *false* when cast on `BOOLEAN`. The negation operator, `!`, will make it into *true* and the clause following the `if` statement will be executed. Inside this clause function `perror` is going to print the name of the file, pointed to by `name`, followed by a colon and by a brief explanation of the error condition encountered. Then we abort having set the exit status to 3. It is here that we can trap an incorrectly entered file name. The failure to open the file may be due to many reasons though. Amongst them may be, e.g., that there is already a write protected file of this name, or that the user has no write permission on the directory in which the program executes.

If `fopen` returns anything but `NIL`, this is interpreted as *true*, the negation operator `!` makes it into false, and then we go the remainder of the program.

The remainder is short and sweet. We seed the random number generator with 28 (nothing special about this number) and then write the random numbers generated by function `rand` on the file. Observe that we don't write a character at the time. Instead we collect about a million of random integers on an array called `junk`, and only after the array is full, we push its content onto the file in a single long write. This is the most efficient way of writing data: large blocks, not little drops. But then recall that the blocks in our program are not so large as to overwrite output buffers. A million of integers fill 4 MBs only, which is nothing.

Function `fwrite` is called in a way reminiscent of `fopen`. This function returns the number of *items* of size `sizeof(int)` it has managed to write. This number should be equal to `ARRAY_LENGTH` if the write has been successful. Otherwise, we have a write error. For example, we may have run out of disk space, or out of quota, and then the number is going to be less than `ARRAY_LENGTH`. If we detect that `fwrite` has *not* written as many integers as it should have,

we call function `perror`, which should tell us what happened, and then abort having set the exit status to 4.

The last operation the program does is closing the file. Function `fclose` should return zero if there are no problems, otherwise it is going to return some error code. We don't analyze the error code, relying on function `perror` instead. If we have encountered an error condition at this stage, we abort the program having set the exit status to 5.

Otherwise, i.e., if everything has gone just fine, we exit the program cleanly, and set the exit status to zero.

We are going to make and install this program by calling UNIX `make`. But before we can do this we have to edit `Makefile` first. And here it is:

```
#
# %Id: Makefile,v 1.1 2003/09/11 19:31:57 gustav Exp %
#
# %Log: Makefile,v %
# Revision 1.1 2003/09/11 19:31:57 gustav
# Initial revision
#
DESTDIR = /N/B/gustav/bin
MANDIR = /N/B/gustav/man/man1
CC = cc
TARGET = mkrandfile

all: $(TARGET)

$(TARGET): $(TARGET).o
$(CC) -o $@ $(TARGET).o

$(TARGET).o: $(TARGET).c
$(CC) -c $(TARGET).c

install: all $(TARGET).1
[ -d $(DESTDIR) ] || mkdirhier $(DESTDIR)
install $(TARGET) $(DESTDIR)
[ -d $(MANDIR) ] || mkdirhier $(MANDIR)
install $(TARGET).1 $(MANDIR)

clean:
rm -f *.o $(TARGET)

clobber: clean
rcsclean
```

Our target `all` is defined as being `$(TARGET)`, which here evaluates to `mkrandfile`. Observe that this `Makefile` is quite general and you can reuse it for simple programs with other names too.

To make it we have to have `$(TARGET).o`, i.e., `mkrandfile.o` first. Once we have it, we link it by calling

```
$(CC) -o $@ $(TARGET).o
```

which evaluates to

```
cc -o mkrandfile mkrandfile.o
```

because `$@` always evaluates to the target itself.

To make `mkrandfile.o`, we must have the source file, `mkrandfile.c`. Once we have made any changes to it, we compile it with the command

```
$(CC) -c $(TARGET).c
```

which evaluates to

```
cc -c mkrandfile.c
```

This command creates `mkrandfile.o`, but it doesn't link it.

The installation is done by calling UNIX program `install`, which works a little like `copy`, but it can do some checking, set permissions, ownership, etc. Here we install our application in the directory pointed to by `$(DESTDIR)`, which is simply a `bin` subdirectory in my home directory on the AVIDD cluster. We also install the manual page that describes the program in the `man/man1` subdirectory of my home directory.

Observe that we check if the directories exist in the first place, and we make them if they don't. The program `mkdirhier`, which belongs in the X11 (X11R6 in case of AVIDD) toolkit, will make the whole directory hierarchy if needed.

All three files, `mkrandfile.c`, `mkrandfile.1` and `Makefile` are under the RCS control. RCS stands for Revision Control System and it helps maintain the codes in an orderly fashion. All changes made to the code are remembered and can be reversed. RCS maintains the log and version number for every file under its management too.

Here's how it works:

```
[gustav@bh1 mkrandfile]$ pwd
/N/B/gustav/src/mkrandfile
[gustav@bh1 mkrandfile]$ ls
RCS
[gustav@bh1 mkrandfile]$ make
co RCS/Makefile,v Makefile
RCS/Makefile,v --> Makefile
revision 1.1
done
co RCS/mkrandfile.c,v mkrandfile.c
RCS/mkrandfile.c,v --> mkrandfile.c
revision 1.1
done
cc -c mkrandfile.c
cc -o mkrandfile mkrandfile.o
[gustav@bh1 mkrandfile]$ make install
```

```

co RCS/mkrandfile.1,v mkrandfile.1
RCS/mkrandfile.1,v --> mkrandfile.1
revision 1.1
done
[ -d /N/B/gustav/bin ] || mkdirhier /N/B/gustav/bin
install mkrandfile /N/B/gustav/bin
[ -d /N/B/gustav/man/man1 ] || mkdirhier /N/B/gustav/man/man1
install mkrandfile.1 /N/B/gustav/man/man1
[gustav@bh1 mkrandfile]$ ls
Makefile RCS mkrandfile mkrandfile.1 mkrandfile.c mkrandfile.o
[gustav@bh1 mkrandfile]$ make clobber
rm -f *.o mkrandfile
rcsclean
rm -f Makefile
rm -f mkrandfile.c
rm -f mkrandfile.1
[gustav@bh1 mkrandfile]$

```

The source lives in the `/N/B/gustav/src/mkrandfile/RCS` directory. The directory right above it is empty with the exception of the `RCS` subdirectory. The command `make` recognizes the presence of `RCS` and checks out the required files with the `RCS` command `co`. The first file to be checked out is, of course, `Makefile`. Then the file is analyzed and `make` commences to execute instructions in it. The source is compiled then linked. `make install` installs the binary and the manual entry in the appropriate directories, after it has checked that they exist in the first place.

The `make` process leaves various files in the source directory, i.e., the directory that is above the `RCS` subdirectory. The command `make clobber` calls `make clean` first. This deletes the object file and the executable. Then the `RCS` command `rcsclean` deletes images of files that are already archived in the `RCS` subdirectory.

The manual file is written in UNIX `troff`. `troff` is a rather ancient text processor (more than 30 years old). But it is so firmly embedded in UNIX and UNIX tradition (UNIX was developed specially for `troff`) that it is impossible, or at least unwise, to untangle the two.

Here is what the manual page source looks like:

```

.\" Process this file with
.\" groff -man -Tascii mkrandfile.1
.\"
.\" %Id: mkrandfile.1,v 1.2 2003/09/11 21:05:30 gustav Exp %
.\"
.\" %Log: mkrandfile.1,v %
.\" Revision 1.2 2003/09/11 21:05:30 gustav
.\" Filled the whole of TH.
.\"
.\" Revision 1.1 2003/09/11 19:32:07 gustav
.\" Initial revision

```

```

.\
.TH MKRANDFILE 1 "SEPTEMBER 2003" I590/7462 "I590 Programmer's Manual"
.SH NAME
mkrandfile \- create a file full of randomized integers
.SH SYNOPSIS
.B mkrandfile
-f \fIfilename\fR
-l \fIsize\fR
[-h]

.SH DESCRIPTION
.B mkrandfile
creates a file full of randomized integers. Can be used to create
files for disk-based benchmarks.

.SH OPTIONS
.IP \fB-f\fR
must be followed by output filename. e.g. foo

.IP \fB-l\fR
must be followed by number of 4MB blocks. e.g. 100

.IP \fB-h\fR
Print a brief help message

.SH DIAGNOSTICS
Self explanatory and numerous.

.SH EXAMPLES

$ mkrandfile -f /N/gpfs/gustav/tryme -l 10
.br
writing on /N/gpfs/gustav/tryme
.br
writing 10 blocks of 1048576 random integers
.br
$ ls -l /N/gpfs/gustav/tryme
.br
-rw-r--r-- 1 gustav ucs 41943040 Aug 29 14:22 /N/gpfs/gustav/tryme
.br

.SH BUGS
Command line processing may be brittle.

.SH AUTHOR
Zdzislaw Meglicki <gustav@indiana.edu>

Read man 7 man on AVIDD to learn about the meaning of troff and man
directives used in the above listing.

```



Having made and installed the program, you can bring up its man page by typing:

```
[gustav@bh1 gustav]$ man mkrandfile

MKRANDFILE(1)          I590 Programmer's Manual          MKRANDFILE(1)

NAME
    mkrandfile - create a file full of randomized integers

SYNOPSIS
    mkrandfile -f filename -l size [-h]

DESCRIPTION
    mkrandfile creates a file full of randomized integers.
    Can be used to create files for disk-based benchmarks.

OPTIONS
    -f      must be followed by output filename.  e.g. foo

    -l      must be followed by number of 4MB blocks.  e.g.
            100

    -h      Print a brief help message

DIAGNOSTICS
    Self explanatory and numerous.

EXAMPLES
    $ mkrandfile -f /N/gpfs/gustav/tryme -l 10
    writing on /N/gpfs/gustav/tryme
    writing 10 blocks of 1048576 random integers
    $ ls -l /N/gpfs/gustav/tryme
    -rw-r--r--  1 gustav   ucs      41943040 Aug 29 14:22
    /N/gpfs/gustav/tryme

BUGS
    Command line processing may be brittle.

AUTHOR
    Zdzislaw Meglicki <gustav@indiana.edu>

I590/7462          SEPTEMBER 2003          MKRANDFILE(1)

[gustav@bh1 gustav]$
```

Let me summarize what we have done in this section. First we have written and discussed a very, very, very simple UNIX C program, which we are going to use to generate a very large file of random integers in the following sections. We have also written a Makefile and a man page for this program. Finally, we

have run `make` to compile, link and install the program and its man page.

The three files: the source code, the Makefile and the manual page, together constitute the application. If any one of the three is missing, the job is botched. The job can be botched, of course, in many other ways too, e.g., the program may have bugs, the manual may not format correctly, and the Makefile may have incorrect instructions in it. But you must always remember that it is not enough to just write the program. You must provide the means of making it into an application and you must provide documentation for it too.

### Exercises

- 1 Function `sscanf` returns the number of input items assigned. These may be *fewer than* what the code provides for, or even zero. Use the return value of function `sscanf` to capture a possible input error in the '1' case of the `getopt` while loop and exit the program, raising an error flag, if such has been detected.
- 2 Implement a command line option that lets the user change the value of `ARRAY_LENGTH`.
  - Hint 1** `ARRAY_LENGTH` must be replaced with a variable.
  - Hint 2** Because `ARRAY_LENGTH` is now a variable, `junk` can no longer be a statically sized array. Instead you will have to allocate it dynamically with, e.g., `malloc`.
  - Hint 3** You must check if `malloc` has been successful, before you can do anything else.
- 3 The listings in the previous section present RCS identification strings bordered by percent signs. They should actually be bordered by dollar signs, but this results in unfortunate interference with the RCS system used to maintain this document. Read `man 7 co` on the AVIDD cluster to learn about RCS keywords. Add option `-v` that prints the version of the program (its revision number) as generated automatically by RCS.
- 4 Amend the manual page to reflect the enhancements made to the program.
- 5 Use the new program to check if GPFS I/O depends on the size of the `junk` array. Is there an optimum value?

### Using PFTP on AVIDD

PFTP is a program very similar to FTP, but it can move data in parallel between parallel file systems such as GPFS and parallel massive data storage systems such as HPSS. PFTP clients are installed in `/usr/local/hpss` both on the IUB and on the IUPUI head nodes.

There are two PFTP clients in that directory, a kerberized version, which is called

```
krb5_gss_pftp_client
```

and a non-kerberized one, which is called

```
pftp_client
```

The kerberized version is secure and this is what you should use normally, but it doesn't work correctly at present due to some configuration glitches that still have to be resolved. But I will show you briefly how it should work.

In order to use the Kerberized version of PFTP you have to acquire your Kerberos credentials in the HPSS cell first. The name of the cell is dce1.indiana.edu. To acquire the credentials use the program kinit:

```
[gustav@bh1 gustav]$ kinit gustav@dce1.indiana.edu
Password for gustav@dce1.indiana.edu:
[gustav@bh1 gustav]$ klist
Ticket cache: FILE:/tmp/krb5cc_43098
Default principal: gustav@dce1.indiana.edu
```

```
Valid starting Expires Service principal
09/04/03 15:18:52 09/05/03 01:18:52 krbtgt/dce1.indiana.edu@dce1.indiana.edu
```

```
Kerberos 4 ticket cache: /tmp/tkt43098
klist: You have no tickets cached
[gustav@bh1 gustav]$
```

You can check the status of your Kerberos credentials with the command klist, as I have done in the example above.

Once you have acquired the credentials, you can connect to HPSS as follows:

```
[gustav@bh1 gustav]$ krb5_gss_pftp_client hpss.iu.edu 4021
Parallel block size set to 4194304.
Connected to hpss.iu.edu.
```

```
[ Message of the day comes here. It may be out of date. ]
```

```
220 hpss01.ucs.indiana.edu FTP server
(HPSS 4.3 PFTPD V1.1.1 Fri Jul 19 13:59:25 EST 2002) ready.
334 Using authentication type GSSAPI; ADAT must follow
GSSAPI accepted as authentication type
GSSAPI authentication succeeded
Preauthenticated FTP to hpss.iu.edu as gustav:
232 GSSAPI user ../../dce1.indiana.edu/gustav is authorized as
../../dce1.indiana.edu/gustav
230 User ../../dce1.indiana.edu/gustav logged in.
Remote system type is UNIX.
Using binary mode to transfer files.
```

```
[ Other messages come here. ]
```

```
Multinode is Disabled.
ftp> pwd
257 "../../dce1.indiana.edu/fs/mirror/g/u/gustav" is current directory.
ftp> quit
221 Goodbye.
[gustav@bh1 gustav]$
```

Program `krb5_gss_pftp_client` passes your Kerberos credentials, in a very secure fashion – your Kerberos password never travels over the network and you don’t have to type it in explicitly anyway – to the PFTP authentication helper that runs, in this case, on `hpss01.ucs.indiana.edu`, and the latter completes your authentication and authorization.

You can keep your Kerberos credentials as long as you need, but eventually they will expire after some 10 hours or so anyway. It is a good practice to destroy the credentials before you quit the system. To do so issue the command `kdestroy`:

```
[gustav@bh1 gustav]$ kdestroy
[gustav@bh1 gustav]$ klist
klist: No credentials cache found (ticket cache FILE:/tmp/krb5cc_43098)
```

```
Kerberos 4 ticket cache: /tmp/tkt43098
klist: You have no tickets cached
[gustav@bh1 gustav]$
```

Kerberos credentials acquired with `kinit` survive the logout process. The reason for this is to let you leave a background process that may depend on the credentials. Of course, if you have such process running, you should not destroy the credentials.

Connecting with `pftp_client` instead of `krb5_gss_pftp_client` is much the same, with the single exception of having to type in your HPSS password manually. The password is transmitted over the network without encryption. This is the principal reason for using the Kerberized version.

The IUPUI cluster and the IUPUI HPSS component are better configured for moving data in between at present, and so we’ll switch to the IUPUI cluster in the following examples.

But before we go any further, let us create a nice, large file containing randomized integers on GPFS:

```
[gustav@ih1 gustav]$ cd /N/gpfs/gustav
[gustav@ih1 gustav]$ mkrandfile -h
synopsis: mkrandfile -f <file> -l <length>
[gustav@ih1 gustav]$ time mkrandfile -f test -l 1000
writing on test
writing 1000 blocks of 1048576 random integers

real    5m19.838s
user    0m39.450s
sys     0m24.090s
[gustav@ih1 gustav]$ ls -l
total 4096000
-rw-r--r--  1 gustav  ucs      4194304000 Sep  4 15:46 test
[gustav@ih1 gustav]$
```

It took 5 minutes and 20 seconds to write this 4 GB file on GPFS, which yields about 13 MB/s. Some of this time was spent calling the random number generation function, but that should be much faster than IO. What transfer rate are we going to get on reading it?

```
[gustav@ih1 gustav]$ time cat test > /dev/null
```

```
real    0m29.001s
user    0m0.130s
sys     0m15.730s
[gustav@ih1 gustav]$
```

This time it only took 29 seconds, which yields transfer rate of about 144 MB/s. This is very fast and we might suspect that the file is either buffered or that UNIX cheats on the `> /dev/null` redirection and simply drops the file without reading.

It is easy to check on the latter as follows:

```
[gustav@ih1 gustav]$ time cat test > test1

real    6m20.750s
user    0m0.040s
sys     0m21.780s
[gustav@ih1 gustav]$
```

Here we read 4 GBs and as we do so, we write 4 GBs at the same time. If the 144 MB/s transfer rate was real, we should be able to do this in less than 5 minutes and 20 seconds (for writing) plus 30 seconds (for reading), which is 5 minutes and 50 seconds. We do it actually in 6 minutes and 20 seconds, i.e, 30 seconds longer. This is close enough, the more so as data flows through various memory buffers in this process and there is some time-consuming data copying involved. We can therefore guess that UNIX does not cheat on `> /dev/null` and the reading indeed proceeds at about 144 MB/s.

Because there are four GPFS servers on the IUPUI AVIDD component, this yields about 36 MB/s per server, which is quite normal for reading from disk arrays.

Why then is writing to GPFS so slow compared to reading?

The reason for this is that writing is by necessity sequential. The system does not know a priori how long the file is going to be. The system receives data sequentially from the generating process and it writes it sequentially, as files are usually written, sending portions of data first to the first GPFS server, then to the second one, then to the third, fourth and then back to the first one, as the file gets striped over GPFS. In order to speed up this process, we would have to write the file in parallel – not sequentially – and we are going to learn how to do this down the road.

But let us get back to HPSS and let us now transfer the file from GPFS to HPSS. Here is how I go about it:

```
[gustav@ih1 gustav]$ cd /N/gpfs/gustav
[gustav@ih1 gustav]$ ls -l
total 8192000
-rw-r--r--  1 gustav  ucs      4194304000 Sep  4 15:46 test
-rw-r--r--  1 gustav  ucs      4194304000 Sep  4 15:59 test1
```

So, here I went to GPFS on AVIDD-I (the IUPUI cluster) and checked that my 4GB files are there. Now I am going to connect to HPSS using `pftp_client`:

```
[gustav@ih1 gustav]$ pftp_client hpss.iu.edu
Parallel block size set to 4194304.
Connected to hpss.iu.edu.
```

[ message of the day ]

```
220 hpss-s12.uits.iupui.edu FTP server
    (HPSS 4.3 PFTPD V1.1.1 Wed Sep 11 15:18:05 EST 2002) ready.
Name (hpss.iu.edu:gustav): gustav
331 Password required for ../../dce1.indiana.edu/gustav.
Password:
230 User ../../dce1.indiana.edu/gustav logged in.
Remote system type is UNIX.
Using binary mode to transfer files.
```

[ other messages ]

Multinode is Disabled.

OK, so now I'm in. I land in my HPSS home directory, which is linked to our institutional DFS. This directory is not suitable for very large files – you must never put anything as large as 4GB there. Instead I switch to the so called “hpssonly” directory, where I can place files of arbitrary size.

```
ftp> pwd
257 "../../dce1.indiana.edu/fs/mirror/g/u/gustav" is current directory.
ftp> cd /:/hpssonly/g/u/gustav
250 CWD command successful.
```

Now I am going to activate the multinode transfer mode. This means that the data will flow in parallel. There is a file in /usr/local/etc, called HPSS.conf, which specifies the configuration and the nodes through which data should flow directly.

```
ftp> multinode
    Processing the multinode list, please wait.....
Multinode is on.
```

It is not enough to request multinode transfer mode. You have to specify how many data streams you want to have in the transfer. In our case we have four HPSS servers and four GPFS servers, so we need four parallel data streams:

```
ftp> setpwidth 4
Parallel stripe width set to (4).
    Processing the multinode list, please wait.....
```

Now we need to request the HPSS Class of Service. This class is actually a default when you make a connection to HPSS from AVIDD-I. It is a striped class of service, over four HPSS servers, and it is configured to handle very large files:

```
ftp> quote site setcos 45
200 COS set to 45.
```

Finally, we request the transfer itself. Observe that we don't put the file, as we would if we worked with a normal FTP. Instead we pput it. pput is the parallel version of put. You can see responses from four PFTP client nodes and the transfer commences.

```
ftp> pput test
200 Command Complete (4194304000, test, 0, 4, 4194304).
    Processing the multinode list, please wait.....
200 Command Complete.
```

```

200 Command Complete.
200 Command Complete.
200 Command Complete.
150 Transfer starting.
226 Transfer Complete.(moved = 4194304000).
4194304000 bytes sent in 37.12 seconds (107.75 Mbytes/s)
200 Command Complete.

```

The transfer rate was nearly 108 MB/s. This is very fast for FTP. In fact, you could not get such a high transfer rate with a normal sequential FTP.

Let us check that the file is indeed in HPSS:

```

ftp> ls -l
200 PORT command successful.
150 Opening ASCII mode data connection for file list.
-rw-r----- 1 gustav 1000 365684740 Oct 9 2001 FRIDAYZONE1.MPG
-rw-r----- 1 gustav 1000 361170948 Oct 9 2001 FRIDAYZONE2.MPG
-rw-r----- 1 gustav 1000 1164392 Jul 17 1999 IU-HPSS.tar.gz
drwxr-x--- 2 gustav 1000 512 Jul 10 2001 MPIO
drwxr-x--- 2 gustav 1000 512 Feb 13 2001 new
-rw-r----- 1 gustav ovpit 4194304000 Sep 4 16:57 test
226 Transfer complete.
391 bytes received in 0.08 seconds (4.90 Kbytes/s)
ftp> quit
221 Goodbye.
[gustav@ih1 gustav]$

```

Well, it is there, all 4 GBs of it.

Now, how to get it back?

Proceed as before, but this time use `pget` instead of `pput`. Here is the example. I begin by going back to my GPFS directory and delete files `test` and `test1` from it, then connect to HPSS as before.

```

[gustav@ih1 gustav]$ cd /N/gpfs/gustav
[gustav@ih1 gustav]$ ls
test test1
[gustav@ih1 gustav]$ rm *
[gustav@ih1 gustav]$ pftp_client hpss.iu.edu
Parallel block size set to 4194304.
Connected to hpss.iu.edu.

[ message of the day ]

220 hpss-s12.uits.iupui.edu FTP server
(HPSS 4.3 PFTPD V1.1.1 Wed Sep 11 15:18:05 EST 2002) ready.
Name (hpss.iu.edu:gustav): gustav
331 Password required for ../dce1.indiana.edu/gustav.
Password:
230 User ../dce1.indiana.edu/gustav logged in.
Remote system type is UNIX.
Using binary mode to transfer files.

[ other messages]

Multinode is Disabled.
ftp> cd /:/hpssonly/g/u/gustav
250 CWD command successful.

```

```

ftp> ls -l
200 PORT command successful.
150 Opening ASCII mode data connection for file list.
-rw-r----- 1 gustav 1000 365684740 Oct 9 2001 FRIDAYZONE1.MPG
-rw-r----- 1 gustav 1000 361170948 Oct 9 2001 FRIDAYZONE2.MPG
-rw-r----- 1 gustav 1000 1164392 Jul 17 1999 IU-HPSS.tar.gz
drwxr-x--- 2 gustav 1000 512 Jul 10 2001 MPI0
drwxr-x--- 2 gustav 1000 512 Feb 13 2001 new
-rw-r----- 1 gustav ovpit 4194304000 Sep 4 16:57 test
226 Transfer complete.
391 bytes received in 0.11 seconds (3.50 Kbytes/s)
ftp> multinode
Processing the multinode list, please wait.....
Multinode is on.
ftp> setpwidth 4
Parallel stripe width set to (4).
Processing the multinode list, please wait.....

```

Now I transfer the file back to AVIDD with the parallel version of `get`, i.e., with `pget`:

```

ftp> pget test
200 Command Complete (4194304000, test, 0, 4, 4194304).
Processing the multinode list, please wait.....
200 Command Complete.
200 Command Complete.
200 Command Complete.
200 Command Complete.
150 Transfer starting.
226 Transfer Complete.(moved = 4194304000).
4194304000 bytes received in 2 minutes,3.57 seconds (32.37 Mbytes/s)
200 Command Complete.
ftp> quit
221 Goodbye.
[gustav@ih1 gustav]$ ls -l
total 4096000
-rw-r--r-- 1 gustav ucs 4194304000 Sep 4 17:14 test
[gustav@ih1 gustav]$

```

And I get my file back. The return transfer rate was only about 32 MB/s, which is about 8 MB/s/node. This is markedly slower than 108 MB/s we got on writing to HPSS and is caused by the GPFS' poor performance on writes. Still, it is better than the 13 MB/s we saw in our previous experiments.

HPSS performs on these tests very well. You can see HPSS' performance on reads, uncontaminated by GPFS, if you drop the data on `/dev/null`:

```

ftp> multinode
Processing the multinode list, please wait.....
Multinode is on.
ftp> setpwidth 4
Parallel stripe width set to (4).
Processing the multinode list, please wait.....
ftp> pget test /dev/null
200 Command Complete (4194304000, test, 0, 4, 4194304).
Processing the multinode list, please wait.....
200 Command Complete.
200 Command Complete.

```



```
200 Command Complete.
200 Command Complete.
150 Transfer starting.
226 Transfer Complete.(moved = 4194304000).
4194304000 bytes received in 28.84 seconds (138.72 Mbytes/s)
200 Command Complete.
ftp>
```

The transfer rate on 4-way parallel reads from HPSS is nearly 140 MB/s – if only there was a matching sink on the AVIDD side to drop the data on.

Very special thanks are due to the Distributed Storage Systems Group (DSSG) administrators, who invested a lot of effort, as well as a lot of skills and knowledge into getting these transfer rates so high (from the HPSS side).

In real life you should expect IO transfer rates to vary. They will depend on various environmental factors such as system load, GPFS load, network load and HPSS load as well.

### Exercises

- 1 Obtain a window on a computational node with the `qsub -I` command (see section 4.3.1 for more information on `qsub -I`). Generate a 4GB GPFS file with `mkrandfile` and copy it to your HPSS “hpssonly” directory using `pftp_client`, `multinode` and `setpwidth 4`. What data transfer rate has been reported by PFTP?
- 2 Delete the GPFS file created in exercise 3.4.3, then copy its image from HPSS back onto GPFS using `pftp_client`, `multinode` and `sepwidth 4`. What data transfer rate has been reported by PFTP?
- 3 Repeat transfers to and from HPSS in order to observe fluctuations in transfer rates. What are they caused by?
- 4 Delete the HPSS copy of the file.
- 5 Exit the session on the computational node.



## Chapter 4

# Working with PBS

PBS stands for “Portable Batch System”. It is a suite of programs for submitting jobs to distributed computational resources, e.g., to clusters such as our AVIDD. There are many versions of PBS floating around. For example, there is OpenPBS developed originally for NASA and “Scalable OpenPBS” distributed by Superclusters.org, a site that is run by Cluster Resources, a company in Utah. Then there is “Professional PBS Pro” distributed by OpenPBS.org, which is one of many shopping windows of Altair Engineering, a company located in Michigan.

The version installed on AVIDD at present is “Scalable OpenPBS” with the Maui scheduler.

As I have already mentioned in section 3.3, you can find PBS manual entries in `/usr/local/man`, and, this is about all there is there in terms of user documentation.

In the following sections you will learn how to make sense of these manuals and how to use PBS on the AVIDD cluster.

### 4.1 PBS Configuration

The first thing we are going to look at is the PBS configuration. And here you want to find about the names of various queues that may have been configured on the system. The queues usually are associated with various resources. For example, there may be queues for short jobs and queues for long jobs. There may be queues for sequential jobs and queues for parallel jobs. There may be queues for exclusive jobs, i.e., jobs that want the whole node to themselves, and queues for jobs that don’t mind sharing nodes with other jobs. There may be queues for jobs requiring a lot of memory and jobs that don’t need that much and queues for various architectures as well, because PBS can be used to manage a heterogeneous cluster.

The command you can use to find about PBS configuration is `qstat`. In particular `qstat -q` will tell you about queues and their parameters.

Running this command on avidd-b.iu.edu returns:

```
[gustav@bh1 gustav]$ qstat -q

server: bh1

Queue          Memory CPU Time Walltime Node Run Que Lm  State
-----
bg             --    --    --    --    17  3 --   E R
-----
                                17  3

[gustav@bh1 gustav]$
```

This means that there is only one queue configured on avidd-b at the time I'm writing this tutorial. The queue has no memory limit, no CPU time limit, no wall time limit and no node number limit either.

You can use the same command to find about queues configured on the avidd-i.iu.edu as follows:

```
[gustav@bh1 gustav]$ ssh ih1 qstat -q

server: ih1

Queue          Memory CPU Time Walltime Node Run Que Lm  State
-----
bg             --    --    --    --   182 343 --   E R
-----
                                182 343

[gustav@bh1 gustav]$
```

Well, there is also only one queue configured there, which has the same name (but it is not the same queue, mind you, because it is configured on a different system). That queue has no memory, cpu time, wall clock time and no node number limits either.

This is not common on HPC systems, unless you have one that's used little or by a small number of users. You will find, if you connect to our SP, that there are great many queues configured there.

You can see what jobs run on the system currently by typing `qstat` without any options. And so, for example, on the avidd-b, we have:

```
[gustav@bh1 gustav]$ qstat
Job id      Name          User          Time Use S Queue
-----
10177.bh1   calmob       surajago      107:06:4 R bg
11137.bh1   nokkmmcalmob surajago      0 Q bg
11617.bh1   klv_ptanal_lopt kevidale     31:54:37 R bg
11628.bh1   klv_ptanal_lopt kevidale     31:51:01 R bg
11629.bh1   klv_ptanal_lopt kevidale     31:43:24 R bg
11630.bh1   klv_ptanal_dach kevidale     32:00:26 R bg
11635.bh1   yeast3      jfan         0 Q bg
11970.bh1   c8_scan     steige       17:41:52 R bg
12020.bh1   klv_ptanal_dach kevidale     11:21:29 R bg
12082.bh1   helB1      mkohtani     08:33:18 R bg
12150.bh1   STDIN      ivdgl        00:00:00 R bg
12151.bh1   STDIN      ivdgl        00:00:00 R bg
12152.bh1   STDIN      ivdgl        00:00:00 R bg
```



It tells us that there are 17 jobs running at present and 3 queued ones. The 3 queued jobs are not queued because of lack of resources probably. Rather they may depend on some of the jobs that are currently executing.

There are many more options to `qstat`, and we are going to learn about some of them as we need them. But `qstat` and `qstat -q` will do for the time being.

## 4.2 Submitting, Inspecting and Cancelling PBS Jobs

In this section we are going to submit our first PBS job.

A PBS job is simply a shell script, possibly with some PBS directives. PBS directives look to shell like shell comments, so it ignores them, but PBS picks them up and processes the job accordingly.

We are going to start by submitting a very simple shell script, which doesn't have any PBS directives.

```
[gustav@bh1 PBS]$ pwd
/N/B/gustav/PBS
[gustav@bh1 PBS]$ ls
job.sh
[gustav@bh1 PBS]$ cat job.sh
#!/bin/bash
hostname
date
exit 0
[gustav@bh1 PBS]$
```

The shell is submitted with the command `qsub`:

```
[gustav@bh1 PBS]$ qsub job.sh
12248.bh1.avidd.iu.edu
[gustav@bh1 PBS]$
```

The command returns the job id in return. This is the id (but with the domain name stripped off) that appears in the first column of `qstat` listing. `qsub` returns the standard output produced by the job on a file in the same directory from the job was submitted. Standard error is also returned on another file in the same directory:

```
[gustav@bh1 PBS]$ ls
job.sh job.sh.e12248 job.sh.o12248
[gustav@bh1 PBS]$
```

The name of the output file is produced by appending “.o” followed by the job id number to the name of the script submitted to `qsub`. The name of the error file is produced by appending “.e” followed by the job id number to the name of the script. In this case the error file is empty and the standard output file contains:

```
[gustav@bh1 PBS]$ cat job.sh.o12248
bc89
Sun Sep 7 16:27:25 EST 2003
[gustav@bh1 PBS]$
```

and this tells us that the job was executed on bc89.avidd.iu.edu.

Now let me show you how you can submit a very large number of jobs automatically. We begin by executing the following simple multiline shell expression:

```
[gustav@bh1 PBS]$ i=10; while [ $i -gt 0 ]
> do
>   echo $i
>   i='expr $i - 1'
> done
10
9
8
7
6
5
4
3
2
1
[gustav@bh1 PBS]$
```

The expression works as follows. First we initialize `i` to 10. Then we start the `while` loop, which checks if the value of `i`, i.e., `$i` is still greater than zero. Within the body of the loop we print the value of `i` and then we decrement it by 1. The loop stops when the value of `i` becomes zero.

Now we are going to repeat the same multiline command, but this time we are going to insert `qsub job.sh` within the body of the loop:

```
[gustav@bh1 PBS]$ i=10; while [ $i -gt 0 ]
> do
>   echo $i
>   qsub job.sh
>   i='expr $i - 1'
> done
10
12249.bh1.avidd.iu.edu
9
12250.bh1.avidd.iu.edu
8
12251.bh1.avidd.iu.edu
7
12252.bh1.avidd.iu.edu
6
12253.bh1.avidd.iu.edu
5
12254.bh1.avidd.iu.edu
4
12255.bh1.avidd.iu.edu
3
12256.bh1.avidd.iu.edu
2
12257.bh1.avidd.iu.edu
1
12258.bh1.avidd.iu.edu
[gustav@bh1 PBS]$
```

These jobs are very small and they should execute very quickly:

```
[gustav@bh1 PBS]$ ls
0                job.sh.e12251  job.sh.e12256  job.sh.o12250  job.sh.o12255
job.sh           job.sh.e12252  job.sh.e12257  job.sh.o12251  job.sh.o12256
job.sh.e12248   job.sh.e12253  job.sh.e12258  job.sh.o12252  job.sh.o12257
job.sh.e12249   job.sh.e12254  job.sh.o12248  job.sh.o12253  job.sh.o12258
job.sh.e12250   job.sh.e12255  job.sh.o12249  job.sh.o12254
[gustav@bh1 PBS]$
```

Let us see what we can find in the output files:

```
[gustav@bh1 PBS]$ cat job.sh.o*
bc89
Sun Sep  7 16:27:25 EST 2003
bc89
Sun Sep  7 16:43:18 EST 2003
bc68
Sun Sep  7 16:43:20 EST 2003
bc67
Sun Sep  7 16:43:20 EST 2003
bc67
Sun Sep  7 16:43:21 EST 2003
bc66
Sun Sep  7 16:43:22 EST 2003
bc65
Sun Sep  7 16:43:23 EST 2003
bc65
Sun Sep  7 16:43:24 EST 2003
bc63
Sun Sep  7 16:43:25 EST 2003
bc61
Sun Sep  7 16:43:26 EST 2003
bc61
Sun Sep  7 16:43:27 EST 2003
[gustav@bh1 PBS]$
```

We can see that the jobs were sent to various nodes, depending on which were available.

You can easily submit thousands of jobs this way. But it is better to write a job submitting script first, then test it with the `qsub` line commented out, just to make sure that your counting and, *most importantly*, stopping is going to work as you expect, and only then uncomment the `qsub` line and run the script again.

Our job executes so fast that we can hardly catch it in action. We are going to slow it down by letting it *sleep* for a hundred seconds before exiting.

Here is our modified version.

```
[gustav@bh1 PBS]$ cat job.sh
#!/bin/bash
hostname
date
sleep 100
date
exit 0
[gustav@bh1 PBS]$
```

And just to make sure that it's not going to hang forever, we are going to execute it interactively and check that it sleeps for 100 seconds only:



```
[gustav@bh1 PBS]$ time ./job.sh
bh1
Sun Sep  7 16:53:41 EST 2003
Sun Sep  7 16:55:21 EST 2003

real    1m40.029s
user    0m0.000s
sys     0m0.010s
[gustav@bh1 PBS]$
```

This works just fine: the job took 1 minute and 40 seconds, which is 100 seconds, to execute. Now we are going to submit it with `qsub` and we are going to look at it with `qstat`.

```
[gustav@bh1 PBS]$ qsub job.sh
12259.bh1.avidd.iu.edu
[gustav@bh1 PBS]$ qstat 12259.bh1
```

Job id	Name	User	Time Use	S	Queue
12259.bh1	job.sh	gustav		0 R	bg

```
[gustav@bh1 PBS]$
```

You can look at just the job that is of interest to you, by giving its ID as an argument to `qstat`. The “R” in the 5th column indicates that the job is running. The 5th column is the status column the other values you can see there are

**E** the job is exiting after having run

**H** the job is held – this means that it is not going to run until it is *released*

**Q** the job is queued and will run when the resources become available

**R** the job is running

**T** the job is being transferred to a new location – this may happen, e.g., if the node the job had been running on crashed

**W** the job is waiting – you can submit jobs to run, e.g., after 5PM

Now, suppose that we have submitted a job and it is waiting in the queue. Its status is going to be Q. Then we discover that there may be a problem with the job, but we aren’t sure. What are we to do? Well, we can “place a hold” on the job by issuing the command `qhold`:

```
[gustav@bh1 PBS]$ qsub job.sh
12259.bh1.avidd.iu.edu
[gustav@bh1 PBS]$ qhold 12259.bh1
```

If the job is still in the queue, it’ll be put on hold. Its status will change from Q to H. You can now check if the submitted program and data are OK, and if they are, you can *release* the hold by calling `qrls`:

```
[gustav@bh1 PBS]$ qrls 12259.bh1
```

This will make the job eligible to run again. Its status will change from H back to Q.

Another scenario arises when the job that you may have some doubts about is already running. You can send signals to jobs running under PBS by calling the command `qsig`. The synopsis of `qsig` is

```
qsig -s signal job_id
```

The signal can be given either as a number, e.g., 2, or as a signal name, e.g., SIGINT. Numbers and names of signals are described in section 7 of the Linux manual. You can read the corresponding manual entry by issuing the command:

```
[gustav@bh1 gustav]$ man 7 signal
```

In the following example, we are going to submit our job that sleeps for 100 seconds and then we are going to send an interrupt signal to it (it is signal number 2 that is called SIGINT). This signal is generated by pressing control-C on normally configured Linux keyboard.

```
[gustav@bh1 PBS]$ qsub job.sh
12351.bh1.avidd.iu.edu
[gustav@bh1 PBS]$ qsig -s SIGINT 12351.bh1
[gustav@bh1 PBS]$ ls
job.sh job.sh.e12351 job.sh.o12351
[gustav@bh1 PBS]$ cat job.sh.o12351
bc67
Sun Sep 7 17:19:35 EST 2003
[gustav@bh1 PBS]$ qstat 12351.bh1
qstat: Unknown Job Id 12351.bh1.avidd.iu.edu
[gustav@bh1 PBS]$
```

The job has indeed been killed.

It is usually better to send signals other than SIGKILL (signal number 9), because SIGKILL cannot be caught and normally you want to catch a signal and act on it, e.g., clean the mess before exiting.

A more ruthless way to get rid of an unwanted job is to run `qdel` on it. This command deletes a job from PBS. If the job runs, the command sends SIGKILL to it. If the job is merely queued, the command deletes it from the queue. Here's the example:

```
[gustav@bh1 PBS]$ qsub job.sh
12390.bh1.avidd.iu.edu
[gustav@bh1 PBS]$ qdel 12390.bh1
[gustav@bh1 PBS]$ ls
job.sh job.sh.e12390 job.sh.o12390
[gustav@bh1 PBS]$ cat job.sh.o12390
bc89
Sun Sep 7 17:26:01 EST 2003
[gustav@bh1 PBS]$
```

PBS commands covered in this section

**qsub** submit a job for execution

**qstat** examine the status of a job (we have discussed what this status may be)

**qhold** put a job on hold

**qrls** release a job

**qsig** send a signal to a job

**qdel** delete a job

## 4.3 Specification of PBS Jobs

So far we have used PBS in its simplest form relying on defaults. We never had to specify queues or other job parameters. Although some of the specifications may be superfluous on the AVIDD cluster right now, because there is only one queue there and the queue has no limits imposed on it, nevertheless, you need to learn how to describe PBS jobs more elaborately, in case you want to use other facilities, e.g., NCSA, or in case AVIDD gets used more heavily in future and its administrators will be forced to configure a richer queueing environment. Furthermore we are going to learn some other tricks in this section that will be useful for conditional execution of various jobs, i.e., jobs that may depend on other jobs to run correctly.

### 4.3.1 Interactive PBS Jobs

Normally when you call `qsub` PBS grabs your script and goes away. You get the prompt returned to you almost immediately and whatever output is produced from your job goes into a designated file. But what if you want to run an interactive job, e.g., a matlab session? This is not, in general, the right way to use supercomputers, because interactive jobs waste enormous amount of CPU time, mostly because humans are so slow. Even if you could type 1000 characters per second, you would still waste enormous amount of CPU time, because computers can easily type 500,000,000 characters per second or more (and high end graphic displays, the so called frame buffers, can display the characters this fast, but only as pixels).

The way to run such an interactive job is to call `qsub -I`. The `-I` option will make PBS arrange for an interactive login session on an available AVIDD node and your TTY will be reconnected to that session. On exiting the session you'll get back to the original TTY. Here is how it works:

```
[gustav@bh1 gustav]$ hostname
bh1
[gustav@bh1 gustav]$ pwd
/N/B/gustav
[gustav@bh1 gustav]$ qsub -I
qsub: waiting for job 12529.bh1.avidd.iu.edu to start
qsub: job 12529.bh1.avidd.iu.edu ready

[gustav@bc55 gustav]$ hostname
bc55
[gustav@bc55 gustav]$ pwd
/N/B/gustav
[gustav@bc55 gustav]$ exit
qsub: job 12529.bh1.avidd.iu.edu completed
[gustav@bh1 gustav]$
```

I start from the head node, bh1, and request an interactive session. PBS takes my request, assigns it a job ID of 12529.bh1, and then gives me a shell on bc55. On exiting the shell on bc55 the PBS job terminates and I get back to the head node, bh1.

The computational nodes of the AVIDD cluster are configured for computation only. They have no X11 installed on them. This means that you cannot run any X11 tools on those nodes and you cannot even run emacs there, which is a pity, because emacs can be used as a computational tool. It is, after all, a Lisp machine emulator. But you can compile and run your own X11 program on computational nodes, because they have the full X11 library installed, and you can compile and run your own emacs too. And you can copy an existing X11 binary from `/usr/X11R6/bin` to your `$HOME/bin` too.

When you run X11 tools on computational nodes, the traffic is routed through `bh3.uits.indiana.edu` (on `avidd-b`), so that you should either add `bh3.uits.indiana.edu` to the list of hosts allowed to display windows on your X11 server, or – a better practice – use Xauthority.

Here is a simple example that shows how to do this:

1. On my desktop X11 server I allow access to `bh3` with `xhost +bh3.uits.indiana.edu`.
2. I copy `xclock` and `xterm` binary on the head node from `/usr/X11R6/bin` to my `$HOME/bin`.
3. Now I enter the interactive PBS session:

```
[gustav@bh1 gustav]$ cp /usr/X11R6/bin/xclock bin/xclock
[gustav@bh1 gustav]$ cp /usr/X11R6/bin/xterm bin/xterm
[gustav@bh1 gustav]$ qsub -I
qsub: waiting for job 12546.bh1.avidd.iu.edu to start
qsub: job 12546.bh1.avidd.iu.edu ready

[gustav@bc55 gustav]$ export DISPLAY=woodlands.tqc.iu.edu:0.0
[gustav@bc55 gustav]$ xclock &
[1] 5709
[gustav@bc55 gustav]$ xterm &
[2] 5710
[gustav@bc55 gustav]$ exit
qsub: job 12546.bh1.avidd.iu.edu completed
[gustav@bh1 gustav]$
```

Exiting from `bc55` kills both `xclock` and `xterm`.

Since you have copied `xterm` to your private `bin` already, you can get yourself a more functional interactive window on a computational node simply by submitting a script, which calls `xterm`, without the `-I` option. Here is the example of the script:

```
[gustav@bh1 PBS]$ cat xterm.sh
#!/bin/bash
xterm -display woodlands.tqc.iu.edu:0.0
exit 0
[gustav@bh1 PBS]$ qsub xterm.sh
12550.bh1.avidd.iu.edu
[gustav@bh1 PBS]$
```

Submitting the script to PBS will produce, after a short while, an `xterm` window running on a PBS allocated node on your X11 server. Exiting the shell in the window terminates the job.

### 4.3.2 Not Quite Interactive PBS Jobs

As I have mentioned in the previous section, very often you don't really need to make an interactive contact with the program you want to run under PBS, even if the program requires some interactive input. If the input does not depend on the program's output, you can simply use the "here-input" feature of the shell.

The "here-input" is constructed in the following way:

```
$ my_command << EOF
  one_line_of_input
  another_line_of_input
EOF
```

Here is an example of a simple script that utilizes this feature:

```
[gustav@bh1 PBS]$ cat bc.sh
#!/bin/bash
bc << EOF
2 + 2
3 - 1
scale=10
3.14 - 2.17
EOF
exit 0
[gustav@bh1 PBS]$ ./bc.sh
4
2
.97
[gustav@bh1 PBS]$
```

The program `bc` is an interactive UNIX calculator. But here, instead of typing the stuff interactively, we tell shell to type it for us: everything until the line that begins with `EOF`.

We can submit this job to PBS and it will work the same, with the only difference that the output is going to be written on a file:

```
[gustav@bh1 PBS]$ qsub bc.sh
12626.bh1.avidd.iu.edu
[gustav@bh1 PBS]$ ls
bc.sh  bc.sh.e12626  bc.sh.o12626  job.sh  xterm.sh
[gustav@bh1 PBS]$ cat bc.sh.o12626
4
2
.97
[gustav@bh1 PBS]$
```

### 4.3.3 Simple PBS Directives

If you don't like the names of PBS output and error files, if you don't want your job to go to the default queue, if you have any other special requests, you can specify them by the means of `qsub` command line switches or PBS directives.

PBS directives are exactly the same as the `qsub` command line switches, but they are passed to PBS through the script comment lines. For example, in order to change the name of the output file you can use

```
qsub -o file_name
```

or you can insert

```
#PBS -o file_name
```

in the preamble of the script. The effect will be the same.

The three letters, PBS, that mark the directive can be altered too, with the command line switch

```
-C new_string
```

But it's better not to tinker with it, since it may lead to unnecessary confusion down the road.

To rename the error file use the directive:

```
#PBS -e file_name
```

The default name of the job, which shows in the second column of the `qstat` listing, is the name of the submitted file. If you enter the job interactively, then the name evaluates to `STDOUT`. Interactive job submission is not advisable, because mistakes are easy to make, so I haven't talked about it, but if you're curious, this is how it is done:

```
[gustav@bh1 PBS]$ qsub
hostname
date
exit 0
^D
12672.bh1.avidd.iu.edu
[gustav@bh1 PBS]$
```

The output files are then called `STDIN.o12672` and `STDIN.e12672`:

```
[gustav@bh1 PBS]$ ls
STDIN.e12672 STDIN.o12672 bc.sh job.sh xterm.sh
[gustav@bh1 PBS]$ cat STDIN.o12672
bc68
Mon Sep  8 21:36:52 EST 2003
[gustav@bh1 PBS]$
```

You can change the name of the job with the directive

```
#PBS -N job_name
```

The submitted script is interpreted using the user's login shell by default. PBS *does not* read the

```
#!/bin/bash
```

line the scripts usually begin with. But this can be altered by using the directive

```
#PBS -S /bin/bash
```

You can use also perl or any other scripting language as long as its comment character is the sharp sign, # (some Schemes allow for the sharp to be used as a comment character too). PBS will scan all the initial comment lines for PBS directives *until* the first executable line. Any directives embedded in the script *after* some executable lines will be ignored.

If you don't specify the queue name, the job gets submitted to the default queue. The default queue at most institutions is a short job queue. So normally you have to specify the queue name for longer jobs. This is done with the directive

```
#PBS -q bg
```

Let us wrap up the `-o`, `-e`, `-N`, `-S` and `-q` directives into the following script.

```
[gustav@bh1 PBS]$ cat simple.sh
#PBS -S /bin/bash
#PBS -N host
#PBS -o host_out
#PBS -e host_err
#PBS -q bg
hostname
date
exit 0
[gustav@bh1 PBS]$ qsub simple.sh
12673.bh1.avidd.iu.edu
[gustav@bh1 PBS]$ ls
STDIN.e12672 bc.sh      host_out  simple.sh
STDIN.o12672 host_err  job.sh   xterm.sh
[gustav@bh1 PBS]$ cat host_out
bc68
Mon Sep  8 21:58:06 EST 2003
[gustav@bh1 PBS]$
```

Another two useful PBS directives result in e-mail being sent to you when the job starts, or terminates or is aborted. The corresponding option is `-m` and it must be followed by

**n** in which case no mail is sent

**a** in which case mail is sent when the job gets aborted

**b** in which case mail is sent when the job begins execution

**e** in which case mail is sent when the job terminates execution

Option **a** is the default. **a**, **b** and **e** can be combined, e.g., **abe**. The corresponding PBS directive looks as follows:

```
#PBS -m abe
```

Now, who should the mail be sent to? This you can specify by using the `-M` directive, e.g.,

```
#PBS -M gustav@indiana.edu
```

You can also send mail to several users, e.g.,

```
#PBS -M gustav@indiana.edu,stewart@iu.edu
```

Let us see how this works in combination with our simple job described above.

```
[gustav@bh1 PBS]$ cat simple.sh
#PBS -S /bin/bash
#PBS -N host
#PBS -o host_out
#PBS -e host_err
#PBS -q bg
#PBS -m abe
#PBS -M gustav@indiana.edu
```

```

hostname
date
exit 0
[gustav@bh1 PBS]$ qsub simple.sh
12866.bh1.avidd.iu.edu
[gustav@bh1 PBS]$ ls
bc.sh host_err host_out job.sh nodes.sh simple.sh xterm.sh
[gustav@bh1 PBS]$ cat host_out
bc88
Wed Sep 10 16:15:46 EST 2003
[gustav@bh1 PBS]$

```

Mail sent to `gustav@indiana.edu` is automatically redirected to my workstation, `woodlands.tqc.iu.edu`. Let us have a look then if there are any messages there sent from AVIDD. Here is the updated mail listing from my emacs session:

```

 1 10-Sep escience-admin@cs.indiana [108] \
    [Escience] Computer's ability to verify proof is an illusion
 2- 10-Sep to: qc@WOODLANDS.tqc.iu.e [62] \
    Quantum dynamics of a single vortex
 3- 10-Sep adm@bh1.uits.indiana.edu [36] \
    PBS JOB 12866.bh1.avidd.iu.edu
 4- 10-Sep adm@bh1.uits.indiana.edu [31] \
    PBS JOB 12866.bh1.avidd.iu.edu

```

Two messages have arrived from `bh1` indeed. The messages may arrive out of order, because this is how e-mail usually works. In this case the second message (number 4 in the emacs listing) corresponds to the job beginning its execution:

```

Envelope-to: gustav@woodlands.tqc.iu.edu
Delivery-date: Wed, 10 Sep 2003 16:16:19 -0500
Date: Wed, 10 Sep 2003 16:15:46 -0500
From: adm <adm@bh1.uits.indiana.edu>
To: gustav@indiana.edu
Subject: PBS JOB 12866.bh1.avidd.iu.edu

```

```

PBS Job Id: 12866.bh1.avidd.iu.edu
Job Name: host
Begun execution

```

and the first message corresponds to the termination of the job:

```

Envelope-to: gustav@woodlands.tqc.iu.edu
Delivery-date: Wed, 10 Sep 2003 16:16:16 -0500
Date: Wed, 10 Sep 2003 16:15:46 -0500
From: adm <adm@bh1.uits.indiana.edu>
To: gustav@indiana.edu
Subject: PBS JOB 12866.bh1.avidd.iu.edu

```

```

PBS Job Id: 12866.bh1.avidd.iu.edu
Job Name: host
Execution terminated
Exit_status=0
resources_used.cput=00:00:00
resources_used.mem=0kb
resources_used.vmem=0kb
resources_used.walltime=00:00:00

```

A good idea is to have the message about your job having been aborted or terminated sent to your cellular phone, if you have this kind of service. This way the message will reach you wherever you may be.



### 4.3.4 Jobs Dependent On Other Jobs

In this section we are going to discuss jobs that may depend on other jobs. In order to concoct such jobs we are going to begin by writing a very, very, very simple UNIX program that reads a file in chunks of various sizes and counts number of bytes in it. Then we are going to use it, together with `mkrandfile` in PBS scripts.

Another reason for writing the program first is to give you an opportunity to exercise your UNIX and C programming skills some more.

#### Program `xrandfile`

Here is the listing of the program:

```

/*
 * Read an existing file full of random integers and perform
 * some arbitrary operation on it.
 *
 * %Id: xrandfile.c,v 1.2 2003/09/12 21:28:33 gustav Exp %
 *
 * %Log: xrandfile.c,v %
 * Revision 1.2 2003/09/12 21:28:33 gustav
 * Converted mkrandfile to xrandfile: an application that reads the file and
 * counts the number of bytes in it.
 *
 * Revision 1.1 2003/09/12 17:55:12 gustav
 * Initial revision
 *
 */

#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<errno.h>
#define BLOCK_SIZE 1048576
#define SYNOPSIS printf ("synopsis: %s -f <file> [ -l <blocks_per_read> ]\n", argv[0])
#define CLOSE if (fclose(fp) != 0) { perror (name); exit (5); }

main(argc, argv)
    int argc;
    char *argv[];
{
    FILE *fp;
    int *junk, size_of_junk, n_of_items, blocks_per_read = 1;
    long long count;

    /* Note: we are working in a 32-bit architecture. In this architecture
     long is the same as int (see limits.h) and LONG_MAX = INT_MAX
     = 2^31 - 1 = 2147483647. We have to use (long long), for which
     LLONG_MAX = 2^63 - 1 = 9223372036854775807. This is going to
     have some very limiting ramifications, when we get to use ROMIO.
    */

    /* variables for reading the command line */

```

```

extern char *optarg;
char *name = NULL;
int c;

/* error handling */

extern int errno;

size_of_junk = blocks_per_read * BLOCK_SIZE * sizeof(int);

while ((c = getopt(argc, argv, "f:l:h")) != EOF)
    switch(c) {
        case 'f':
            name = optarg;
            (void)printf("reading %s\n", name);
            break;
        case 'l':
            if ((1 != sscanf (optarg, "%d", &blocks_per_read)) ||
                (blocks_per_read < 1)) {
SYNOPTIS;
exit(1);
            }
            else {
size_of_junk = blocks_per_read * BLOCK_SIZE * sizeof(int);
printf("reading in chunks of size %d bytes\n", size_of_junk);
            }
            break;
        case 'h':
            SYNOPTIS;
            exit(0);
        case '?':
            SYNOPTIS;
            exit(1);
    }

if (name == NULL) {
    SYNOPTIS;
    exit(2);
}

if (!(junk = (int*) malloc(size_of_junk))) {
    perror ("malloc");
    exit(3);
}
else
    printf("allocated %d bytes to junk\n", size_of_junk);

if (!(fp = fopen(name, "r"))) {
    perror (name);
    exit(4);
}

count = 0LL;
while (!feof(fp)){
    n_of_items = fread (junk, sizeof(int), blocks_per_read * BLOCK_SIZE, fp);
    if (ferror(fp)){

```

```

        perror("fread");
        CLOSE;
        exit (6);
    }
    count = (long long) (count + n_of_items);
}

printf ("read %lld bytes\n", (long long)(count * sizeof(int)));
free((void *)junk);
CLOSE;
exit (0);
}

```

Let me explain briefly how the program works. We begin by reading the command line first. The options are as before, but this time the option `-l` has a different meaning. It specifies the size of the reading buffer in units of 4 MB. Because the size of the reading buffer is no longer predefined, we have to allocate it dynamically.

Observe that when I process the option `-l`, I use the number of parameters matched by function `sscanf` in order to detect a possible error. The line of the code that prints the synopsis of the program is now represented by a macro `SYNOPSIS`, which is defined in the preamble of the program.

After we have finished with reading the command line and checking for possible input errors, we allocate space for the `junk` array by calling `malloc`. If `malloc` returns `NULL`, which means that it has failed to allocate requested space, we print the error message and exit. The amount of requested space in bytes is stored on the variable `size_of_junk`, which is calculated at the beginning of the program and then recalculated again if the `-l` option is used. This way we set a default value for it, which is 4 MB.

Having allocated the space, we open the file for *reading* this time.

Then we initialize the variable `count`, which will be used to count number of integers found on the file, to `0LL`. This notation means “zero in the long-long format”. We have to use “long-long” for `count`, in order to count number of integers that may be above `INT_MAX`, which is only  $2^{31} - 1 = 2147483647$ , i.e., about 2 billion. Our GPFS files may contain more than 2 billion integers.

The reading of the file is done within the `while` loop, which checks for the *end of file* at the top of the loop. We read chunks of data of size `size_of_junk` with `fread`, which returns number of items, in this case they are 32-bit integers, read in a single operation. The number of items read is then added to `count`.

But before we increment `count` we check if the read has not returned an error. This is not done by investigating the value returned by `fread`, but by calling a special function `ferror`. This function operates on the file pointer, `fp`, as does the function that detects the end-of-file condition, `feof`. If there has been an error, we print the error message, then attempt to close the file (this is what the `CLOSE` macro does), and exit raising error flag 6.

Assuming that all has gone well, we exit the `while` loop cleanly and print number of *bytes* read from the file on standard output. Then we *free* the allocated storage, close the file, and exit.

The Makefile for this program looks exactly the same as the Makefile for `mkrandfile`, with the exception that the name `mkrandfile` is replaced with `xrandfile`.

The manual entry for the program looks as follows:

```

XRANDFILE(1)          I590 Programmer's Manual          XRANDFILE(1)

NAME
    xrandfile - read a file in chunks of various sizes and
    count number of bytes in it

SYNOPSIS
    xrandfile -f filename [ -l blocks-per-read ] [-h]

DESCRIPTION
    xrandfile reads a file in chunks of various sizes and
    counts number of bytes in it. Can be used for disk-based
    benchmarks.

OPTIONS
    -f      This parameter must be followed by input filename,
            e.g. foo.

    -l      This parameter is optional. It must be followed by
            an integer, which specifies the number of 4MB
            blocks per read. The default is one 4MB block per
            read.

    -h      Print a brief help message.

DIAGNOSTICS
    Self explanatory and numerous.

EXAMPLES
    $ xrandfile -f test -l 10
    reading test
    reading in chunks of size 41943040 bytes
    allocated 41943040 bytes to junk
    read 419430400 bytes
    $ ls -l test
    -rw-r--r--  1 gustav  ucs      419430400 Sep 12 16:20
    test

BUGS
    Command line processing may be fragile. There is no check-
    ing if the requested buffer size does not exceed INT_MAX,
    although malloc may catch this.

AUTHOR
    Zdzislaw Meglicki <gustav@indiana.edu>

I590/7462          SEPTEMBER 2003          XRANDFILE(1)

```

### A Four Part Job

We are now going to combine `mkrandfile` and `xrandfile` to construct a PBS job that has four distinct parts:

1. Prepare a directory on the AVIDD GPFS.
2. Generate a data file in the directory.
3. Process the data file.
4. Clean up the GPFS directory and exit.

This is a simplistic prototype of a typical supercomputer job. The “generate a data file” part may include staging the file from, e.g., HPSS, prior to executing other parts of the code. The “clean up” part may include uploading a modified data file to HPSS and deleting any unnecessary junk that the job may have generated on the GPFS.

Here is the script:

```
[gustav@bh1 PBS]$ cat process.sh
#PBS -S /bin/bash
#PBS -N process
#PBS -o process_out
#PBS -e process_err
#PBS -q bg
#PBS -m abe
#PBS -M gustav@indiana.edu
#
# Prepare a directory on the AVIDD GPFS.
[ -d /N/gpfs/gustav ] || mkdir /N/gpfs/gustav
cd /N/gpfs/gustav
rm -f test
# Generate a data file in the directory.
mkrandfile -f test -l 100
# Process the data file.
xrandfile -f test -l 4
# Clean up the GPFS directory and exit.
rm -f test
exit 0
[gustav@bh1 PBS]$
```

Let us submit the script and inspect the output of the job:

```
[gustav@bh1 PBS]$ qsub process.sh
13744.bh1.avidd.iu.edu
[gustav@bh1 PBS]$ qstat 13744.bh1.avidd.iu.edu
Job id      Name      User      Time Use S Queue
-----
13744.bh1   process   gustav    0 R bg
[gustav@bh1 PBS]$ qstat 13744.bh1.avidd.iu.edu
qstat: Unknown Job Id 13744.bh1.avidd.iu.edu
[gustav@bh1 PBS]$ cat process_out
writing on test
writing 100 blocks of 1048576 random integers
reading test
reading in chunks of size 16777216 bytes
```

```
allocated 16777216 bytes to junk
read 419430400 bytes
[gustav@bh1 PBS]$
```

Well, it all works as expected.

In the following sections we are going to learn how we can submit a multi-stage task like this in the form of several mutually dependent PBS jobs.

### Submitting Jobs From Within Jobs

In this section we are going to split the job discussed in section 4.3.4 into four separate jobs. The first job will prepare the GPFS directory and having finished its task, it will submit the second job. The second job will then generate the data file, and having done so it will submit the third job. The third job will process the data file and then it will submit the fourth job, which will clean up and exit the sequence. The jobs are constructed to be run on the IUPUI cluster, `avidd-i.iu.edu`.

Here is what the first job script looks like:

```
[gustav@ih1 PBS]$ cat first.sh
#PBS -S /bin/bash
#PBS -N first
#PBS -o first_out
#PBS -e first_err
#PBS -q bg
#
# first.sh
#
# Prepare a directory on the AVIDD GPFS.
[ -d /N/gpfs/gustav ] || mkdir /N/gpfs/gustav
cd /N/gpfs/gustav
rm -f test
echo "/N/gpfs/gustav prepared and cleaned."
# Now submit second.sh.
ssh ih1 "cd PBS; /usr/pbs/bin/qsub second.sh"
echo "second.sh submitted."
# Exit cleanly.
exit 0
[gustav@ih1 PBS]$
```

The new element in this job is the line:

```
ssh ih1 "cd PBS; /usr/pbs/bin/qsub second.sh"
```

Remember that the job will not run on the head node. It will run on a computational node. But the PBS on the AVIDD cluster is configured so that you cannot submit jobs from computational nodes. So here we have to execute `qsub` as a remote command on the IUPUI head node `ih1` by using the secure shell, since this is the only remote execution shell supported on the cluster.

The first command passed to `ssh` is “`cd PBS`”. On having made the connection the secure shell will land me in my home directory. But I don’t want to submit the job from there, because then the job output and error files will be generated in my home directory too. Instead I want *all* output and error files to be written on my `~/PBS` subdirectory. So we go to `~/PBS` first.

Then we submit the job. Observe that I use the full path name of the `qsub` command. The default `bash` configuration on the AVIDD cluster is such that the remote shell cannot find `qsub` otherwise. This, of course, I could fix by tweaking my own environment until it does (the `PATH` should normally be defined on `.bashrc`, not on `.bash_profile`), but it is a good practice to specify the full path of the command in this context anyway.

The script `second.sh` submitted by `first.sh` looks as follows:

```
[gustav@ih1 PBS]$ cat second.sh
#PBS -S /bin/bash
#PBS -N second
#PBS -o second_out
#PBS -e second_err
#PBS -q bg
#PBS -j oe
#
# second.sh
#
# The AVIDD GPFS directory should have been prepared by first.sh.
# Generate the data file.
cd /N/gpfs/gustav
time mkrandfile -f test -l 1000
ls -l test
echo "File /N/gpfs/gustav/test generated."
# Now submit third.sh.
ssh ih1 "cd PBS; /usr/pbs/bin/qsub third.sh"
echo "third.sh submitted."
# Exit cleanly.
exit 0
[gustav@ih1 PBS]$
```

There is only one novelty in this script, which you haven't seen yet. I am using a new PBS directive:

```
#PBS -j oe
```

This directive merges the standard error and standard output and writes both on the standard output file. If we were to use

```
#PBS -j eo
```

the two streams would be merged too, and the output would be written on the standard error file instead.

The reason I want both streams merged in this case is because the UNIX command `time` writes its diagnostics, i.e., the amount of CPU and wall clock time used by the program, on standard error. But I want this to be written together with the length of the file generated on standard output, in case I want to check the IO.

After this script has finished generating the file, it will submit the third script, called `third.sh`. Here is what the third script looks like:

```
[gustav@ih1 PBS]$ cat third.sh
#PBS -S /bin/bash
#PBS -N third
#PBS -o third_out
#PBS -e third_err
```

```

#PBS -q bg
#PBS -j oe
#
# third.sh
#
# Process the data file generated by second.sh.
cd /N/gpfs/gustav
time xrandfile -f test -l 4
echo "File /N/gpfs/gustav/test processed."
# Submit fourth.sh.
ssh ih1 "cd PBS; /usr/pbs/bin/qsub fourth.sh"
echo "fourth.sh submitted."
# Exit cleanly.
exit 0
[gustav@ih1 PBS]$

```

Here I have also requested that standard error and standard output streams be merged.

And finally the last, fourth script, which is called `fourth.sh`:

```

[gustav@ih1 PBS]$ cat fourth.sh
#PBS -S /bin/bash
#PBS -N fourth
#PBS -o fourth_out
#PBS -e fourth_err
#PBS -q bg
#
# fourth.sh
#
# Clean up everything in the GPFS directory
cd /N/gpfs/gustav
rm -f test
echo "Directory /N/gpfs/gustav cleaned."
exit 0
[gustav@ih1 PBS]$

```

Here is how to work all this. You submit the whole sequence on the IUPUI head node `ih1` by submitting just the first of the four scripts. The rest takes care of itself:

```

[gustav@ih1 PBS]$ qsub first.sh
13658.ih1.avidd.iu.edu
[gustav@ih1 PBS]$ while sleep 10
> do
>   qstat | grep gustav
> done
13659.ih1      second      gustav      0 R bg
...
13659.ih1      second      gustav      00:00:26 R bg
...
13659.ih1      second      gustav      00:00:46 R bg
...
13675.ih1      third       gustav      0 R bg
...
^C
[gustav@ih1 PBS]$ ls
Makefile  first_err  fourth_err  nodes.sh  second_out  third_out
bc.sh     first_out  fourth_out  process.sh  simple.sh  xterm.sh

```



```

first.sh fourth.sh job.sh      second.sh  third.sh
[gustav@ih1 PBS]$ cat first_out
/N/gpfs/gustav prepared and cleaned.
13659.ih1.avidd.iu.edu
second.sh submitted.
[gustav@ih1 PBS]$ cat second_out
writing on test
writing 1000 blocks of 1048576 random integers

real    5m8.000s
user    0m39.280s
sys     0m17.240s
-rw-r--r--  1 gustav  ucs      4194304000 Sep 13 13:28 test
File /N/gpfs/gustav/test generated.
13675.ih1.avidd.iu.edu
third.sh submitted.
[gustav@ih1 PBS]$ cat third_out
reading test
reading in chunks of size 16777216 bytes
allocated 16777216 bytes to junk
read 4194304000 bytes

real    0m42.039s
user    0m0.020s
sys     0m10.730s
File /N/gpfs/gustav/test processed.
13678.ih1.avidd.iu.edu
fourth.sh submitted.
[gustav@ih1 PBS]$ cat fourth_out
Directory /N/gpfs/gustav cleaned.
[gustav@ih1 PBS]$

```

Observe that IO is better on the computational nodes than on the head node. The reading program `xrandfile`, which has very little computation (the user CPU time is only 0.02s), returns transfer rate of 95 MB/s (remember that  $1 \text{ MB} = 2^{20} \text{ B} = 1048576 \text{ B}$ ).

Why do things this way?

If you have a very long job that can be divided into multiple separate tasks that can execute separately it is usually a good idea to do so. In case something goes wrong and the system crashes, or has to be taken down for maintenance, you won't lose the whole lot. In fact, the jobs may simply run without any problems at all, and the maintenance schedule will simply slide in between. Furthermore, if the system is configured so that there are restrictions on the wall clock time or CPU time consumed by PBS jobs (wall clock time restrictions make more sense in this context than CPU time restrictions – can you figure out why?), you may not be able to fit everything into a single job.

### A Multi-Job Job

Organizing your jobs into multiple batch files may lead to confusion, especially if files are many and they depend on each other in funny ways. It is better to compile all such tasks on a single file and submit it several times in order to execute various parts of the task. But how can we differentiate between various

submissions of the same file? The answer is *with environmental variables*. To begin with the PBS directive

```
#PBS -V
```

will make all variables defined in the environment from which the job is submitted available to the job. We can then add more variables by using the `-v` option followed by variable specifications.

In the following script I have combined `first.sh`, `second.sh`, `third.sh` and `fourth.sh` into a single script. Here is the listing:

```
[gustav@ih1 PBS]$ cat all.sh
#PBS -S /bin/bash
#PBS -q bg
#PBS -V
LOG=/N/B/gustav/PBS/all_log
case $STAGE in
1 )
  [ -d /N/gpfs/gustav ] || mkdir /N/gpfs/gustav
  cd /N/gpfs/gustav
  rm -f test
  echo "/N/gpfs/gustav prepared and cleaned."
  ssh ih1 "cd PBS; /usr/pbs/bin/qsub -v STAGE=2 all.sh"
  echo "Stage 2 submitted."
  ;;
2 )
  cd /N/gpfs/gustav
  time mkrandfile -f test -l 1000
  ls -l test
  echo "File /N/gpfs/gustav/test generated."
  ssh ih1 "cd PBS; /usr/pbs/bin/qsub -v STAGE=3 all.sh"
  echo "Stage 3 submitted."
  ;;
3 )
  cd /N/gpfs/gustav
  time xrandfile -f test -l 4
  echo "File /N/gpfs/gustav/test processed."
  ssh ih1 "cd PBS; /usr/pbs/bin/qsub -v STAGE=4 all.sh"
  echo "Stage 4 submitted."
  ;;
4 )
  cd /N/gpfs/gustav
  rm -f test
  echo "Directory /N/gpfs/gustav cleaned."
  ;;
esac >> $LOG 2>&1
exit 0
[gustav@ih1 PBS]$
```

The action taken by the script depends on the value of `STAGE`. If `$STAGE` is 1 then we prepare `/N/gpfs/gustav` and then submit *the same script*, but this time we set `STAGE` to 2 with the

```
-v STAGE=2
```

option.

In `STAGE 2` we generate the data file with `mkrandfile` and resubmit the same script with `-v STAGE=3`.

In STAGE 3 we process the data file with `xrandfile` and resubmit the script with `-v STAGE=4`.

Finally in STAGE 4 we clean `/N/gpfs/gustav` and exit.

Observe that all output is collected on one file `/N/B/gustav/PBS/all_log` to which each *instantiation* of the script *appends* its output. Also observe that we try to capture standard error on this file too. This is what the redirection

```
2>&1
```

means.

Because we take care of collecting all standard output and standard error, the output and error files generated by PBS will be empty.

The job is submitted as follows:

```
[gustav@ih1 PBS]$ qsub -v STAGE=1 all.sh
14204.ih1.avidd.iu.edu
[gustav@ih1 PBS]$
```

We are then going to see `all.sh` in the `qstat` listing but with a changing ID number.

```
[gustav@ih1 PBS]$ while sleep 10
> do
>   qstat | grep gustav
> done
14205.ih1      all.sh      gustav      0 R bg
...
14205.ih1      all.sh      gustav      00:00:13 R bg
...
14205.ih1      all.sh      gustav      00:00:34 R bg
...
14205.ih1      all.sh      gustav      00:00:57 R bg
...
14208.ih1      all.sh      gustav      0 R bg
...
```

Eventually the execution of all four stages completes and we get the following listing on `all_log`:

```
[gustav@ih1 PBS]$ cat all_log
/N/gpfs/gustav prepared and cleaned.
14205.ih1.avidd.iu.edu
Stage 2 submitted.
writing on test
writing 1000 blocks of 1048576 random integers

real    5m36.706s
user    0m38.510s
sys     0m18.780s
-rw-r--r--  1 gustav  ucs      4194304000 Sep 13 16:35 test
File /N/gpfs/gustav/test generated.
14208.ih1.avidd.iu.edu
Stage 3 submitted.
reading test
reading in chunks of size 16777216 bytes
allocated 16777216 bytes to junk
read 4194304000 bytes
```

```

real    0m33.902s
user    0m0.010s
sys     0m12.210s
File /N/gpfs/gustav/test processed.
14209.ih1.avidd.iu.edu
Stage 4 submitted.
Directory /N/gpfs/gustav cleaned.
[gustav@ih1 PBS]$

```

### PBS Dependency Lists

In this section we are going to do what we have done in section 4.3.4, but we will use PBS facility for defining job dependencies instead. We will have four scripts as before, but the scripts will be simpler and they will not submit other scripts. Instead we are going to tell PBS how our jobs depend on other jobs, so that PBS will wait for the first job to finish before it will release the second job. Then PBS will wait for the second job to finish, before the third job gets released, and so on. All jobs will be submitted at the same time from a single shell script.

The four jobs, `first_1.sh`, `second_1.sh`, `third_1.sh` and `fourth_1.sh` look the same as the jobs in section 4.3.4, `first.sh`, `second.sh`, `verb—third.sh`—and `fourth.sh`, with the exception that the job submission lines were commented out. The real trickery is in the shell script that does the submissions. Here is the script:

```

[gustav@bh1 PBS]$ cat submit_1
#!/bin/bash
FIRST='qsub first_1.sh'
echo $FIRST
SECOND='qsub -W depend=afterok:$FIRST second_1.sh'
echo $SECOND
THIRD='qsub -W depend=afterok:$SECOND third_1.sh'
echo $THIRD
FOURTH='qsub -W depend=afterok:$THIRD fourth_1.sh'
echo $FOURTH
exit 0
[gustav@bh1 PBS]$

```

Command `qsub` returns the job ID and this is normally printed on standard output. Here we capture the output of `qsub` in variables `FIRST`, `SECOND`, `THIRD` and `FOURTH`. The second job is submitted with option

```
-W depend=afterok:$FIRST
```

This means that the job itself is going to be put on hold until the first job has completed *with no errors*. Only then the second job is going to be released. The third and fourth jobs are treated similarly.

Let us run the script and see what happens:

```

[gustav@bh1 PBS]$ ./submit_1
13876.bh1.avidd.iu.edu
13877.bh1.avidd.iu.edu
13878.bh1.avidd.iu.edu
13879.bh1.avidd.iu.edu
[gustav@bh1 PBS]$ qstat | grep gustav

```

```

13876.bh1      first      gustav      0 Q bg
13877.bh1      second    gustav      0 H bg
13878.bh1      third     gustav      0 H bg
13879.bh1      fourth    gustav      0 H bg
[gustav@bh1 PBS]$ qstat -f 13878.bh1
Job Id: 13878.bh1.avidd.iu.edu
Job_Name = third
Job_Owner = gustav@bh1.avidd.iu.edu
job_state = H
queue = bg
server = bh1.avidd.iu.edu
Checkpoint = u
ctime = Sat Sep 13 14:33:26 2003
depend = afterok:13877.bh1.avidd.iu.edu@bh1.avidd.iu.edu,
        beforeok:13879.bh1.avidd.iu.edu@bh1.avidd.iu.edu
Error_Path = bh1.avidd.iu.edu:/N/B/gustav/PBS/third_err
Hold_Types = s
Join_Path = oe
Keep_Files = n
Mail_Points = a
mtime = Sat Sep 13 14:33:26 2003
Output_Path = bh1.avidd.iu.edu:/N/B/gustav/PBS/third_out
Priority = 0
qtime = Sat Sep 13 14:33:26 2003
Rerunable = True
Resource_List.ncpus = 1
Resource_List.nodect = 1
Resource_List.nodes = 1
Resource_List.walltime = 00:30:00
Shell_Path_List = /bin/bash
Variable_List = PBS_O_HOME=/N/B/gustav,PBS_O_LOGNAME=gustav,
                PBS_O_PATH=/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/usr/X11R6/b
                in:/usr/local/gm/bin:/usr/lpp/mmfs/bin:/opt/intel/compiler70/ia32/bin:/
                usr/local/maui/bin:/usr/pbs/bin:/usr/pbs/sbin:/opt/pgi/linux86/bin:/N/h
                pc/totalview/bin:/opt/xcat/bin:/opt/xcat/sbin:/opt/xcat/i686/bin:/opt/x
                cat/i686/sbin:/N/B/gustav/bin,PBS_O_MAIL=/var/spool/mail/gustav,
                PBS_O_SHELL=/bin/bash,PBS_O_HOST=bh1.avidd.iu.edu,
                PBS_O_WORKDIR=/N/B/gustav/PBS,PBS_O_QUEUE=bg

[gustav@bh1 PBS]$

```

We have generated four jobs, which were all submitted at roughly the same time. But only the first job is queued, whereas the remaining three jobs are on hold. Requesting the full listing of the third job with `qstat -f` shows the dependency:

```

depend = afterok:13877.bh1.avidd.iu.edu@bh1.avidd.iu.edu,
        beforeok:13879.bh1.avidd.iu.edu@bh1.avidd.iu.edu

```

The job can be started only *after* 13877 has completed without errors. Observe that PBS has recognized another dependency, which I have not specified explicitly. Namely that after this job, 13878, has completed without errors, then job 13879 should be started, i.e., that there is another job that depends on this one.

The dependency is specified by using the `-W` option to `qsub`. The option is generally used for *additional attributes*, of which *dependency* is one. The word `depend` that flags this attribute must be followed by a *list* of jobs on which the submitted job depends qualified with types of dependencies, e.g.,

```
-W depend=afterok:13876.bh1.avidd.iu.edu:13877.bh1.avidd.iu.edu
```

Here we state that the job can be released from hold *only* after *two* preceding jobs, 13876.bh1.avidd.iu.edu and 13877.bh1.avidd.iu.edu, have completed their run without errors.

The jobs get released one after another. This can be seen by running `qstat` every now and then:

```
[gustav@bh1 PBS]$ qstat | grep gustav
13878.bh1      third          gustav          00:00:05 R bg
13879.bh1      fourth         gustav          0 H bg
[gustav@bh1 PBS]$
```

Eventually everything completes and we are left with four logs in the PBS directory:

```
[gustav@bh1 PBS]$ cat *_out
/N/gpfs/gustav prepared and cleaned.
Directory /N/gpfs/gustav cleaned.
writing on test
writing 1000 blocks of 1048576 random integers

real    2m42.813s
user    0m40.040s
sys     0m14.760s
-rw-r--r--  1 gustav  ucs      4194304000 Sep 13 14:44 test
File /N/gpfs/gustav/test generated.
reading test
reading in chunks of size 16777216 bytes
allocated 16777216 bytes to junk
read 4194304000 bytes

real    2m51.521s
user    0m0.000s
sys     0m12.600s
File /N/gpfs/gustav/test processed.
[gustav@bh1 PBS]$
```

Observe that the IO on writing is 32 MB/s and only 23 MB/s on reading. This illustrates yet again how much IO can vary depending on the system load and configuration.

## 4.4 Checkpointing and Resubmission

In this section we will discuss how to time, checkpoint and automatically resubmit PBS jobs.

PBS can checkpoint under Cray Unicos automatically, but not under Linux. But checkpointing is a very intricate matter anyway, and if the state of the program depends on external files as well as on its memory image, then automatic checkpointing may not capture the full state of the program correctly. Additional complications arise when you attempt to checkpoint a parallel program. Consequently, it is usually a good idea to equip your program in its own checkpointing ability. This is not difficult and should be done for every serious project – especially if *you* are the developer.

The procedures discussed in this section are not limited to PBS. They should work for any batch submission system, as long as the batch jobs are described in terms of shell scripts, and as long as the system in question is IEEE-1003 (POSIX) compliant. They are applicable both to sequential and parallel jobs.

There are four issues that need to be addressed when automatically checkpointing and resubmitting your PBS jobs.

**Timing the job** Your job must know how much CPU or wall-clock time it used so far, and how much time there is still left.

**Saving the state of the job** This usually involves dumping a data file which contains an essential summary of the state of the system that is being computed. That file will be read when the job is restarted, and computation will commence from the point reached when the file has been dumped.

**Informing the parent process (usually a shell) that the computation should be continued**

This can be done, for example, by exiting the job with a non-zero exit status. Alternatively you could write a specific message on a log file (to be searched for by the shell script when the job exits) or create an empty flag file.

**Resubmitting the job** Depending on whether the job should be continued, the PBS script, before exiting, should either

1. resubmit itself, possibly with certain new flags or variables set up, or
2. clean up and inform the user that the computation has been completed.

#### 4.4.1 Timing a Job

Probably most C-language programmers know how to time their jobs, because functions `time` and `clock` are parts of the standard C library, which is defined by ANSI C specifications.

Function `time` takes a pointer to `time_t` as an argument and returns a value of `time_t` on exit.

On the AVIDD system `time_t` is defined in a somewhat convoluted manner. First it is defined as equivalent to `__time_t` on `/usr/include/time.h` and then `__time_t` is defined as `long int` on `/usr/include/bits/types.h`. But, as I have already remarked in section 4.3.4, `long int` is only a 32-bit integer on the AVIDD's IA32 nodes – as specified on `/usr/include/limits.h` in the `#if __WORDSIZE == 64` clause. Consequently, `time_t` can count up to `LONG_MAX`, which on the 32-bit architecture is 2147483647.

If the pointer passed to function `time` is not `NULL`, the return value is also placed in whatever location the pointer points at. The returned value is the current calendar time, in seconds, since the Epoch, i.e., 00:00:00 GMT, 1st of

January 1970: popularly celebrated as the day when UNIX was born. Because the maximum value returned is `LONG_MAX`, this gives us about

$$\frac{2147483647 \text{ s}}{365 \text{ days/year} \times 24 \text{ hours/day} \times 3600 \text{ seconds/hour}} \approx 68 \text{ years}$$

until the clock on 32-bit systems is going to wrap around, which is not until 2038.

If you think there won't be any 32-bit systems by that time and that they won't rely on the old fashioned UNIX function `time`, don't be so sure. We've seen in the year 2000 how much bad code and antiquated computer architectures lingered in corporate computer rooms and I am quite certain that we are going to have similar problems in 2038.

You would use function `time` in order to find out about the elapsed wall-clock time. If you know that, say, your queue allows only up to two wall-clock hours (7200 seconds) per job, by checking how much wall-clock time you've used so far, you will know how much time there is still left too.

Function `clock` does not take any arguments and returns a value of type `clock_t`, which on the AVIDD IA32 nodes is defined to be equivalent to `__clock_t` on `/usr/include/time.h` and then `__clock_t` is defined as `long int` on `/usr/include/bits/types.h`. So `clock_t` can also count up to  $2^{31} - 1 = 2147483647$  only.

This function returns CPU time that elapsed since the execution of the program commenced. The returned time is not in seconds. It is in clock cycles. There is a constant `CLOCKS_PER_SEC` defined on `/usr/include/time.h`, which tells how many clock cycles there are per second. So, in order to find out how many CPU seconds you have used so far, you have to divide the result obtained by calling `clock` by `CLOCKS_PER_SEC`. This is what section 3 of Linux manual will tell you.

But there is a slight complication here. Because of the insanity of the CAE XSH convention `CLOCKS_PER_SEC` is forcibly set to 1 million on *all* XSI-conformant systems, regardless of how fast their real clock is. So `CLOCKS_PER_SEC` is useless for us. On the other hand, since all other standard unix facilities use `CLOCKS_PER_SEC` too, as you will see soon enough, including PBS, we can just as well stick to it. After all we want to use it to protect our job against a possible PBS CPU-time-out.

Function `clock` wraps around very often. Present day chips are clocked at very high speeds, e.g., 3 GHz, and at these speeds, the output is going to wrap around every 0.7 seconds or so. Luckily, although I am going to show you how to use `clock` in the example below, PBS queues are seldom configured with respect to CPU time. It is usually the wall-clock time that really matters and not the CPU time, since the whole idea of a batch queue system is to protect the system against hogging<sup>1</sup>. You can easily imagine a job that is IO-bound and uses very little CPU time. Such a job could hog the resources for a very long wall-clock time, if the queue was CPU-time bound only.

The following example illustrates how to use functions `time` and `clock`.

<sup>1</sup>In our context to hog means "to take in excess of one's due", as you can check in the on-line Webster dictionary.



```

/*
 * Perform 10,000,000 square roots and divisions, sleep for 5 seconds.
 * Used to exercise time and clock functions.
 *
 * %Id: c-time.c,v 1.3 2003/09/18 21:57:08 gustav Exp %
 *
 * %Log: c-time.c,v %
 * Revision 1.3 2003/09/18 21:57:08 gustav
 * *** empty log message ***
 *
 * Revision 1.2 2003/09/18 21:55:03 gustav
 *
 */

#include <time.h> /* functions time and clock defined here */
#include <stdio.h> /* function printf defined here */
#include <unistd.h> /* function sleep defined here */
#include <stdlib.h> /* function exit defined here */
#include <math.h> /* function sqrt defined here */

main()
{
    time_t t0, t1; /* time_t is defined on <bits/types.h> as long */
    clock_t c0, c1; /* clock_t is defined on <bits/types.h> as long */

    long count;
    double a, b, c;

    printf ("using UNIX function time to measure wallclock time ... \n");
    printf ("using UNIX function clock to measure CPU time ... \n");

    t0 = time(NULL);
    c0 = clock();

    printf ("\tbegin (wall):          %ld seconds\n", (long) t0);
    printf ("\tbegin (CPU):           %ld cycles\n", (long) c0);

    printf ("\t\tsleep for 5 seconds ... \n");
    sleep(5);

    printf ("\t\tperform some computation ... \n");
    for (count = 1; count < 10000000; count++) {
        a = sqrt((double)count); /* square root is a very slow operation */
        b = 1.0/a; /* division is also a rather slow operation */
        c = b - a; /* subtraction takes very little time */
    }

    t1 = time(NULL);
    c1 = clock();

    printf ("\tend (wall):          %ld seconds\n", (long) t1);
    printf ("\tend (CPU);           %ld cycles\n", (long) c1);
    printf ("\telapsed wall clock time: %ld seconds\n", (long) (t1 - t0));
    printf ("\telapsed CPU time:      %f seconds\n", (float) (c1 - c0)/CLOCKS_PER_SEC);

    exit(0);
}

```

We compile this program with `Makefile` copied from our earlier programs. The only thing you need to change is to add

```
-lm
```

at the end of the command line that links the program in order to load the mathematics library, which contains the code of function `sqrt`.

```
[gustav@bh1 c-time]$ make
co RCS/Makefile,v Makefile
RCS/Makefile,v --> Makefile
revision 1.1
done
co RCS/c-time.c,v c-time.c
RCS/c-time.c,v --> c-time.c
revision 1.3
done
cc -c c-time.c
cc -o c-time c-time.o -lm
[gustav@bh1 c-time]$
```

And now we can run this program as follows

```
[gustav@bh1 c-time]$ time ./c-time
using UNIX function time to measure wallclock time ...
using UNIX function clock to measure CPU time ...
begin (wall):          1063922790 seconds
begin (CPU):           0 cycles
    sleep for 5 seconds ...
    perform some computation ...
end (wall):            1063922796 seconds
end (CPU);             1080000 cycles
elapsed wall clock time: 6 seconds
elapsed CPU time:      1.080000 seconds

real    0m6.204s
user    0m1.080s
sys     0m0.000s
[gustav@bh1 c-time]$
```

Observe that the CPU time calculated by our program agrees perfectly with the CPU time returned by the Linux command `time`. Our estimate of the wall clock time is off by 0.2 s, because our quantum of time is one second.

It is possible to measure elapsed time more accurately than down to a second. The UNIX program `time` obviously does. There is a Fortran function `etime`, which returns elapsed time with the resolution to a nanosecond, but this function does not have a simple, portable C-language equivalent. For the purpose of timing a program running for some hours under PBS the one-second resolution is good enough. For the purpose of timing loops inside the program, of course, it isn't, but then you are interested in the CPU time, not in the wall-clock time and you should use function `clock`, which yields the best resolution possible, i.e., down to a single clock cycle.

### Exercises

- 1 Use function `time` to wall-clock time writes and reads in programs `mkrandfile` (section 3.4.3) and `xrandfile` (section 4.3.4). You must ensure that individual writes and reads are sufficiently large for this to work, because function `time` returns time with the accuracy of one second, which is a very long time in computing. But this may just about work, if individual writes and reads are huge and take long time.
- 2 Implement an option that can be used to turn timing on. Design the program's logic so that timing is turned off by default.

#### 4.4.2 Restoring and Saving the State of a Job

In this section we shall discuss how to save and then restore the state of the computation between successive invocations of a program via PBS. The basic idea is that the only way any information can be transferred between successive invocations of a program is either

1. through a file, or
2. through an environmental variable, or
3. through command line switches

Transferring data through a file is perhaps the most common practice. Using files you can transfer very large amounts of data: e.g., the whole state of a 3D flow, or the whole state of a protein, or the whole state of a car in a crash simulation. Basically, files can be used to transfer any information from one instantiation of a program to another, including even small items of information, such as whether the program should restart a computation from a previously reached state, or whether it should start a new computation.

Instead of writing on files, the program, in principle, can also write on user's environment. On the next invocation the program can check for existence and state of certain predefined environmental variables, and obtain required information that way. This method is good for transferring small amounts of information, e.g., the name of a checkpoint file, or the request to initialize a run, but not for very large data sets.

Of course, using environmental variables will not work if the variables themselves are not transmitted from one PBS process to another one. And because we cannot submit jobs directly from computational nodes on the AVIDD cluster and have to resort to the `ssh` kludge instead, transmitting the whole environment is going to be rather difficult. In section 4.3.4 we have used the `-v` option to `qsub` to transmit and set just one environmental variable and this did the trick there. We are going to resort to a similar technique here.

We are going to write the information about the name of the checkpoint file and whether the job should be continued at the end of a log file. After our application exits, the PBS script responsible for the execution of the application

can inspect the log, and if it finds the instruction that the job should be continued, it can resubmit itself with appropriate values of certain environmental variables passed to its next instantiation by the means of the `-v` options. When the script is reincarnated by PBS, our application will begin by checking for the presence of these variables in the environment and for their content. From there it will learn if it should continue or reinitialize the computation, and if it should continue, where it should look for information about the state reached by the previous run.

The following listing shows a very simple C-language program which, if requested, reads the state of a computation from a file. If not requested it initializes a new computation. Then some further computation is performed and the new state is again saved on a file.

```

/*
 * Obtain information about the status of the computation and the name of the
 * checkpoint file (if present) from the environment. Open the checkpoint
 * file if the job is to be continued and read the state of the computation
 * from it.
 *
 * Sleep for 5 seconds (to make the computation look more substantial)
 * and perform the computation itself.
 *
 * If checkpointing has been requested then if the job has been restarted,
 * rename the old checkpoint file, then write the state of the computation
 * on a new checkpoint file.
 *
 * %Id: rsave.c,v 1.1 2003/09/19 19:24:08 gustav Exp %
 * %Log: rsave.c,v %
 * Revision 1.1 2003/09/19 19:24:08 gustav
 * Initial revision
 *
 */

#include <stdio.h> /* has definitions of printf, fprintf, fopen, fscanf,
                  fclose, fflush, perror, rename and BUFSIZ */
#include <stdlib.h> /* has definitions of getenv and exit */
#include <unistd.h> /* has definition of sleep */
#include <string.h> /* has definitions of strcpy and strcat */

main()
{
    char *restart_name, *restart, old_restart_name[BUFSIZ];
    FILE *restart_file;
    int n;

    /* Is this a continued job or a new one? */

    if (! (restart = getenv ("RSAVE_RESTART"))) {
        printf ("Starting a new run.\n");
        n = 0;
    }
    else {
        if (! (restart_name = getenv ("RSAVE_CHECKFILE"))) {
            fprintf (stderr, "error: no checkpoint file for the restart job\n");

```

```

        exit (1);
    }
    else {
        printf ("Restarting the job from %s.\n", restart_name);
        if (! (restart_file = fopen(restart_name, "r"))) {
            perror (restart_name);
            exit (2);
        }
        else {
            if (! (fscanf (restart_file, "%d", &n) > 0)) {
                fprintf (stderr, "%s: input file format error\n", restart_name);
                exit (3);
            }
            else {
                fclose (restart_file);
            }
        }
    }
}

printf ("n = %d\n", n);
printf ("\tcomputing ... "); fflush (stdout);
sleep (5);
n++;
printf ("done.\n");
printf ("n = %d\n", n);

if (! (restart_name = getenv ("RSAVE_CHECKFILE"))) {
    printf ("checkpointing not requested, exiting...\n");
    exit (0);
}
else {
    if (restart) {
        strcpy (old_restart_name, restart_name);
        strcat (old_restart_name, ".old");
        printf ("renaming old restart file to %s\n", old_restart_name);
        if (0 > rename (restart_name, old_restart_name)) {
            perror (old_restart_name);
            exit (4);
        }
    }
    printf ("saving data on %s\n", restart_name);
    if (! (restart_file = fopen (restart_name, "w"))) {
        perror (restart_name);
        exit (5);
    }
    else {
        fprintf (restart_file, "%d\n", n);
        fclose (restart_file);
    }
}
exit (0);
}

```

I'll explain how this program works in detail below, but first let's just see what it does:

```
[gustav@bh1 rsave]$ make
```

```

co RCS/Makefile,v Makefile
RCS/Makefile,v --> Makefile
revision 1.1
done
co RCS/rsave.c,v rsave.c
RCS/rsave.c,v --> rsave.c
revision 1.1
done
cc -c rsave.c
cc -o rsave rsave.o
[gustav@bh1 rsave]$ env | grep RSAVE
[gustav@bh1 rsave]$ export RSAVE_CHECKFILE=rsave.dat
[gustav@bh1 rsave]$ ./rsave
Starting a new run.
n = 0
      computing ... done.
n = 1
saving data on rsave.dat
[gustav@bh1 rsave]$ export RSAVE_RESTART=yes
[gustav@bh1 rsave]$ ./rsave
Restarting the job from rsave.dat.
n = 1
      computing ... done.
n = 2
renaming old restart file to rsave.dat.old
saving data on rsave.dat
[gustav@bh1 rsave]$ cat rsave.dat
2
[gustav@bh1 rsave]$ ./rsave
Restarting the job from rsave.dat.
n = 2
      computing ... done.
n = 3
renaming old restart file to rsave.dat.old
saving data on rsave.dat
[gustav@bh1 rsave]$ ./rsave
Restarting the job from rsave.dat.
n = 3
      computing ... done.
n = 4
renaming old restart file to rsave.dat.old
saving data on rsave.dat
[gustav@bh1 rsave]$

```

Here is the promised explanation of the program in detail.

The first thing that the program does, is to check for the existence of the environmental variable `RSAVE_RESTART`. If the variable does not exist, the program starts a new run and initialises `n` to 0.

If the variable `RSAVE_RESTART` exists (it doesn't really matter what is its value) then we first check if another variable, which should specify the name of the checkpoint file, `RSAVE_CHECKFILE`, exists too. If it doesn't then we have no way to find the name of the checkpoint file. So in that case we print an error message, flag an error on exit (value 1) and exit.

If the variable `RSAVE_CHECKFILE` exists then we use its value as the name of the checkpoint file, print a message about restarting the job from that file and

attempt to open it for reading.

If for some reason the file cannot be opened, we print the diagnostic on standard output (with `perror`), flag an error (value 2) and exit.

If the file has been opened without problems we try to read an integer number from it. That integer is the whole object of our simple computation in this program and it represents the state of the computation.

It may happen that for some reason the checkpoint file does not contain that integer. In that case we print the corresponding error message, flag an error (value 3) and exit.

But if everything goes well, by this time we should have our state of the system in hand, so we close the checkpoint file (in case of an error exit the file would be closed automatically) and commence the computation.

The computation is quite trivial. We simply increment the integer read from the file by 1. In order to add a little more body to the program we also sleep for 5 seconds (this is called putting on weight). We will need that sleep in our next example, which will combine timing with saving and restoring.

Once the computation is finished we again check the environmental variable `RSAVE_CHECKFILE`. Observe that this variable has not been looked up so far by the branch of the program, that does the initialisation. That is why we do it here again, even though the other branch, which is responsible for the restarting of the job, would have looked it up already.

If the variable `RSAVE_CHECKFILE` is not defined, we write the message that “checkpointing has not been requested” and exit. No error condition is flagged this time.

If the variable `RSAVE_CHECKFILE` exists, and if the job is a restarted one, then we attempt to rename the original restart file to whatever its old name was with a suffix `.old` appended.

If for some reason that cannot be done, we print a diagnostic on standard error using `perror`, flag an error (value 4) and exit.

Otherwise, having renamed the old restart file, we attempt to open, this time for writing, a new file bearing the old name. If for some reason that cannot be done a diagnostic is printed on standard error with `perror`, an error exit is flagged (value 5) and the program aborts.

Otherwise, i.e., if all went well and we have the new restart file opened, we write the new value of `n` on it, close it, and exit with status 0.

This is really quite simple stuff. Whatever complexity there is in the presented example, it derives from my attempt to make the program robust. Regardless of whether variables `RSAVE_RESTART` and `RSAVE_CHECKFILE` exist, regardless of whether the data file itself exists, the program should always do something more or less sensible, write meaningful error messages if need be, and exit gracefully conveying a meaningful exit value to the shell. For seasoned C and C++ programmers all this is just bread and butter.

### 4.4.3 Restoring, Timing and Saving a Job: the Complete Application

In this section we shall combine job timing with job restoring and saving, and produce a complete application, which, in the next section, will be combined with a PBS script, so as to produce an automatically resubmitting job.

The program is a slight modification of our restore and save example. There are no really new elements here, which would require a broader explanation.

The additional logic that is laid out on top of the restore and save example is as follows.

We begin by checking for a new environmental variable, `RSAVE_TIME_LIMIT`. If that variable does not exist then we assume that time allowed for this job is unlimited and things work more or less as before. If the variable exists then we attempt to read its value assuming that it is going to be a number. If it is not a number we print an error message and exit. If it is a number then the number is assigned to variable `time_limit` and assumed to represent the number of wall-clock seconds allocated to this job.

As I have mentioned before, what really matters to other users and to the system administrators is how long they have to wait, in terms of wall-clock time, until your job gets out of the way. For this reason I use the wall-clock timer, i.e., function `time`. However, you can modify the programs easily to look up the CPU time instead.

Once the information about the time limit is obtained we proceed exactly as before, until we get to the part of the program that does the computation. Instead of just incrementing number `n` and sleeping for 5 seconds, we enter a loop.

If no timing has been requested the loop keeps incrementing `n` and sleeping, until `n` becomes greater than `LAST_N`. The latter is an arbitrary constant, which in our toy example represents something like a convergence criterion. Once the “convergence” has been reached, the `finished` flag is set to `TRUE` and the loop exits.

Things are more interesting if timing of the job has been requested (by setting the environmental variable `RSAVE_TIME_LIMIT` to some number of seconds). In that case we measure time taken by one iteration of the loop, and we check how much time there is still left after the iteration has finished. If there is still enough time to perform another iteration we continue, if not, the loop exits.

Because saving the data, cleaning up, and executing the PBS script may take additional time we have to include a `SAFETY_MARGIN` while calculating time that still remains. In this case we set `SAFETY_MARGIN` to 10 seconds, but if you have to save a very large data set, you should probably reserve a couple of minutes.

Flagging the resubmission is accomplished as follows.

Before exiting, we check if the whole job is finished, which it will be once our “convergence” criterion is satisfied. If the job is finished we write `FINISHED` on standard output. Otherwise we write `CONTINUE`.

If the standard output has been logged on a file, after the program exits, the PBS script can inspect the log, and resubmit itself if it finds the word `CONTINUE`



in the log.

Here is the program. The parameters `LAST_N` and `SAFETY_MARGIN` have been implemented as `cpp` constants. I could also read them from the environment, a command line, or from an input file, but that would clutter the example.

The program always executes the statements of the `do ... while` loop at least once, because the exit condition is tested at the end of the loop.

Observe that the variable `quit_time` is initialised to 11. That way, if timing is not requested, it remains always positive and the `while` test fires up only when the job is finished. Furthermore the variable `timing` is initialised to `TRUE`, and becomes `FALSE` only if there is no environmental variable `RSAVE_TIME_LIMIT`. The job is assumed to be unfinished on entry (the variable `finished` is initialised to `FALSE`) and becomes finished only when `n` becomes greater than `LAST_N`. This means that once `n` becomes greater than `LAST_N`, you can still submit the job and it will always increment `n` by 1 before exiting.

```

/*
 * rts: Restart Time and Save
 *
 * %Id: rts.c,v 1.1 2003/09/19 20:55:37 gustav Exp %
 *
 * %Log: rts.c,v %
 * Revision 1.1 2003/09/19 20:55:37 gustav
 * Initial revision
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <time.h>

#ifndef TRUE
# define TRUE 1
#endif
#ifndef FALSE
# define FALSE 0
#endif

#ifndef LAST_N
# define LAST_N 30
#endif

#ifndef SAFETY_MARGIN
# define SAFETY_MARGIN 10
#endif

main()
{
    char *restart_name, *restart, old_restart_name[BUFSIZ];
    FILE *restart_file;
    int n, finished = FALSE, timing = TRUE;
    time_t t0, t1, t2, loop_time, time_left, time_limit, quit_time = 11;

```

```

char *time_limit_string;

/* Check the clock at the beginning of the run */
t0 = time(NULL);

/* Check how much time we have for this job */
if (! (time_limit_string = getenv ("RSAVE_TIME_LIMIT"))) {
    printf ("Unlimited time for this job.\n");
    timing = FALSE;
}
else {
    if (! (0 < sscanf (time_limit_string, "%d", &time_limit))) {
        fprintf (stderr, "Error: bad format of RSAVE_TIME_LIMIT\n");
        exit (1);
    }
    else {
        printf ("Time for this job limited to %d seconds.\n", time_limit);
    }
}

/* Is this a continued job or a new one? */

if (! (restart = getenv ("RSAVE_RESTART"))) {
    printf ("Starting a new run.\n");
    n = 0;
}
else {
    if (! (restart_name = getenv ("RSAVE_CHECKFILE"))) {
        fprintf (stderr, "error: no checkpoint file for the restart job\n");
        exit (1);
    }
    else {
        printf ("Restarting the job from %s.\n", restart_name);
        if (! (restart_file = fopen(restart_name, "r"))) {
            perror (restart_name);
            exit (2);
        }
        else {
            if (! (fscanf (restart_file, "%d", &n) > 0)) {
                fprintf (stderr, "%s: input file format error\n", restart_name);
                exit (3);
            }
            else {
                fclose (restart_file);
            }
        }
    }
}

printf ("n = %d\n", n);
printf ("\tcomputing ... \n"); fflush (stdout);

/* Loop while keeping an eye on the clock */

do {
    if (timing) t1 = time(NULL);

```

```

sleep (5);
n++;

/* Check if the whole simulation has been finished:
   this is our ‘‘convergence’’ criterion.
*/
if (n > LAST_N) finished = TRUE;

/* Check if we still have enough time for the next loop.
*/
if (timing) {
    t2 = time(NULL);
    loop_time = t2 - t1;
    time_left = time_limit - (t2 - t0);
    quit_time = time_left - loop_time - SAFETY_MARGIN;
    printf ("\t\tn = %d, time left = %d seconds\n", n, time_left);
    if ((quit_time <= 0) && (! finished))
printf ("\t\tnRun out of time, exiting ... \n");
}
} while ((quit_time > 0) && (! finished));

printf ("\tdone.\n");
printf ("n = %d\n", n);

if (! (restart_name = getenv ("RSAVE_CHECKFILE"))) {
    printf ("checkpointing not requested, exiting...\n");
    exit (0);
}
else {
    if (restart) {
        strcpy (old_restart_name, restart_name);
        strcat (old_restart_name, ".old");
        printf ("renaming old restart file to %s\n", old_restart_name);
        if (0 > rename (restart_name, old_restart_name)) {
            perror (old_restart_name);
            exit (4);
        }
    }
    printf ("saving data on %s\n", restart_name);
    if (! (restart_file = fopen (restart_name, "w"))) {
        perror (restart_name);
        exit (5);
    }
    else {
        fprintf (restart_file, "%d\n", n);
        fclose (restart_file);
    }
    if (! finished)
        printf ("CONTINUE\n");
    else
        printf ("FINISHED\n");
}
exit (0);
}

```

Here is how this job is run. First I submit it with the environmental variable `RSAVE_RESTART` unset, which initialises the job. Then I set `RSAVE_RESTART` to

yes and resubmit the job, which restarts from where it left.

The job is allowed to run no longer than 30 seconds at a time. Given the safety margin of 10 seconds and a single iteration time of 5 seconds this should let our program do 4 iterations. But the `while` clause tests for `quit_time > 0` not for `quit_time >= 0`, so, in effect we end up with 3 iterations instead of 4.

While the computational task remains unfinished, program `rts` writes `CONTINUE` on standard output before it exits. But the last run, when `n` becomes 31, is flagged with the word `FINISHED`.

```
[gustav@bh1 rts]$ make
co RCS/Makefile,v Makefile
RCS/Makefile,v --> Makefile
revision 1.1
done
co RCS/rts.c,v rts.c
RCS/rts.c,v --> rts.c
revision 1.1
done
cc -c rts.c
cc -o rts rts.o
[gustav@bh1 rts]$ env | grep RSAVE
RSAVE_CHECKFILE=rts.dat
RSAVE_TIME_LIMIT=30
[gustav@bh1 rts]$ ./rts
Time for this job limited to 30 seconds.
Starting a new run.
n = 0
    computing ...
        n = 1, time left = 25 seconds
        n = 2, time left = 20 seconds
        n = 3, time left = 15 seconds
        Run out of time, exiting ...
    done.
n = 3
saving data on rts.dat
CONTINUE
[gustav@bh1 rts]$ export RSAVE_RESTART=yes
[gustav@bh1 rts]$ ./rts
Time for this job limited to 30 seconds.
Restarting the job from rts.dat.
n = 3
    computing ...
        n = 4, time left = 25 seconds
        n = 5, time left = 20 seconds
        n = 6, time left = 15 seconds
        Run out of time, exiting ...
    done.
n = 6
renaming old restart file to rts.dat.old
saving data on rts.dat
CONTINUE
[gustav@bh1 rts]$ ./rts
Time for this job limited to 30 seconds.
Restarting the job from rts.dat.
n = 6
    computing ...
```

```
        n = 7, time left = 25 seconds
        n = 8, time left = 20 seconds
        n = 9, time left = 15 seconds
        Run out of time, exiting ...
    done.
n = 9
renaming old restart file to rts.dat.old
saving data on rts.dat
CONTINUE
[gustav@bh1 rts]$

...

[gustav@bh1 rts]$ ./rts
Time for this job limited to 30 seconds.
Restarting the job from rts.dat.
n = 24
    computing ...
        n = 25, time left = 25 seconds
        n = 26, time left = 20 seconds
        n = 27, time left = 15 seconds
        Run out of time, exiting ...
    done.
n = 27
renaming old restart file to rts.dat.old
saving data on rts.dat
CONTINUE
[gustav@bh1 rts]$ ./rts
Time for this job limited to 30 seconds.
Restarting the job from rts.dat.
n = 27
    computing ...
        n = 28, time left = 25 seconds
        n = 29, time left = 20 seconds
        n = 30, time left = 15 seconds
        Run out of time, exiting ...
    done.
n = 30
renaming old restart file to rts.dat.old
saving data on rts.dat
CONTINUE
[gustav@bh1 rts]$ ./rts
Time for this job limited to 30 seconds.
Restarting the job from rts.dat.
n = 30
    computing ...
        n = 31, time left = 25 seconds
    done.
n = 31
renaming old restart file to rts.dat.old
saving data on rts.dat
FINISHED
[gustav@bh1 rts]$
```

#### 4.4.4 Combining the Application with PBS: Automatic Resubmission

In this section I shall demonstrate how our toy application can be run under the PBS, and how you can use its various features to automatically keep resubmitting the job until the whole computational task is finished.

The PBS script begins by changing to the GPFS directory where we run rts and then running program itself. The output is saved on rts.log:

```
cd /N/gpfs/gustav/rts
./rts > rts.log
```

After the job exits the script performs a number of manipulations. First of all, it checks if an environmental variable `RSAVE_STEP` exists. That variable is used to number our PBS runs. If the variable exists, it means that this particular run was already a resubmission. In that case the value of `RSAVE_STEP` is incremented and the old restart file, say, `rts.dat.old` is renamed to something like `rts.dat.3`, where 3 is the `RSAVE_STEP` number. This way we keep the log of the whole computation. In a more complex application the `rts.dat` files could contain images or three dimensional data sets that, if saved, could be used to produce an animation or a CAVE display.

If the variable `RSAVE_STEP` does not exist, it means that this is the initializing run. In that case the variable is created and assigned number 0. We will make it available to the next run by the means of the `qsub -v` option.

The log file, `rts.log` is also saved on something like, say, `rts.log.3`, where 3 is the `RSAVE_STEP` number. Observe that `rts.log.3` corresponds to the run that used `rts.dat.3` as its restart file.

After these manipulations we inspect the log file itself and check if it contains the word `CONTINUE`. If it does we resubmit the next iteration of the job with the command:

```
ssh bh1 "cd /N/B/gustav/PBS; /usr/pbs/bin/qsub \
-v RSAVE_TIME_LIMIT=30,RSAVE_CHECKFILE=rts.dat,RSAVE_RESTART=yes,\
RSAVE_STEP=$RSAVE_STEP rts.sh"
```

We pass `RSAVE_RESTART=yes` to the next job on the `qsub` command line, this way the next job will know it is a resubmission job, not an initialization job. We also pass the `RSAVE_STEP` to the next job, so that it can enumerate the log and data files properly.

If the word `CONTINUE` has not been found in the log file, then we check if the log file contains the word `FINISHED`. If the job is `FINISHED` it is not resubmitted. Instead a mail message is sent to me, in this case, that tells me about the completion of the computation. Observe that the mail message is sent from the head node, `bh1`, via `ssh`.

If neither the word `CONTINUE` nor the word `FINISHED` has been found in the log file, it means that an error condition must have occurred and the job exited mid-way. In that case, the job is not resubmitted and a mail message informing about the error is sent to me.

Here is the whole PBS script in full glory:

```

[gustav@bh1 PBS]$ cat rts.sh
#PBS -S /bin/bash
#PBS -q bg
#PBS -m a
#PBS -M gustav@indiana.edu
#PBS -V
#
cd /N/gpfs/gustav/rts
#
# Execute this step
#
rts > rts.log
#
# If there is $RSAVE_CHECKFILE.old file then
# replace the suffix ".old" with a step number
#
if [ -n "${RSAVE_STEP}" ]
then
  RSAVE_STEP='expr $RSAVE_STEP + 1'
  if [ -n "${RSAVE_CHECKFILE}" ]
  then
    if [ -f $RSAVE_CHECKFILE.old ]
    then
      mv $RSAVE_CHECKFILE.old $RSAVE_CHECKFILE.$RSAVE_STEP
    fi
  fi
else
  RSAVE_STEP=0
fi
#
# save the log of this run
#
cp rts.log rts.log.$RSAVE_STEP
#
# Check if the job is finished and if it is not
# resubmit this file
#
if grep CONTINUE rts.log
then
  ssh bh1 "cd /N/E/gustav/PBS; /usr/pbs/bin/qsub -v RSAVE_TIME_LIMIT=30,RSAVE_CHECKFILE=rts.dat,RSAVE_RESTART=yes,RSAVE
elif grep FINISHED rts.log
then
  ssh bh1 mail -s finished gustav@indiana.edu << EOF
Your job rts has FINISHED
EOF
else
  ssh bh1 mail -s error gustav@indiana.edu << EOF
rts: error exit, check the log file
EOF
fi
exit 0
[gustav@bh1 PBS]$

```

And here is how this script is submitted and what happens afterwards.

```

[gustav@bh1 PBS]$ env | grep RSAVE
RSAVE_TIME_LIMIT=30
RSAVE_CHECKFILE=rts.dat

```

```
[gustav@bh1 PBS]$ qsub rts.sh
17038.bh1.avidd.iu.edu
[gustav@bh1 PBS]$
```

Observe that only `RSAVE_TIME_LIMIT` and `RSAVE_CHECKFILE` have been defined. All other variables will be defined by the PBS script as they become needed.

The job runs happily resubmitting itself every time the program `rts` exits and producing numerous log and data files. In this case I have separated the PBS output, which is written on my job submission directory `/N/B/gustav/PBS` and the `rts` output, which is written on my GPFS directory `/N/gpfs/gustav/rts`. Eventually I get a message that looks as follows

```
Envelope-to: gustav@woodlands.tqc.iu.edu
Delivery-date: Fri, 19 Sep 2003 17:40:03 -0500
Date: Fri, 19 Sep 2003 17:39:26 -0500
From: Zdzislaw Meglicki <gustav@bh1.uits.indiana.edu>
To: gustav@indiana.edu
Subject: finished
```

Your job `rts` has FINISHED

and then I can go to my PBS and GPFS directories and look at the logs.

I find the following PBS output and error files in `/N/B/gustav/PBS`:

```
[gustav@bh1 PBS]$ pwd
/N/B/gustav/PBS
[gustav@bh1 PBS]$ ls rts.sh.*
rts.sh.e17040 rts.sh.e17045 rts.sh.e17050 rts.sh.o17044 rts.sh.o17049
rts.sh.e17041 rts.sh.e17046 rts.sh.o17040 rts.sh.o17045 rts.sh.o17050
rts.sh.e17042 rts.sh.e17047 rts.sh.o17041 rts.sh.o17046
rts.sh.e17043 rts.sh.e17048 rts.sh.o17042 rts.sh.o17047
rts.sh.e17044 rts.sh.e17049 rts.sh.o17043 rts.sh.o17048
[gustav@bh1 PBS]$
```

Files `rts.sh.o*` show the job resubmission history:

```
[gustav@bh1 PBS]$ cat rts.sh.o*
CONTINUE
17041.bh1.avidd.iu.edu
CONTINUE
17042.bh1.avidd.iu.edu
CONTINUE
17043.bh1.avidd.iu.edu
CONTINUE
17044.bh1.avidd.iu.edu
CONTINUE
17045.bh1.avidd.iu.edu
CONTINUE
17046.bh1.avidd.iu.edu
CONTINUE
17047.bh1.avidd.iu.edu
CONTINUE
17048.bh1.avidd.iu.edu
CONTINUE
17049.bh1.avidd.iu.edu
CONTINUE
```



```
17050.bh1.avidd.iu.edu
FINISHED
[gustav@bh1 PBS]$
```

and files `rts.sh.e*` are, mercifully, empty.

Switching now to `/N/gpfs/gustav/rts` shows:

```
[gustav@bh1 PBS]$ cd /N/gpfs/gustav/rts
[gustav@bh1 rts]$ ls
rts.dat    rts.dat.3  rts.dat.7  rts.log.0  rts.log.3  rts.log.7
rts.dat.1  rts.dat.4  rts.dat.8  rts.log.1  rts.log.4  rts.log.8
rts.dat.10 rts.dat.5  rts.dat.9  rts.log.10 rts.log.5  rts.log.9
rts.dat.2  rts.dat.6  rts.log    rts.log.2  rts.log.6
[gustav@bh1 rts]$
```

Files `rts.dat.*` contain the *animation* of the computation process:

```
[gustav@bh1 rts]$ cat 'ls -t rts.dat.*'
30
27
24
21
18
15
12
9
6
3
[gustav@bh1 rts]$
```

If these files contained images, I could make them into a movie. Files `rts.log.*` contain the detailed history of the whole computation:

```
[gustav@bh1 rts]$ cat 'ls -t rts.log.*'
Time for this job limited to 30 seconds.
Restarting the job from rts.dat.
n = 30
    computing ...
        n = 31, time left = 25 seconds
    done.
n = 31
renaming old restart file to rts.dat.old
saving data on rts.dat
FINISHED
Time for this job limited to 30 seconds.
Restarting the job from rts.dat.
n = 27
    computing ...
        n = 28, time left = 25 seconds
        n = 29, time left = 20 seconds
        n = 30, time left = 15 seconds
        Run out of time, exiting ...
    done.
n = 30
renaming old restart file to rts.dat.old
saving data on rts.dat
CONTINUE
...
Time for this job limited to 30 seconds.
```

```
Restarting the job from rts.dat.
n = 3
  computing ...
    n = 4, time left = 25 seconds
    n = 5, time left = 20 seconds
    n = 6, time left = 15 seconds
    Run out of time, exiting ...
  done.
n = 6
renaming old restart file to rts.dat.old
saving data on rts.dat
CONTINUE
Time for this job limited to 30 seconds.
Starting a new run.
n = 0
  computing ...
    n = 1, time left = 25 seconds
    n = 2, time left = 20 seconds
    n = 3, time left = 15 seconds
    Run out of time, exiting ...
  done.
n = 3
saving data on rts.dat
CONTINUE
[gustav@bh1 rts]$
```

I have requested in the PBS script that PBS should send mail to me only when the job gets aborted. This is because I don't want to be flooded by excessive amount of mail. The PBS directive `#PBS -m e` would send me e-mail every time the script exits, not when the whole computation is done.

# Chapter 5

## MPI and MPI-IO

### 5.1 Introduction

#### 5.1.1 The History of MPI

Various message passing environments were developed in early 80s. Some were developed for special purpose machines such as the Caltech's N-cube (our Geoffrey Fox was amongst the authors of that software), some were developed for networks of UNIX workstations, e.g., the Argonne's P4 library and PICL and PVM from Oak Ridge. The Ohio Supercomputer Center developed a message passing library called LAM. There was a package developed specially for quantum chemistry called TCGMSG and commercially available libraries like Express that derived from the N-cube system.

By early 1992 it was clear that the authors of these numerous libraries duplicated each other's efforts and busied themselves re-inventing the wheel all the time. In late 1992 a meeting was called during the Supercomputing 92 conference and the attendants agreed to develop and then implement a common standard for message passing that would incorporate all the interesting ideas developed so far and build on them. And this is how MPI, the Message Passing Interface, was born.

Some message passing libraries, like ISIS developed by Cornell University's Ken Birman, didn't fit the somewhat stiff model proposed by the participants of the conference, and so they decided to go it alone in a quite different direction of fault-tolerant distributed computing systems that helped them make heaps of money (ISIS was deployed at major stock exchange operations around the world) and ended up eventually at the door of Microsoft, which incorporated ISIS technology into its clustering product called Wolfpack.

ISIS was based on the insanely great idea of *virtual synchrony*, but this idea wouldn't play in context of scientific computing, where synchronizing processes, even if virtually only, would carry a heavy performance price. At the same time, MPI was biased unashamedly towards supercomputing and the issues of fault tolerance and synchronization were not considered critical.

There were many industrial participants in the MPI club that helped finance the endeavour. Amongst them were Convex, Cray, IBM, Intel, Meiko, nCUBE, NEC and Thinking Machines. Some of these vendors bit the dust, but many continued to prosper and benefited from the development of MPI. IBM, especially, is in the latter group. IBM SP is an MPI machine. Today, all PC and UNIX workstation clusters are MPI systems too and MPI programs are run on the Earth Simulator, Cray X1, and large SMPs.

The first MPI standard, called MPI-1 was completed in May 1994. The second MPI standard, MPI-2, was completed in 1998. There was so much enthusiasm back in 1994 that first MPI-1 implementations were released only about a year later, the most popular ones being the Argonne’s MPICH (based on the P4 package and Chameleon – hence the “CH” suffix) and Ohio LAM MPI. There were still many supercomputer vendors around back then too, and they released their own implementations of MPI.

But by 1998, the time MPI-2 standard was formalized, much of this enthusiasm evaporated and the first implementation of this more advanced standard had to wait until November 2002. So MPI-2 is still very new. Yet, it is of special interest to us, because it is *only* in MPI-2 that parallel IO operations were defined. These operations were invented and implemented originally in a package called MPI-IO that was developed for NASA prior to MPI-2 standardization. MPI designers liked it so much that they incorporated all of it into the new MPI-2 standard.

There is only one freeware MPI-IO version that floats about, and that I know of, and it is called ROMIO. It was developed by the same people who developed the original MPICH and their younger colleagues. ROMIO can be combined with MPI-1, as an external MPI library or incorporated directly into MPI-2. It works much the same in both cases.

### 5.1.2 MPI Documentation and Literature

The bible of MPI programmers is the on-line manual, which can be read in the HTML format. It is located at: <http://www-unix.mcs.anl.gov/mpi/>. You will find both “MPI Standard 1.1” and “MPI Standard 2.0” there. You can also find information about MPICH and MPICH2 there and download both packages.

There is a couple of books about MPI that are rather helpful too. First there is the introduction to MPI by the very authors of the library and its implementation, MPICH, “Using MPI – 2nd Edition: Portable Parallel Programming with the Message Passing Interface (Scientific and Engineering computation)” by William Gropp, Ewing Lusk, Anthony Skjellum and Rajeev Thakur. You can find this book on Amazon.com. It costs \$42. The next book is “Using MPI-2: Advanced Features of the Message Passing Interface (Scientific and Engineering Computation)” by William Gropp, Ewing Lusk and Rajeev Thakur. This book is the follow-up to the first book and costs \$42 too.

The on-line MPI documentation can be purchased in a book format from Amazon.com as well as a two volume set, “MPI: The Complete Reference” by

William Gropp, Marc Snir, Bill Nitzberg and Ewing Lusk, for \$68. Volume 2 covers MPI-IO.

You do not have to buy any of these texts for this course. Apart from various examples you'll see in these pages, you can also download MPI programming examples from Argonne. But if you intend to develop serious MPI applications, you should probably get all four books.

My suggestion is that you should hold on with any purchases until this course is finished. This will give you a better idea of what this whole business is about. You can buy the books then, if you think you will really need them.

### 5.1.3 What is in MPI?

A lot! MPI is a very comprehensive library and MPI-2 has about all of it including some elements of fault-tolerance, process migration, etc.

The most fundamental functions in MPI, `MPI_Send` and `MPI_Recv`, provide not only for buffered sends and receives, but also for typing of transferred data. They also support the concept of a *communicator*. A communicator is a group of processes within which the ranking, i.e., process differentiation, and communication take place. A single program may comprise several overlapping or completely separate communicators. This concept is crucial to implementations of parallel code libraries. Otherwise, there would always be a risk of library communications interfering with other communications within the parallel program.

MPI provides a very rich set of collective communication functions, virtual topologies, hooks for debugging and profiling, blocking and non-blocking sends and receives, and support for heterogeneous networks. MPI-2 adds numerous clarifications, specification of portable MPI process startup (previously every MPI implementation would use different ways of starting MPI programs), new data type manipulation functions and new predefined types, support for dynamic process creation and management, support for one-sided communications, i.e., for the ability to write data directly into other processes' memories, portable high-performance IO in the form of MPI-IO, and support for C++ and Fortran 90.

MPI is obviously a very large library. The listing below shows *all* MPI-1 functions. There are 128 of them. MPI-2 adds at least this many if not more.

MPI_ABORT	MPI_ADDRESS	MPI_ALLGATHER	MPI_ALLGATHERV
MPI_ALLREDUCE	MPI_ALLTOALL	MPI_ALLTOALLV	MPI_ATTR_DELETE
MPI_ATTR_GET	MPI_ATTR_PUT	MPI_BARRIER	MPI_BCAST
MPI_BSEND	MPI_BSEND_INIT	MPI_BUFFER_ATTACH	MPI_BUFFER_DETACH
MPI_CANCEL	MPI_CARTDIM_GET	MPI_CART_COORDS	MPI_CART_CREATE
MPI_CART_GET	MPI_CART_MAP	MPI_CART_RANK	MPI_CART_SHIFT
MPI_CART_SUB	MPI_COMM_COMPARE	MPI_COMM_CREATE	MPI_COMM_DUP
MPI_COMM_FREE	MPI_COMM_GROUP	MPI_COMM_RANK	MPI_COMM_REMOTE_GROUP
MPI_COMM_REMOTE_SIZE	MPI_COMM_SIZE	MPI_COMM_SPLIT	MPI_COMM_TEST_INTER
MPI_DIMS_CREATE	MPI_ERRHANDLER_CREATE	MPI_ERRHANDLER_FREE	MPI_ERRHANDLER_GET
MPI_ERRHANDLER_SET	MPI_ERROR_CLASS	MPI_ERROR_STRING	MPI_FINALIZE
MPI_GATHER	MPI_GATHERV	MPI_GET_COUNT	MPI_GET_ELEMENTS
MPI_GET_PROCESSOR_NAME	MPI_GRAPHDIMS_GET	MPI_GRAPH_CREATE	MPI_GRAPH_GET
MPI_GRAPH_MAP	MPI_GRAPH_NEIGHBORS	MPI_GRAPH_NEIGHBORS_COUNT	MPI_GROUP_COMPARE
MPI_GROUP_DIFFERENCE	MPI_GROUP_EXCL	MPI_GROUP_FREE	MPI_GROUP_INCL
MPI_GROUP_INTERSECTION	MPI_GROUP_RANGE_EXCL	MPI_GROUP_RANGE_INCL	MPI_GROUP_RANK
MPI_GROUP_SIZE	MPI_GROUP_TRANSLATE_RANKS	MPI_GROUP_UNION	MPI_IBSEND
MPI_INIT	MPI_INITIALIZED	MPI_INTERCOMM_CREATE	MPI_INTERCOMM_MERGE
MPI_IPROBE	MPI_IRECV	MPI_IRSEND	MPI_ISEND
MPI_ISSEND	MPI_KEYVAL_CREATE	MPI_KEYVAL_FREE	MPI_OP_CREATE
MPI_OP_FREE	MPI_PACK	MPI_PACK_SIZE	MPI_PCONTROL
MPI_PROBE	MPI_RECV	MPI_RECV_INIT	MPI_REDUCE
MPI_REDUCE_SCATTER	MPI_REQUEST_FREE	MPI_RSEND	MPI_RSEND_INIT
MPI_SCAN	MPI_SCATTER	MPI_SCATTERV	MPI_SEND
MPI_SENDREC	MPI_SENDREC_REPLACE	MPI_SEND_INIT	MPI_SSEND
MPI_SSEND_INIT	MPI_START	MPI_STARTALL	MPI_TEST
MPI_TESTALL	MPI_TESTANY	MPI_TESTSOME	MPI_TEST_CANCELLED
MPI_TOPO_TEST	MPI_TYPE_COMMIT	MPI_TYPE_CONTIGUOUS	MPI_TYPE_EXTENT
MPI_TYPE_FREE	MPI_TYPE_HINDEXED	MPI_TYPE_HVECTOR	MPI_TYPE_INDEXED
MPI_TYPE_LB	MPI_TYPE_SIZE	MPI_TYPE_STRUCT	MPI_TYPE_UB
MPI_TYPE_VECTOR	MPI_UNPACK	MPI_WAIT	MPI_WAITALL
MPI_WAITANY	MPI_WAITSSOME	MPI_WTICK	MPI_WTIME

But you don't have to panic. In great many cases you can do all you need with just 6 fundamental MPI functions:

**MPI\_Init** Initialize MPI processes;

**MPI\_Comm\_size** Find out about the number of the processes in the MPI communicator;

**MPI\_Comm\_rank** What is my rank number within the pool of MPI communicator processes?

**MPI\_Send** Send a message.

**MPI\_Recv** Receive a message.

**MPI\_Finalize** Close down MPI processes and prepare for exit.

All the other functions, 128 minus 6, are auxiliary. They make programmer's life much, much easier by encapsulating frequently used parallel programming procedures in convenience functions.

### 5.1.4 Programming Examples

You can download example MPI programs as well as MPI benchmarks from <http://www-unix.mcs.anl.gov/mpi/>. This page is implemented in the form of two frames. Once you have connected click on "MPICH Home Page" in the left frame. Then scroll down to "Demonstrations", and click on "Example programs". The independent address of the page you'll get to is <http://www.mcs.anl.gov/mpi/mpich/demo.h>. This page lets you download the parallel implementation of the Mandelbrot set calculation.

You will find examples discussed in the "Using MPI" books at:

- `ftp://ftp.mcs.anl.gov/pub/mpi/usingmpi-1st`,
- `ftp://ftp.mcs.anl.gov/pub/mpi/usingmpi2` and
- `ftp://ftp.mcs.anl.gov/pub/mpi/using/examples`.

And, of course, you will find annotated examples of MPI programs in these pages too.

### 5.1.5 Running MPI on the AVIDD Clusters

Running MPI on the AVIDD Clusters is a little problematic because there are several MPI versions and implementations there. What is of special interest to us is MPI-2, because MPI-IO is integral to it. For this reason we have installed a beta version of Argonne's MPICH2 in `/N/hpc/mpich2`. This directory should be mounted on all computational nodes and head nodes of both IUPUI and IUB clusters.

You will also need to define an environmental variable `MPD_USE_USER_CONSOLE` and, possibly, `LD_RUN_PATH` and `LD_LIBRARY_PATH` (because there are dynamic run-time libraries in `/N/hpc/mpich2/lib`). Without `MPD_USE_USER_CONSOLE` defined in your environment the MPICH2 execution engine is going to do some rather weird things with the name of the socket file with the sad effect that correct communication within the engine will not get established and the engine will shut down.

But first and foremost you must create a file `.mpd.conf` in your home directory. This file must be readable by you only and you must be the only person allowed to write on it too:

```
$ cd
$ touch .mpd.conf
$ chmod 600 .mpd.conf
```

Now you must enter the following line on this file:

```
password=yoopee
```

Replace the word “yoopee” with your favourite password, of course. Do not use your AVIDD or your IU Net password. This password is for the MPICH2 MPD system only.

I forgot to tell you about this in the laboratory class and that is why MPICH2 worked for me only. Then I forgot about it altogether, and ended up very perplexed and suspected MPICH2 of the most horrible bugs imaginable. MPICH2 is a beta release at this stage, to be sure, so surprises are possible, but it's not this bad.

The easiest way to get your `PATH`, `MANPATH`, `LD_LIBRARY_PATH`, `LD_RUN_PATH` and `MPD_USE_USER_CONSOLE` right is to copy `.bashrc`, `.bash_profile` and `.inputrc` files from my home directory to your home.

Proceed as follows. After you have logged on, issue the commands:

```

$ cd
$ cp .bashrc .bashrc.ORIG
$ cp .bash_profile .bash_profile.ORIG
$ cp .inputrc .inputrc.ORIG
$ cp ~gustav/.bashrc .bashrc
$ cp ~gustav/.bash_profile .bash_profile
$ cp ~gustav/.inputrc .inputrc
$ chmod 755 .bashrc .bash_profile
$ chmod 644 .inputrc

```

Having done this logout and login again.

If you know what you are doing and if you prefer to use shells other than `bash`, have a look at these files and then set up your environment similarly. The most important thing is to

1. Ensure that your `$HOME/bin` is in front of the command search path, so that you can overwrite system commands with your own.
2. Ensure that the MPICH2 directory `/N/hpc/mpich2/bin` is the second directory in your command search path, so that you'll get MPI-2 start up commands, as well as Python-2.3 and other tools used by MPI-2, in place of whatever may be currently installed on the system – the system-wide version of Python, for example, is older and doesn't work with MPICH2.

To check that everything is as it ought to be try the following commands:

```

gustav@bh1 $ cd
gustav@bh1 $ ls -l .mpd.conf
-rw-----  1 gustav  ucs           16 Oct  2 18:58 .mpd.conf
gustav@bh1 $ cat .mpd.conf
password=frabjous
gustav@bh1 $ env | grep PATH
LD_RUN_PATH=/N/B/gustav/lib:/N/hpc/mpich2/lib
LD_LIBRARY_PATH=/N/B/gustav/lib:/N/hpc/mpich2/lib
MANPATH=/N/B/gustav/man:/N/B/gustav/share/man:/N/hpc/mpich2/man:\
N/hpc/mpich2/share/man:/usr/local/man:/usr/local/share/man:\
usr/man:/usr/share/man:/usr/X11R6/man
PATH=/N/B/gustav/bin:/N/hpc/mpich2/bin:/usr/local/bin:/bin:\
usr/bin:/usr/X11R6/bin:/usr/pbs/bin:/usr/local/hpss:.
gustav@bh1 $ env | grep MPD
MPD_USE_USER_CONSOLE=yes
gustav@bh1 $

```

The first directory in `LD_RUN_PATH`, `LD_LIBRARY_PATH`, `MANPATH` and `PATH`, of course, should be replaced with your private `bin`, `lib` and `man`.

Now you should run the following command on the IUB cluster

```

gustav@bh1 $ for i in `cat ~gustav/.bcnodes`
> do
>   echo -n "$i: "
>   ssh $i date
> done
bc01-myri0: Wed Oct  1 16:12:56 EST 2003
bc02-myri0: Wed Oct  1 16:12:56 EST 2003
bc03-myri0: Wed Oct  1 16:12:56 EST 2003
bc04-myri0: Wed Oct  1 16:12:57 EST 2003

```



```
...
bc93-myri0: Wed Oct  1 16:13:39 EST 2003
bc94-myri0: Wed Oct  1 16:13:39 EST 2003
bc95-myri0: Wed Oct  1 16:13:39 EST 2003
bc96-myri0: Wed Oct  1 16:13:40 EST 2003
gustav@bh1 $
```

and similarly on the IUPUI cluster:

```
gustav@ih1 $ for i in `cat ~gustav/.icnodes`
> do
>   echo -n "$i: "
>   ssh $i date
> done
ic01-myri0: Wed Oct  1 16:15:14 EST 2003
ic02-myri0: Wed Oct  1 16:15:14 EST 2003
ic03-myri0: Wed Oct  1 16:15:15 EST 2003
ic04-myri0: Wed Oct  1 16:15:15 EST 2003
...
ic93-myri0: Wed Oct  1 16:18:24 EST 2003
ic94-myri0: Wed Oct  1 16:18:24 EST 2003
ic95-myri0: Wed Oct  1 16:18:24 EST 2003
ic97-myri0: Wed Oct  1 16:18:25 EST 2003
```

Please let me know if these commands hang on any of the nodes. The files `~gustav/.bcnodes` and `~gustav/.icnodes` contain the lists of currently functional computational nodes with working Myrinet interfaces both at IUPUI and IUB. These lists may change every now and then, in which case we may have to repeat this procedure.

The purpose of this procedure is to populate your `~/.ssh/known_hosts` file with the computational nodes' keys. The `ssh` command inserts the key automatically in your `known_hosts` if it is not there. But in the process it writes a message on standard output that may confuse the MPICH2 startup scripts.

Now you are almost ready to run your first MPI job. First copy two more files from my home directory. Do it as follows:

```
$ cd
$ [ -d bin ] || mkdir bin
$ [ -d PBS ] || mkdir PBS
$ cp ~gustav/bin/hellow2 bin
$ chmod 755 bin/hellow2
$ cp ~gustav/PBS/mpi.sh PBS
$ chmod 755 PBS/mpi.sh
```

Now submit the job to PBS as follows:

```
$ cd ~/PBS
$ qsub mpi.sh
$ qstat | grep 'whoami'
21303.bh1      mpi          gustav          0 R bg
$ !!
21303.bh1      mpi          gustav          0 R bg
$
```

After you have submitted the job, you can monitor its progress through the PBS system with

```
$ qstat | grep 'whoami'
```

every now and then. But the job should run quickly, unless the system is very busy. If everything works as it ought to, you will find `mpi_err` and `mpi_out` in your working directory after the job completes. The first file should be empty and the second will contain the output of the job, which should be similar to:

```
$ cat mpi_err
$ cat mpi_out
Local MPD console on bc68
bc68_33575
bc47_34123
bc46_34056
bc49_34697
bc48_33551
bc53_34095
bc54_35385
bc55_34714
time for 100 loops = 0.124682068825 seconds
0: bc68
2: bc46
1: bc47
3: bc49
4: bc48
6: bc54
5: bc53
7: bc55
bc68: hello world from process 0 of 8
bc47: hello world from process 1 of 8
bc46: hello world from process 2 of 8
bc49: hello world from process 3 of 8
bc48: hello world from process 4 of 8
bc53: hello world from process 5 of 8
bc54: hello world from process 6 of 8
bc55: hello world from process 7 of 8
$
```

This should work on both AVIDD clusters.

Let us have a look at the PBS script:

```
gustav@bh1 $ cat mpi.sh
#PBS -S /bin/bash
#PBS -N mpi
#PBS -o mpi_out
#PBS -e mpi_err
#PBS -q bg
#PBS -m a
#PBS -V
#PBS -l nodes=8
NODES=8
HOST='hostname'
echo Local MPD console on $HOST
# Specify Myrinet interfaces on the hostfile.
grep -v $HOST $PBS_NODEFILE | sed 's/$/-myri0/' > $HOME/mpd.hosts
# Boot the MPI2 engine.
mpdboot --totalnum=$NODES --file=$HOME/mpd.hosts
sleep 10
# Inspect if all MPI nodes have been activated.
```

```

mpdtrace -l
# Check the connectivity.
mpdringtest 100
# Check if you can run trivial non-MPI jobs.
mpdrun -l -n $NODES hostname
# Execute your MPI program.
mpiexec -n $NODES hellow2
# Shut down the MPI2 engine and exit the PBS script.
mpdallexit
exit 0
gustav@bh1 $

```

There is a new PBS directive here, which we haven't encountered yet. The option `-l` lets you specify the list of resources required for the job. In this case there is only one item in the list, `nodes=8`, and this item states that you need eight nodes from the PBS in order to run the job. PBS is going to return the names of the nodes on the file, whose name is conveyed in the environmental variable `PBS_NODEFILE`. The nodes are listed on this file one per line.

Also observe that we have used the `-V` directive. By doing this we have imported all environmental variables, including the wretched `MPD_USE_USER_CONSOLE`, which is essential for `MPICH2`.

The first thing we do in the script is to convert the node names to their Myrinet equivalents. This is easy to do, because the Myrinet names are obtained by appending `-myri0` to the name of the node, returned on `$PBS_NODEFILE`. This is what the first command in the script does:

```
grep -v $HOST $PBS_NODEFILE | sed 's/$/-myri0/' > $HOME/mpd.hosts
```

There is one complication here though. We are removing from this list the name of the node on which the script runs with the command `grep -v $HOST`. This is because `MPICH2` is going to create a process on this host anyway. If we left this host's name on the file, `MPICH2` would create two processes on it.

Once the names returned on `$PBS_NODEFILE` have been converted to the Myrinet names, we save them on `$HOME/mpd.hosts`. Now we are ready to start the `MPICH2` engine. The command that does this is

```
mpdboot --totalnum=$NODES --file=$HOME/mpd.hosts
```

Program `mpdboot` is a Python-2.3 script that boots the `MPICH2` engine by spawning `MPICH2` supervisory processes, called `mpds` (pronounced “em-pea-dee-s”), on nodes specified on `$HOME/mpd.hosts`.

We have to give `mpdboot` a few seconds to complete its job. We do this by telling the script to `sleep` for 10 seconds. The `mpds` are spawned by `ssh`, which is why we had to get all the keys in place in the first place. Silly problems may show up if the keys are not there.

The command

```
mpdtrace -l
```

inspects the `MPICH2` engine and lists names of all nodes on which `mpds` are running. With the `-l` option, it also lists the names of the sockets used by the `mpds` to communicate with each other.

The next command

```
mpdringtest 100
```

times a simple message going around the ring of mpds, in this case 100 times.

These two commands, `mpdtrace` and `mpdringtest`, tell us that the MPICH2 engine is ready. We can now execute programs on it. These don't have to be MPICH2 programs though. You can execute any UNIX program under the MPICH2 engine. But if they are not MPICH2 programs, they will not communicate with each other. You will just get a number of independent instantiations of those programs running on individual nodes. The script demonstrates this by running the UNIX command `hostname` under the MPICH2 engine:

```
mpdrun -l -n $NODES hostname
```

The option `-l` asks `mpdrun` to attach process labels to any output that the instantiations of `hostname` on MPICH2 nodes may produce.

At long last we commence the execution of a real MPICH2 program. The program's name is `helloworld`. It is an MPI version of "Hello World". This program should be picked up from your `$HOME/bin` assuming that it is present in your command search `PATH`. The command to run this program under the MPICH2 engine is

```
mpiexec -n $NODES hellow
```

Observe that we don't have to use *all* nodes given to us by the PBS, but, of course, it would be silly not to, unless there is a special reason for it. Program `mpiexec` is not specific to MPICH2. MPI-2 specification says that such a program must be provided for the execution of MPI jobs. There was no such specification in the original MPI.

After `helloworld` exits, we are done and we shut down the MPICH2 engine with

```
mpdallexit
```

## Exercises

- 1 You should try all this out from your account. Try asking for a larger number of nodes. Remember to change both the PBS directive `-l nodes=` and `NODES=` in the shell script.
- 2 There is another MPI program in `/N/B/gustav/bin`, called `cpi`. Copy this program to your `$HOME/bin` and modify the script above to execute `cpi` instead of `helloworld`. What does this program do?
- 3 You can startup, manipulate and close the MPICH2 engine interactively from the head node too, but only if you're quick. Otherwise the skulker that runs on the computational nodes will kill your MPD processes there. Try the following from your account:

```
gustav@bh1 $ cd
gustav@bh1 $ pwd
/N/B/gustav
gustav@bh1 $ ls -l mpd.hosts
-rw-r--r--  1 gustav  ucs          77 Oct  1 16:48 mpd.hosts
gustav@bh1 $ cat mpd.hosts
bc55-myri0
```

```
bc54-myri0
bc53-myri0
bc49-myri0
bc48-myri0
bc47-myri0
bc46-myri0
gustav@bh1 $ mpdboot
gustav@bh1 $
```

Observe that we have started `mpdboot` without any options. By default `mpdboot` will check if there is a file called `mpd.hosts` in your working directory. If such a file exists `mpdboot` will read host names from it and it will try to spawn mpds on those hosts.

```
gustav@bh1 $ mpdtrace -l
bh1_49212
bc47_34173
bc48_33601
bc46_34106
bc53_34145
bc49_34746
bc55_34766
bc54_35437
gustav@bh1 $ mpdringtest 1000
time for 1000 loops = 1.432543993 seconds
gustav@bh1 $ mpdrun -l -n 8 hostname
0: bh1
2: bc48
5: bc49
1: bc47
6: bc55
3: bc46
7: bc54
4: bc53
gustav@bh1 $ mpiexec -n 8 hellow2
bh1: hello world from process 0 of 8
bc48: hello world from process 2 of 8
bc47: hello world from process 1 of 8
bc46: hello world from process 3 of 8
bc49: hello world from process 5 of 8
bc53: hello world from process 4 of 8
bc55: hello world from process 6 of 8
bc54: hello world from process 7 of 8
gustav@bh1 $ mpdallexit
gustav@bh1 $
```

Please contact me if any of the above doesn't work for you.

## 5.2 Basic MPI

### 5.2.1 Hello World

Program `hellow2`, which you have executed on the MPICH2 engine in section 5.1.5 (hopefully), is a very slight modification of a similar program distributed

with MPICH2 source. We are going to have a look at this program here and I am also going to show you how to compile it, link and install.

Here is the program:

```
gustav@bh1 $ cat hellow2.c
/*
 * Find about the size of the communicator, your rank within it, and the
 * name of the processor you run on.
 *
 * %Id: hellow2.c,v 1.1 2003/09/29 15:58:12 gustav Exp %
 *
 * %Log: hellow2.c,v %
 * Revision 1.1 2003/09/29 15:58:12 gustav
 * Initial revision
 *
 */

#include <stdio.h> /* printf and BUFSIZ defined there */
#include <stdlib.h> /* exit defined there */
#include <mpi.h> /* all MPI-2 functions defined there */

int main(argc, argv)
int argc;
char *argv[];
{
    int rank, size, length;
    char name[BUFSIZ];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Get_processor_name(name, &length);

    printf("%s: hello world from process %d of %d\n", name, rank, size);

    MPI_Finalize();

    exit(0);
}
gustav@bh1 $
```

You have already seen how this program runs in section 5.1.5.

All MPI programs must begin with `MPI_Init(&argc, &argv)` and end with `MPI_Finalize()`. It is not an error to insert *non-MPI* program statements in front of `MPI_Init` or after `MPI_Finalize`.

Function `MPI_Init` initializes the MPI system. MPI processes are spawned and ranked. Communication channels get established. The default communicator, `MPI_COMM_WORLD`, is created. This function *must* be called before any other MPI function. But you must not call it again. Subsequent calls to `MPI_Init` will produce an error.

At the end of an MPI program you always *must* call `MPI_Finalize`. This function cleans up the MPI machine. Once it has been called no other MPI function will work. Not to call `MPI_Finalize` and `exit` will result in exit error.

But this function alone is not equivalent to the UNIX function `exit`. It only exits the MPI machine, not the UNIX process.

Between `MPI_Init` and `MPI_Finalize` you have an MPI program. Let's see what's there in our `hellow2` example.

The first call to `MPI_Comm_rank` informs every process that participates in the MPI communicator about its rank number. This is how processes can find out who they are and what their role is within the pool. This function takes two arguments. The first one is the MPI communicator, and here we use the default communicator, `MPI_COMM_WORLD`, which the MPI engine creates at the very beginning. All processes invoked by the program belong to this communicator. The second argument is a pointer to integer. On return, each process is going to get its rank number written on this location.

Why should the processes bother about their rank numbers? This is so that they can then differentiate their actions *depending* on this number. Otherwise they would all have to do exactly the same thing. In the case of this simple example program, they will write messages on standard output, and each MPI process will write a somewhat different message.

The next call to `MPI_Comm_Size` tells the participating processes about the total number of processes in this communicator. This is also something they need to know. For example, if the program is to work on a very long array, the processes will have to know how many of them there are in the pool in order to work out for themselves which portion of the array they should work on. Function `MPI_Comm_size` takes the MPI communicator as the first argument and returns the size of the communicator in the location pointed to by the second argument.

Finally we call function `MPI_Get_processor_name`. We call this function only out of curiosity. It is seldom necessary for the processes themselves to know what physical processors they run on. But this function is there in the MPI standard and it was meant to be used in process migration. Here we simply use it to show that the MPI processes indeed run on different CPUs. We could just as well have used the standard UNIX function `gethostname`, but this would return the same name if the MPI program was to run on a large SMP, or a large single parallel machine like, say, Cray X1. On the other hand, function `MPI_Get_processor_name` will in this case return a specific number of the physical processor a given MPI process runs on.

After the MPI processes have collected all this information, each of them prints on standard output a message about

- the specific CPU it runs on,
- its rank number within the `MPI_COMM_WORLD` communicator,
- the total number of processes in this communicator.

Now, how to make and install this program? Its Makefile looks as follows:

```
gustav@bh1 $ cat Makefile
#
```

```

# %Id: Makefile,v 1.1 2003/09/29 16:20:16 gustav Exp %
#
# %Log: Makefile,v %
# Revision 1.1 2003/09/29 16:20:16 gustav
# Initial revision
#
#
DESTDIR = /N/B/gustav/bin
MANDIR  = /N/B/gustav/man/man1
CC = cc
CFLAGS = -I/N/hpc/mpich2/include
LIBS = -L/N/hpc/mpich2/lib
LDFLAGS = -lmpich
TARGET = hellow2

all: $(TARGET)

$(TARGET): $(TARGET).o
        $(CC) -o $@ $(TARGET).o $(LIBS) $(LDFLAGS)

$(TARGET).o: $(TARGET).c
        $(CC) $(CFLAGS) -c $(TARGET).c

install: all $(TARGET).1
        [ -d $(DESTDIR) ] || mkdirhier $(DESTDIR)
        install $(TARGET) $(DESTDIR)
        [ -d $(MANDIR) ] || mkdirhier $(MANDIR)
        install $(TARGET).1 $(MANDIR)

clean:
        rm -f *.o $(TARGET)

clobber: clean
        rcs-clean
gustav@bh1 $

```

The CFLAGS tell the compiler to look for MPI-2 definitions in `/N/B/gustav/include`. Similarly, we tell the loader that it should get objects from the library in `/N/B/gustav/lib`. Finally, the specific library that should be used in `libmpich.a`, which is what the switch `-lmpich` means.

Here's how we make the program:

```

gustav@bh1 $ make
co RCS/Makefile,v Makefile
RCS/Makefile,v --> Makefile
revision 1.1
done
co RCS/hellow2.c,v hellow2.c
RCS/hellow2.c,v --> hellow2.c
revision 1.1
done
cc -I/N/hpc/mpich2/include -c hellow2.c
cc -o hellow2 hellow2.o -L/N/hpc/mpich2/lib -lmpich
gustav@bh1 $

```

And now we install it:

```

gustav@bh1 $ make install

```



```

co RCS/hellow2.1,v hellow2.1
RCS/hellow2.1,v --> hellow2.1
revision 1.1
done
[ -d /N/B/gustav/bin ] || mkdirhier /N/B/gustav/bin
install hellow2 /N/B/gustav/bin
[ -d /N/B/gustav/man/man1 ] || mkdirhier /N/B/gustav/man/man1
install hellow2.1 /N/B/gustav/man/man1
gustav@bh1 $

```

The manual page for the program describes how to run it under MPICH2:

```

HELLOW2(1)          I590 Programmer's Manual          HELLOW2(1)

```

#### NAME

hellow2 - for each MPI process print its rank number and name of the processor it runs on.

#### SYNOPSIS

```
mpiexec -n <number-of-processes> hellow2
```

#### DESCRIPTION

hellow2 prints the name of the processor, the process rank number and the size of the communicator for each MPI process.

#### OPTIONS

No hellow2 specific options

#### DIAGNOSTICS

No hellow2 specific diagnostics

#### EXAMPLES

```

$ mpdboot -n 8
$ mpiexec -n 8 hellow2
bc89: hello world from process 0 of 8
bc31: hello world from process 2 of 8
bc29: hello world from process 1 of 8
bc33: hello world from process 3 of 8
bc34: hello world from process 5 of 8
bc30: hello world from process 4 of 8
bc35: hello world from process 6 of 8
bc32: hello world from process 7 of 8
$ mpdallexit

```

#### AUTHOR

The simplest MPI program possible. Author unknown.

I590/7462

October 2003

HELLOW2(1)

You will find a script in `/N/hpc/mpich2/bin` called `mpicc`. This script knows about the location of includes and libraries and simplifies the compilation process. And so, instead of calling libraries and includes explicitly, as I have done in the Makefile, you could simply compile this program as follows:

```

gustav@bh1 $ which mpicc
/N/hpc/mpich2/bin/mpicc
gustav@bh1 $ mpicc -o hellow2 hellow2.c
gustav@bh1 $

```

## 5.2.2 Greetings, Master

The “Hello World” program from section 5.2.1 ran in parallel, but participating processes did not exchange any messages, so the parallelism was trivial.

In this section we’re going to have a look at our first non-trivial parallel program in which processes are going to exchange information.

We are going to designate one of the processes a master process. The master process will find out the name of the processor it runs on and it will broadcast the name to all other processes. They will respond by sending greetings to the master process. The master process will collect all greetings and it will then display them on standard output.

Here is the program:

```
gustav@bh1 $ cat greetings.c
/*
 * The master process broadcasts the name of the CPU it runs on to
 * the pool. All other processes respond by sending greetings
 * to the master process, which collects the messages and displays
 * them on standard output.
 *
 * %Id: greetings.c,v 1.2 2003/09/29 19:47:29 gustav Exp %
 *
 * %Log: greetings.c,v %
 * Revision 1.2 2003/09/29 19:47:29 gustav
 * Added "The" to the comment.
 *
 * Revision 1.1 2003/09/29 19:37:58 gustav
 * Initial revision
 *
 */
#include <stdio.h> /* functions sprintf, printf and BUFSIZ defined there */
#include <string.h> /* function strcpy defined there */
#include <stdlib.h> /* function exit defined there */
#include <mpi.h> /* all MPI-2 functions defined there */

#define TRUE 1
#define FALSE 0
#define MASTER_RANK 0 /* It is traditional to make process 0 the master. */

main(argc, argv)
    int argc;
    char *argv[];
{
    int count, pool_size, my_rank, my_name_length, i_am_the_master = FALSE;
    char my_name[BUFSIZ], master_name[BUFSIZ], send_buffer[BUFSIZ],
        recv_buffer[BUFSIZ];
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &pool_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Get_processor_name(my_name, &my_name_length);

    if (my_rank == MASTER_RANK) {
        i_am_the_master = TRUE;
```

```

    strcpy (master_name, my_name);
}

MPI_Bcast(master_name, BUFSIZ, MPI_CHAR, MASTER_RANK, MPI_COMM_WORLD);

if (i_am_the_master)
    for (count = 1; count < pool_size; count++) {
        MPI_Recv (recv_buffer, BUFSIZ, MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG,
                 MPI_COMM_WORLD, &status);
        printf ("%s\n", recv_buffer);
    }
else {
    sprintf(send_buffer, "hello %s, greetings from %s, rank = %d",
           master_name, my_name, my_rank);
    MPI_Send (send_buffer, strlen(send_buffer) + 1, MPI_CHAR,
             MASTER_RANK, 0, MPI_COMM_WORLD);
}

MPI_Finalize();

exit(0);
}

gustav@bh1 $
```

Although at the first glance this program does much the same as our previous example, `helloworld`, it does it differently and in a way that is immediately extensible to quite non-trivial MPI programs. It contains a number of new conceptual elements and some new MPI function calls.

It begins the same way any other MPI program begins. First we call `MPI_Init`, followed by `MPI_Comm_size`, `MPI_Comm_rank`, and finally `MPI_Get_processor_name`. We have discussed these already in section 5.2.1.

Then a new thing happens. The program forks. Only one of the MPI processes, namely the one whose rank is equal to `MASTER_RANK` executes the `if` clause:

```

if (my_rank == MASTER_RANK) {
    i_am_the_master = TRUE;
    strcpy (master_name, my_name);
}

```

whereas all other processes skip this clause, because it's not theirs, and call the next statement that follows, which is

```

MPI_Bcast(master_name, BUFSIZ, MPI_CHAR, MASTER_RANK, MPI_COMM_WORLD);

```

This statement blocks, i.e., the processes that get to it hang and wait *until* process `MASTER_RANK` broadcasts the information.

But the master process is not there yet. Within the `if` clause it sets the logical flag `i_am_the_master` to `TRUE`, whereas, by default, it's been initialized to `FALSE` and so it stays `FALSE` for everybody else. Then the master processes copies the name of the processor it runs on onto a string called `master_name`. At this stage the master process is the only process that has anything in this string. For all other processes `master_name` just contains garbage (you must not assume that UNIX is so kind as to initialize all strings to nulls).

Now the master process joins the broadcast. The broadcast operation works as follows. There is a root process for the broadcast, which is given by the fourth argument to `MPI_Bcast`. Here it is `MASTER_RANK`. The first argument is the buffer. Remember that each of the MPI processes has its own buffer that is pointed to by this variable and these buffers live on different machines and in different memory locations. But they are all referred to as `master_name`. Only the root process, which in our case is the master, has something sensible in its own buffer. The length of the buffer, here it is `BUFSIZ`, is given as the number of items of type that is specified by the third variable, in this case `MPI_CHAR`. In this case all processes execute this broadcast statement from the same line of the code (it doesn't have to be so normally), and this guarantees that their buffers are of identical length and that they will all interpret the data that is going to be sent in the same way, i.e., as characters. The last argument specifies the MPI communicator within which the communication is going to take place.

Now the root node broadcasts the content of its buffer to all other processes and at the end of this operation they all have the same stuff in their buffers.

This type of an operation is called a *collective* operation. Collective operations tend to be rather costly, not only because data is being sent over the whole pool of MPI processes (which may be very large), but also because this operation forces all processes to wait until the operation completes. This enforces synchronization onto the process pool and synchronization is always very costly in terms of CPU time and even wall-clock time.

Anyhow, the master process has broadcast the name of its CPU to all other processes and now the program forks again. The master process does something quite different from all other processes. This is expressed again by the means of the `if` statement, which has one clause for the master process and another clause for all other processes:

```

if (i_am_the_master)
    for (count = 1; count < pool_size; count++) {
        MPI_Recv (recv_buffer, BUFSIZ, MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG,
                 MPI_COMM_WORLD, &status);
        printf ("%s\n", recv_buffer);
    }
else {
    sprintf(send_buffer, "hello %s, greetings from %s, rank = %d",
           master_name, my_name, my_rank);
    MPI_Send (send_buffer, strlen(send_buffer) + 1, MPI_CHAR,
             MASTER_RANK, 0, MPI_COMM_WORLD);
}

```

Now you can fully appreciate why it is so important that MPI processes have ranks. Within its own clause the master process issues `count - 1` receives and follows each successful receive with a print statement that prints whatever the master has received on standard output. On the other hand all other processes construct messages within their own part of the program and send them to the master process. It is usually the case in sequential programs that only one clause of an `if` statement is executed. Here both are executed at the same time, but they run on different CPUs.

Let us have a look at the functions that send and receive data.

Function `MPI_Send` sends `strlen(send_buffer) + 1` items of type `MPI_CHAR` to process of rank `MASTER_RANK`. The items are taken from the buffer pointed to by `send_buffer` and the communication takes place within the `MPI_COMM_WORLD` communicator. The fifth variable, which is here set to 0, is the message tag. It is an arbitrary integer number that is used to differentiate between various messages. The receiving process may wish to receive messages with certain tags only.

Now the master process is going to receive the messages into its own receive buffer that is here called `recv_buffer`. This is the first argument to function `MPI_Recv`. The second argument specifies the length of the receive buffer measured as the number of items of type `MPI_CHAR`, which is the third argument to `MPI_Recv`. Observe that since we do not know a priori how long a message is going to be, we usually have to oversize the receive buffer, but this may be overcome in various ways. There are inquiry functions that can be used to find out how much data has arrived, before the data is actually read. The fourth argument to `MPI_Recv` specifies the rank of the process from which we want to receive the message. We don't have to receive messages as they come. We can make them wait and we can ignore them altogether too. But in this case we say that we want to receive messages from any source (`MPI_ANY_SOURCE`) and with any tag (`MPI_ANY_TAG`). The messages are going to be received within the `MPI_COMM_WORLD` communicator, and the status of the received message is going to be written on a structure called `status`. We will ignore this last bit for the time being.

Let me show you how this is made, installed and run.

```
gustav@bh1 $ pwd
/N/B/gustav/src/I590/greetings
gustav@bh1 $ make
co RCS/Makefile,v Makefile
RCS/Makefile,v --> Makefile
revision 1.1
done
co RCS/greetings.c,v greetings.c
RCS/greetings.c,v --> greetings.c
revision 1.2
done
cc -I/N/hpc/mpich2/include -c greetings.c
cc -o greetings greetings.o -L/N/hpc/mpich2/lib -lmpich
gustav@bh1 $ make install
co RCS/greetings.1,v greetings.1
RCS/greetings.1,v --> greetings.1
revision 1.1
done
[ -d /N/B/gustav/bin ] || mkdirhier /N/B/gustav/bin
install greetings /N/B/gustav/bin
[ -d /N/B/gustav/man/man1 ] || mkdirhier /N/B/gustav/man/man1
install greetings.1 /N/B/gustav/man/man1
gustav@bh1 $
```

The manual page looks like this:

```
GREETINGS(1)          I590 Programmer's Manual          GREETINGS(1)
```

## NAME

greetings - send greetings to the master process

## SYNOPSIS

```
mpiexec -n <number-of-processes> greetings
```

## DESCRIPTION

`greetings` The master process finds about the name of the CPU it runs on and broadcasts it to all. Having received the broadcast other processes construct greeting messages and send them back to the master, who collects them and displays on standard output.

## OPTIONS

No greetings specific options

## DIAGNOSTICS

No greetings specific diagnostics

## EXAMPLES

```
$ mpdboot -n 8
$ mpiexec -n 8 greetings
hello bc89, greetings from bc46, rank = 4
hello bc89, greetings from bc42, rank = 2
hello bc89, greetings from bc43, rank = 1
hello bc89, greetings from bc47, rank = 5
hello bc89, greetings from bc48, rank = 6
hello bc89, greetings from bc44, rank = 3
hello bc89, greetings from bc88, rank = 7
$ mpdallexit
```

## AUTHOR

Still too trivial a program to claim authorship.

I590/7462

October 2003

GREETINGS(1)

I run this program from the following PBS script:

```
gustav@bh1 $ pwd
/N/B/gustav/PBS
gustav@bh1 $ cat greetings.sh
#PBS -S /bin/bash
#PBS -N greetings
#PBS -o greetings_out
#PBS -e greetings_err
#PBS -q bg
#PBS -V
#PBS -m a
#PBS -l nodes=8
NODES=8
HOST='hostname'
echo Local MPD console on $HOST
grep -v $HOST $PBS_NODEFILE | sed 's/$/-myri0/' > $HOME/mpd.hosts
mpdboot --totalnum=$NODES --file=$HOME/mpd.hosts
sleep 10
mpiexec -n $NODES greetings
mpdallexit
exit 0
```

```

gustav@bh1 $ qsub greetings.sh
20674.bh1.avidd.iu.edu
gustav@bh1 $ qstat | grep gustav
20674.bh1      greetings      gustav                O R bg
gustav@bh1 $ !!
qstat | grep gustav
gustav@bh1 $ cat greetings_err
gustav@bh1 $ cat greetings_out
Local MPD console on bc89
hello bc89, greetings from bc43, rank = 4
hello bc89, greetings from bc40, rank = 1
hello bc89, greetings from bc41, rank = 2
hello bc89, greetings from bc44, rank = 5
hello bc89, greetings from bc46, rank = 6
hello bc89, greetings from bc42, rank = 3
hello bc89, greetings from bc88, rank = 7
gustav@bh1 $

```

Observe that the messages do not arrive in any particular order.

This program is a very simple prototype of what you would have to do in order to implement non-parallel IO in absence of MPI-IO. The processes would have to send their data to a master process, which would then collect and organize the data and write it out on a file using normal UNIX IO. Until MPI-IO all MPI programs used to checkpoint this way.

### Exercises

- 1 Modify program `greetings` discussed in section 5.2.2 so that the master process *receives* the messages not out of order but in the order that agrees with the ranking of the senders.
- 2 Write an MPI program in which processes fill their send buffers with random integers instead of greeting strings. Make the send buffers (and the corresponding receive buffer of the master process) 1,048,576 integers long. Make the master process append the received strings of random integers to an external file, the way that the program `mkrandfile` discussed in section 3.4.3 worked.

**Hint** You will have to replace `MPI_CHAR` with `MPI_INT` in `MPI_Send` and `MPI_Recv`. Remember that the length of send and receive buffers must be given in terms of the number of items of a type specified by the next argument, not in bytes.

**Hint** Use the process rank number so that a different seed for function `srand` is generated within each MPI process. Otherwise they will all generate identical strings of pseudo-random integers.

**Hint** Do not make the master process generate any numbers. It will be busy enough collecting data and writing the file.

### 5.2.3 Dividing the Pie

We are now going to analyze the program you should have run already, `cpi`. The source of this program is distributed with MPICH2, and it lives in the directory `/N/hpc/mpich2/src/mpich2-0.94b1/examples`. The file name is `cpi.c`. For your convenience and because it is an open software program, I also quote it below:

```
gustav@bh1 $ pwd
/N/hpc/mpich2/src/mpich2-0.94b1/examples
gustav@bh1 $ cat cpi.c
#include "mpi.h"
#include <stdio.h>
#include <math.h>

double f(double);

double f(double a)
{
    return (4.0 / (1.0 + a*a));
}

int main(int argc, char *argv[])
{
    int done = 0, n, myid, numprocs, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;
    double startwtime = 0.0, endwtime;
    int namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Get_processor_name(processor_name, &namelen);

    fprintf(stdout, "Process %d of %d is on %s\n",
            myid, numprocs, processor_name);
    fflush(stdout);

    n = 10000;                /* default # of rectangles */
    if (myid == 0)
        startwtime = MPI_Wtime();

    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    h = 1.0 / (double) n;
    sum = 0.0;
    /* A slightly better approach starts from large i and works back */
    for (i = myid + 1; i <= n; i += numprocs)
    {
        x = h * ((double)i - 0.5);
        sum += f(x);
    }
    mypi = h * sum;

    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```



```

if (myid == 0) {
    endwtime = MPI_Wtime();
    printf("pi is approximately %.16f, Error is %.16f\n",
          pi, fabs(pi - PI25DT));
    printf("wall clock time = %f\n", endwtime-startwtime);
    fflush(stdout);
}

MPI_Finalize();
return 0;
}
gustav@bh1 $

```

This program calculates  $\pi$  by using the formula

$$\pi = 4 \int_0^1 \frac{1}{1+x^2} dx \quad (5.1)$$

The calculation is numerical, i.e., the segment  $[0, 1]$  is divided into 10,000 equal fragments. Within each fragment we evaluate  $\frac{4}{1+x^2}$  for the middle of the fragment (this is the height of the rectangle) and multiply it by the width of the fragment to obtain the area of the rectangle. Then we add all the areas together and this is our integral, that is also the approximate value of  $\pi$ .

The program begins with the usual incantations to `MPI_Init`, `MPI_Comm_size`, `MPI_Comm_rank` and `MPI_Get_processor_name`. Then each process writes on standard output its rank number (which is called `myid` here), the size of the communicator (which is called `numprocs` here) and the name of the processor it runs on.

We have not talked about it so far, but this writing on standard output is far from trivial. Every process writes on *its own* standard output. In some early versions of MPI, e.g., in LAM MPI, these writes were simply lost. But MPICH2 engine collects all standard outputs from its processes and combines them together on the so called *console node*. This is always the node whose rank is zero, the node on which we run `mpdboot`. The output you see on the PBS output file is the output from the MPICH2 console node.

Now we hardwire the number of division of segment  $[0, 1]$  into `n` and the console node begins the timing of the program by calling function `MPI_Wtime`. This function measures wall-clock time down to a *microsecond*, sic! It is very accurate indeed.

Now the value of `n` is broadcast to all processes. This is completely unnecessary in this particular program, because all nodes know what `n` is from the very beginning, but this program has been modified from its interactive predecessor, which used to read `n` from standard input. It was the console node that used to do the reading.

Now every process calculates the width of the rectangles:

```
h = 1.0 / (double) n;
```

Every process initializes its own local variable `sum` to zero, and then calls  $f(x) = \frac{4}{1+x^2}$  on... the rectangle that corresponds to its own rank number. Then it jumps to the rectangle that corresponds to its rank number plus the number of processes in the pool, and so on. The values of  $f(x) = \frac{4}{1+x^2}$  are added for all rectangles and only at the end they are multiplied by the width of the rectangle. This, of course, saves on unnecessary multiplications.

Now we encounter a new operation, `MPI_Reduce`. This operation takes as its input the content of `my_pi` for each process in the pool (the first argument). We tell it that there is one item (the third argument) of type `MPI_DOUBLE` (the fourth argument) in there. We are then telling it that it should perform summation over all instances of `my_pi` (the fifth argument, `MPI_SUM`) and that it should write the result of the operation on `pi` (the second argument) on the root process (the sixth argument), which in this case is the console process, i.e., process of rank zero. The whole operation is performed within the `MPI_COMM_WORLD` communicator.

At the end of this operation only the root (console) process has the right answer in its location that corresponds to `pi`. But this is fine. In the next clause we instruct the console process to measure the wall clock time again, then print the value of  $\pi$  on standard output together with the estimate of the accuracy of the computation. Then the console process also writes how long it took to carry out the computation.

Here is the output of the program.

```
gustav@bh1 $ pwd
/N/B/gustav/PBS
gustav@bh1 $ cat mpi_out
Local MPD console on bc89
Process 0 of 8 is on bc89
Process 1 of 8 is on bc40
Process 2 of 8 is on bc42
Process 3 of 8 is on bc41
Process 4 of 8 is on bc44
Process 5 of 8 is on bc43
Process 6 of 8 is on bc88
Process 7 of 8 is on bc46
pi is approximately 3.1415926544231247, Error is 0.000000008333316
wall clock time = 0.009541
gustav@bh1 $
```

`MPI_SUM` is not the only operation you can use with `MPI_Reduce`. `MPI` provides a number of predefined reduction operations that can be used in this context, and you can define your own operations too. The predefined ones are:

<code>MPI_MAX</code>	<code>MPI_MIN</code>	<code>MPI_SUM</code>
<code>MPI_PROD</code>	<code>MPI_LAND</code>	<code>MPI_BAND</code>
<code>MPI_LOR</code>	<code>MPI_BOR</code>	<code>MPI_LXOR</code>
<code>MPI_BXOR</code>	<code>MPI_MAXLOC</code>	<code>MPI_MINLOC</code>

### Exercises

- 1 Rewrite the program discussed in section 5.2.3 to read  $n$  from standard input, so that the broadcast operation actually makes sense.

**Hint** Make only the console process read standard input, then broadcast the number to other processes as shown in section 5.2.3.

**Hint** Input the number in the PBS script using the *here input* feature of the shell.

### 5.2.4 Bank Queue

The following program illustrates a rather important parallel programming technique, which is often referred to as a *job queue* or a *bank queue* paradigm. The idea is as follows: you have a number of jobs that you need to attend to, and that are not dependent on each other. The master process maintains the job queue, and sends jobs to slave processes. The slaves labour on the jobs and return the results back to the master. Once a slave has finished working on its last assignment and returned the results to the master, a new job is sent to the slave. That way all processes of an MPI farm are kept busy.

The paradigm can be enriched. For example the slave processes can return not only answers to the master, but also new jobs. The master process may assess those jobs, perhaps compare them to a list of jobs already done that it may keep on a lightweight data base, and if a job is new indeed, it may be added to the queue, whereas if there is already an answer available to that job, the answer may be passed to the slave together with the information that this class of problems has already been solved. This way the queue may grow and shrink dynamically as the computation proceeds.

This is a very flexible paradigm and it is often used to traverse dynamically growing trees, especially in artificial reasoning programs.

The master itself may take part in the work, other than just maintaining the queue and communicating with the slaves. But if there is a lot of work and a lot of slaves, the master process may be too busy attending to the queue and communicating with the slaves to carry out any computation of its own.

When you write a program based on this paradigm you must remember that communication is expensive. Consequently jobs sent to the slaves must be substantial enough to occupy them for a very long time. It is very easy to overload the master process if task granularity is too small. In that case the communication and the inability of the master process to respond quickly enough to slaves' requests may easily become a bottleneck.

Here's the program itself.

In this program we are simply going to multiply matrix  $\mathbf{A}$  by vector  $\mathbf{b}$  in parallel. Every slave process will have its own copy of  $\mathbf{b}$ , and the master is going to send them rows of  $\mathbf{A}$  to multiply by their own copy of  $\mathbf{b}$ . The result of the computation by a slave is going to be a corresponding entry in vector  $\mathbf{c} = \mathbf{A} \cdot \mathbf{b}$ .

A slave process is going to deliver the calculated entry in vector  $c$  back to the master process, which will then place it in an appropriate slot in  $c$  and pass a new row to the slave process at the same time, if there is any more work still to be done. If there is no more work, the master sacks the slave process, but sending it a termination message.

```

/*
 * %Id: bank.c,v 1.1 2003/10/05 19:52:15 gustav Exp %
 *
 * %Log: bank.c,v %
 * Revision 1.1 2003/10/05 19:52:15 gustav
 * Initial revision
 *
 */

#include <stdio.h>      /* [fs]printf, fopen and fclose defined here */
#include <stdlib.h>     /* exit defined here */
#include <sys/types.h> /* chmod defined here */
#include <sys/stat.h>  /* chmod defined here */
#include <mpi.h>

#define COLS 100
#define ROWS 100
#define TRUE 1
#define FALSE 0
#define MASTER_RANK 0

int main ( int argc, char **argv )
{
    int pool_size, my_rank, destination;
    int i_am_the_master = FALSE;
    int a[ROWS][COLS], b[COLS], c[ROWS], i, j;
    int int_buffer[BUFSIZ];
    char my_logfile_name[BUFSIZ];
    FILE *my_logfile;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &pool_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank == MASTER_RANK) i_am_the_master = TRUE;

    sprintf(my_logfile_name, "/N/B/gustav/tmp/bank.%d", my_rank);
    my_logfile = fopen(my_logfile_name, "w");
    (void) chmod(my_logfile_name, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);

    if (i_am_the_master) {

        int row, count, sender;

        for (j = 0; j < COLS; j++) {
            b[j] = 1;
            for (i = 0; i < ROWS; i++) a[i][j] = i;
        }
    }
}

```

```

}

MPI_Bcast(b, COLS, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);

count = 0;
for (destination = 0; destination < pool_size; destination++) {
    if (destination != my_rank) {
        for (j = 0; j < COLS; j++) int_buffer[j] = a[count][j];
        MPI_Send(int_buffer, COLS, MPI_INT, destination, count,
                 MPI_COMM_WORLD);
        fprintf(my_logfile, "sent row %d to %d\n", count, destination);
        count = count + 1;
    }
}

for (i = 0; i < ROWS; i++) {
    MPI_Recv (int_buffer, BUFSIZ, MPI_INT, MPI_ANY_SOURCE,
              MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    sender = status.MPI_SOURCE;
    row = status.MPI_TAG;
    c[row] = int_buffer[0];
    fprintf(my_logfile, "\treceived row %d from %d\n", row, sender);
    if (count < ROWS) {
        for (j = 0; j < COLS; j++) int_buffer[j] = a[count][j];
        MPI_Send(int_buffer, COLS, MPI_INT, sender, count,
                 MPI_COMM_WORLD);
        fprintf(my_logfile, "sent row %d to %d\n", count, sender);
        count = count + 1;
    }
    else {
        MPI_Send(NULL, 0, MPI_INT, sender, ROWS, MPI_COMM_WORLD);
        fprintf(my_logfile, "terminated process %d with tag %d\n", sender, ROWS);
    }
}
for (row = 0; row < ROWS; row++) printf("%d ", c[row]);
printf("\n");

}
else { /* I am not the master */

    int sum, row;

    MPI_Bcast(b, COLS, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);
    fprintf(my_logfile, "received broadcast from %d\n", MASTER_RANK);
    MPI_Recv(int_buffer, COLS, MPI_INT, MASTER_RANK, MPI_ANY_TAG,
              MPI_COMM_WORLD, &status);
    fprintf(my_logfile, "received a message from %d, tag %d\n",
              status.MPI_SOURCE, status.MPI_TAG);
    while (status.MPI_TAG != ROWS) { /* The job is not finished */
        row = status.MPI_TAG; sum = 0;
        for (i = 0; i < COLS; i++) sum = sum + int_buffer[i] * b[i];
        int_buffer[0] = sum;
        MPI_Send (int_buffer, 1, MPI_INT, MASTER_RANK, row, MPI_COMM_WORLD);
        fprintf(my_logfile, "sent row %d to %d\n", row, MASTER_RANK);
        MPI_Recv (int_buffer, COLS, MPI_INT, MASTER_RANK, MPI_ANY_TAG,
                  MPI_COMM_WORLD, &status);
        fprintf(my_logfile, "received a message from %d, tag %d\n",

```

```

        status.MPI_SOURCE, status.MPI_TAG);
    }
    fprintf(my_logfile, "exiting on tag %d\n", status.MPI_TAG);
}

fclose (my_logfile);

MPI_Finalize ();

exit (0);
}

```

The code is compiled and installed with a very simple Makefile:

```

#
# %Id: Makefile,v 1.2 2003/10/05 19:56:13 gustav Exp %
#
# %Log: Makefile,v %
# Revision 1.2 2003/10/05 19:56:13 gustav
# Fixed DESTDIR
#
# Revision 1.1 2003/10/05 19:54:55 gustav
# Initial revision
#
#
DESTDIR = /N/B/gustav/bin

all: bank

bank: bank.c
    mpicc -o bank bank.c

install: all
    install bank $(DESTDIR)

clean:
    rm -f bank.o bank

clobber: clean
    rcs clean

```

And it all works as follows:

```

gustav@bh1 $ pwd
/N/B/gustav/src/I590/bank
gustav@bh1 $ make install
co RCS/Makefile,v Makefile
RCS/Makefile,v --> Makefile
revision 1.2
done
co RCS/bank.c,v bank.c
RCS/bank.c,v --> bank.c
revision 1.1
done
mpicc -o bank bank.c
install bank /N/B/gustav/bin

```

```

gustav@bh1 $ cd
gustav@bh1 $ [ -d tmp ] || mkdir tmp
gustav@bh1 $ rm -f tmp/*
gustav@bh1 $ mpdtrace
bh1
bc34
bc35
bc37
bc36
bc38
bc40
bc39
gustav@bh1 $ mpiexec -n 8 bank
0 100 200 300 400 500 600 700 800 900 1000 1100 1200 1300 1400 1500 1600 \
1700 1800 1900 2000 2100 2200 2300 2400 2500 2600 2700 2800 2900 3000 3100 \
3200 3300 3400 3500 3600 3700 3800 3900 4000 4100 4200 4300 4400 4500 4600 \
4700 4800 4900 5000 5100 5200 5300 5400 5500 5600 5700 5800 5900 6000 6100 \
6200 6300 6400 6500 6600 6700 6800 6900 7000 7100 7200 7300 7400 7500 7600 \
7700 7800 7900 8000 8100 8200 8300 8400 8500 8600 8700 8800 8900 9000 9100 \
9200 9300 9400 9500 9600 9700 9800 9900
gustav@bh1 $ ls tmp
bank.0 bank.1 bank.2 bank.3 bank.4 bank.5 bank.6 bank.7
gustav@bh1 $

```

Files `bank.0` through `bank.7` contain logs from slaves and from the master. We'll have a look at those files later.

But first let us analyze this program in detail. Matrix  $A$  is  $100 \times 100$ . Its dimensions are fixed by

```

#define COLS 100
#define ROWS 100

```

The master process is going to be the process of rank 0:

```

#define MASTER_RANK 0

```

All processes begin their existence from the usual MPI incantations:

```

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &pool_size);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

```

through which they learn about their rank numbers and the size of the pool. Then the master process learns that about its masterhood:

```

if (my_rank == MASTER_RANK) i_am_the_master = TRUE;

```

The variable `i_am_the_master` is `FALSE` by default and it remains so for all other processes.

Then all processes open their own respective log files:

```

sprintf(my_logfile_name, "/N/B/gustav/tmp/bank.%d", my_rank);
my_logfile = fopen(my_logfile_name, "w");
(void) chmod(my_logfile_name, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);

```

Observe that all processes are going to log on the NFS in my `$HOME`. The files that correspond to the processes are numbered by their ranks, which are appended to the string `bank..` The name of the log file is *hardwired* in this program. You

will have to change this if you want to recompile this program and run it from your own account. Furthermore the program does not check for the existence of the directory `/N/B/gustav/tmp`, so this directory must be created before the program is run. This is a typical example of sloppy programming. You would not put anything like this in a commercial application. But, if you are in the process of developing a parallel program, and something doesn't work, and you have to debug every process and all communications that take place, then this is a legitimate way of doing things. It is a parallel equivalent of debugging a sequential program with `printf` statements.

Function `fopen` is going to open the file with rather liberal permissions (e.g., it may be open for writing to all users on the system). To make sure that the permissions are as you expect them to be, e.g., `644`, you can use the `chmod` system call. Here I set the permissions to be `-rw-r--r--`. The constants used in the permissions and the function `chmod` itself are defined on `/usr/include/sys/stat.h`. The default file mode for opening is `0666`.

Now the program splits into two subprograms. There is a separate subprogram for the master and a separate subprogram for the slaves:

```
...

    if (i_am_the_master) {
        blah... blah... blah...
    }
    else { /* I am not the master */
        blah... blah... blah...
    }

    fclose (my_logfile);
    MPI_Finalize();
    exit(0);
}
```

We are going to discuss these subprograms in the separate sections.

### The Master Program

I reprint the master program here for your convenience.

```
    if (i_am_the_master) {

        int row, count, sender;

        for (j = 0; j < COLS; j++) {
            b[j] = 1;
            for (i = 0; i < ROWS; i++) a[i][j] = i;
        }

        MPI_Bcast(b, COLS, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);

        count = 0;
        for (destination = 0; destination < pool_size; destination++) {
            if (destination != my_rank) {
                for (j = 0; j < COLS; j++) int_buffer[j] = a[count][j];
                MPI_Send(int_buffer, COLS, MPI_INT, destination, count,
```



```

        MPI_COMM_WORLD);
    fprintf(my_logfile, "sent row %d to %d\n", count, destination);
    count = count + 1;
}
}

for (i = 0; i < ROWS; i++) {
    MPI_Recv (int_buffer, BUFSIZ, MPI_INT, MPI_ANY_SOURCE,
             MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    sender = status.MPI_SOURCE;
    row = status.MPI_TAG;
    c[row] = int_buffer[0];
    fprintf(my_logfile, "\treceived row %d from %d\n", row, sender);
    if (count < ROWS) {
        for (j = 0; j < COLS; j++) int_buffer[j] = a[count][j];
        MPI_Send(int_buffer, COLS, MPI_INT, sender, count,
                MPI_COMM_WORLD);
        fprintf(my_logfile, "sent row %d to %d\n", count, sender);
        count = count + 1;
    }
    else {
        MPI_Send(NULL, 0, MPI_INT, sender, ROWS, MPI_COMM_WORLD);
        fprintf(my_logfile,
                "terminated process %d with tag %d\n", sender, ROWS);
    }
}
for (row = 0; row < ROWS; row++) printf("%d ", c[row]);
printf("\n");
}

```

The first thing that the master process does is to initialize vector  $\mathbf{b}$  and matrix  $\mathbf{A}$ . Every entry in vector  $\mathbf{b}$  is set to 1 and matrix  $\mathbf{A}$  is set to  $A_{ij} = i$ . This way it will be easy to check that the computation is correct, because

$$c_i = \sum_j A_{ij} b_j = \sum_j A_{ij} = i \times N$$

where  $N$  is the dimension of the matrix, in this case 100. So

$$c_i = 100 \times i$$

Once the master process has initialized  $\mathbf{A}$  and  $\mathbf{b}$  it broadcasts  $\mathbf{b}$  to all other processes:

```
MPI_Bcast(b, COLS, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);
```

Observe that this statement is issued in the master's part of the program. Consequently we will have to issue it again in the slaves' part of the program. Otherwise there won't be anybody on the other side to receive the data.

Now the master process initializes the counter, `count`, which is going to be used to number rows of matrix  $\mathbf{A}$  sent to the slave processes, and sends the first batch of jobs to all slave processes. In this program the master does not participate in the computation, so it does not send a row to itself:

```
count = 0;
for (destination = 0; destination < pool_size; destination++) {
```

```

    if (destination != my_rank) {
        for (j = 0; j < COLS; j++) int_buffer[j] = a[count][j];
        MPI_Send(int_buffer, COLS, MPI_INT, destination, count,
                MPI_COMM_WORLD);
        fprintf(my_logfile, "sent row %d to %d\n", count, destination);
        count = count + 1;
    }
}

```

For clarity, I have made the master process transfer a row from  $\mathbf{A}$  to a send buffer called `int_buffer`, and then send the data to the slave process. The data could be sent directly from matrix  $\mathbf{A}$ , which would be faster, but then you would have to remember how C stores matrices (it stores them in a row-major fashion, so this would actually work in this program).

Every time a matrix row is sent to a slave process, the master process logs its actions on the log file.

The following `for` loop is the tricky part of the master program. The master process waits for a message to arrive, from any process, and from any source.

```

for (i = 0; i < ROWS; i++) {
    MPI_Recv (int_buffer, BUFSIZ, MPI_INT, MPI_ANY_SOURCE,
             MPI_ANY_TAG, MPI_COMM_WORLD, &status);
}

```

How is the master process to know, which slave process is going to be the first with an answer? Some may be slower and busier than others depending on what else runs on their CPUs.

Once a message has arrived, the master process checks where the message has come from by inspecting the status structure associated with the message:

```

sender = status.MPI_SOURCE;

```

and inspects the tag number of the message, so that it is reminded about the row number that the answer relates to (the slave process, of course, will have to send the message so that the row number is conveyed in the tag):

```

row = status.MPI_TAG;

```

The answer itself is then placed in an appropriate slot of vector `c`:

```

c[row] = int_buffer[0];

```

The operation is the logged on the logfile:

```

fprintf(my_logfile, "\treceived row %d from %d\n", row, sender);

```

Now the master process has to respond to the slave process that was so kind to deliver the answer. The master process checks if there is still any work left:

```

if (count < ROWS) {
    blah... blah... blah...
}
else {
    MPI_Send(NULL, 0, MPI_INT, sender, ROWS, MPI_COMM_WORLD);
    fprintf(my_logfile,
            "terminated process %d with tag %d\n", sender, ROWS);
}

```

and if there isn't, it sends a null message to the slave process in question, whose tag is `ROWS`, and logs it on the log file.

Now, in C arrays are numbered from 0 through `length - 1`, and all processes know that matrix  $A$  has `ROWS` rows, numbered from 0 through `ROWS - 1`. So if the tag of the message is `ROWS` the slave process is going to know that something's amiss. As a matter of fact it will know that this is a termination message, so it will go away and wait on `MPI_Finalize` for other processes to finish.

If there is still some work left though, then the master process transfers the corresponding row of matrix  $A$  to its send buffer, `int_buffer`, and sends its content to the slave process that has just delivered the answer. This operation is again logged on the log file, and the counter, that counts how many rows of matrix  $A$  have been sent out so far, is incremented by 1:

```
for (j = 0; j < COLS; j++) int_buffer[j] = a[count][j];
MPI_Send(int_buffer, COLS, MPI_INT, sender, count,
         MPI_COMM_WORLD);
fprintf(my_logfile, "sent row %d to %d\n", count, sender);
count = count + 1;
```

After the master process has collected answers to all problems that it sent to the slave processes, there is nothing else left for it to do. It prints the result of the computation on standard output:

```
for (row = 0; row < ROWS; row++) printf("%d ", c[row]);
printf("\n");
```

then joins the mainlines of the program: it will close the log file and hit `MPI_Finalize()` where all the slave processes should wait for it already.

### The Slave Program

Let us have a look at the slave program. It is reprinted below for your convenience.

```
else { /* I am not the master */

    int sum, row;

    MPI_Bcast(b, COLS, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);
    fprintf(my_logfile, "received broadcast from %d\n", MASTER_RANK);
    MPI_Recv(int_buffer, COLS, MPI_INT, MASTER_RANK, MPI_ANY_TAG,
            MPI_COMM_WORLD, &status);
    fprintf(my_logfile, "received a message from %d, tag %d\n",
            status.MPI_SOURCE, status.MPI_TAG);
    while (status.MPI_TAG != ROWS) { /* The job is not finished */
        row = status.MPI_TAG; sum = 0;
        for (i = 0; i < COLS; i++) sum = sum + int_buffer[i] * b[i];
        int_buffer[0] = sum;
        MPI_Send (int_buffer, 1, MPI_INT, MASTER_RANK, row, MPI_COMM_WORLD);
        fprintf(my_logfile, "sent row %d to %d\n", row, MASTER_RANK);
        MPI_Recv (int_buffer, COLS, MPI_INT, MASTER_RANK, MPI_ANY_TAG,
                MPI_COMM_WORLD, &status);
        fprintf(my_logfile, "received a message from %d, tag %d\n",
                status.MPI_SOURCE, status.MPI_TAG);
    }
    fprintf(my_logfile, "exiting on tag %d\n", status.MPI_TAG);
}
```

The slave processes begin their career by receiving vector  $\mathbf{b}$  that has been broadcast by the master. This is the slave-side of the broadcast call matching the master's call:

```
MPI_Bcast(b, COLS, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);
```

After they have received their copies of  $\mathbf{b}$ , they log this event on their respective log files and wait for the first batch of jobs to be sent to them by the master process.

```
MPI_Recv(int_buffer, COLS, MPI_INT, MASTER_RANK, MPI_ANY_TAG,
        MPI_COMM_WORLD, &status);
```

They log this event on their log files too. Then they get to work. The work is done within the large

```
while (status.MPI_TAG != ROWS) { /* The job is not finished */
    blah... blah... blah...
}
```

loop. Every time a slave process receives a message from the master process it checks if the tag of the message is less than ROWS. Remember that having received a message with tag ROWS implies the termination of the contract.

If the tag is kosher, the slave process does the following:

```
row = status.MPI_TAG; sum = 0;
for (i = 0; i < COLS; i++) sum = sum + int_buffer[i] * b[i];
int_buffer[0] = sum;
MPI_Send (int_buffer, 1, MPI_INT, MASTER_RANK, row, MPI_COMM_WORLD);
fprintf(my_logfile, "sent row %d to %d\n", row, MASTER_RANK);
```

The row number is extracted from the tag of the message. Then the slave process evaluates  $\sum_j A_{ij}b_j$  and sends it back to the master using the same tag. This way the master will know which row number the answer corresponds to. This operation is logged on the log file too.

Finally, the slave process waits for another message from the master process, reads it, and logs it on the log file:

```
MPI_Recv (int_buffer, COLS, MPI_INT, MASTER_RANK, MPI_ANY_TAG,
        MPI_COMM_WORLD, &status);
fprintf(my_logfile, "received a message from %d, tag %d\n",
        status.MPI_SOURCE, status.MPI_TAG);
```

Then it's back to the top of the loop: check the tag, if the tag is OK perform the computation, otherwise print a farewell message on the log file, close the file and hit `MPI_Finalize()`.

## The Logs

Let us have a look at the logs in `/N/B/gustav/tmp`.

```
gustav@bh1 $ pwd
/N/B/gustav/tmp
gustav@bh1 $ ls
bank.0 bank.1 bank.2 bank.3 bank.4 bank.5 bank.6 bank.7
gustav@bh1 $
```

The master's log is on `bank.0`. It begins with:

```
sent row 0 to 1
sent row 1 to 2
sent row 2 to 3
sent row 3 to 4
sent row 4 to 5
sent row 5 to 6
sent row 6 to 7
```

This is followed by the routine work. Answers are received from the workers and new jobs are sent back to them:

```
    received row 1 from 2
sent row 7 to 2
    received row 0 from 1
sent row 8 to 1
    received row 3 from 4
sent row 9 to 4
...
```

Eventually the whole computation is complete and workers have to be sent termination messages:

```
    received row 92 from 3
terminated process 3 with tag 100
    received row 94 from 5
terminated process 5 with tag 100
    received row 95 from 6
terminated process 6 with tag 100
    received row 96 from 7
terminated process 7 with tag 100
    received row 97 from 4
terminated process 4 with tag 100
    received row 98 from 2
terminated process 2 with tag 100
    received row 99 from 1
terminated process 1 with tag 100
```

Now let us have a look at the log file of worker number 3:

```
gustav@bh1 $ cat bank.3
received broadcast from 0
received a message from 0, tag 2
sent row 2 to 0
received a message from 0, tag 10
sent row 10 to 0
received a message from 0, tag 16
sent row 16 to 0
...
received a message from 0, tag 79
sent row 79 to 0
received a message from 0, tag 86
sent row 86 to 0
received a message from 0, tag 92
sent row 92 to 0
received a message from 0, tag 100
exiting on tag 100
gustav@bh1 $
```

The row this worker sends to 0 is the row of vector  $\mathbf{c}$ , i.e., just a single integer.

The idea behind *job queue* paradigm is that you keep all processes as busy as they can get, even if some have to cope with more load than others, e.g., because they run on faster machines. But this works only if the workers don't have to wait for new jobs from the master process longer than it takes them to execute the jobs themselves. If this happens, the cost of communication outweighs the cost of computation so much that you'll be better off executing the whole computation sequentially.

### Exercises

- 1 Insert `MPI_Wtime` calls in the client portion of the code in order to measure time spent on computation and time spent on communication. Make each client write the totals on its log file before calling `MPI_Finalize`.
- 2 The program discussed in this section opens the log files in the directory whose name is hardwired into the code, `/N/B/gustav/tmp`. Rewrite this part of the program to
  1. obtain the value of `HOME` from the user's environment by calling function `getenv(3)`;
  2. check if directory `tmp` exists in `HOME` by calling function `stat(2)`;

**Hint** If the directory you're testing for does not exist function `stat(2)` will return `-1` signifying an error. You will then have to inspect the value of `errno(3)`, and in particular you should check if `errno=ENOENT`, which would mean that there is no file or directory that can be accessed by this path name.

**Hint** If `stat(2)` does not return an error, this means that the file does exist, but you still have to check if the file is a directory and if you will be able to write on it. On a successful return `stat(2)` will fill a structure of type `stat` that corresponds to the file name given to `stat(2)` as the first argument. The `st_mode` field of this structure can be used to test if the file in question is a directory by calling a macro `S_ISDIR`, which is defined on `/usr/include/sys/stat.h`. For example,

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
...
char *file_name;
struct stat buf;
...
if (stat(file_name, &buf) == 0) {
    if (S_ISDIR(buf.st_mode)) {
        /* The file in question is a directory. */
        ...
    } else {
        /* The file in question is not a directory. */
        ...
    }
} else {
```

```

        /* The file in question probably does not exist or it cannot
           be queried. Check errno. */
        ...
    }

```

3. Create this directory if it does not exist by calling function `mkdir(2)`.

These actions should be carried out by the master process only, whereas other processes should wait on a barrier call, `MPI_Barrier`, until the master process has completed its task.

### 5.2.5 Diffusion

In this section I will lay the foundation for an MPI version of a diffusion code based on Jacobi iterations. The idea is that a flat rectangular region, whose dimensions in this token example are going to be  $6 \times 8$ , will be divided amongst 12 processes, each handling a small square  $2 \times 2$ . But because it is a differential problem, apart from the  $2 \times 2$  data squares, the processes will also have to maintain additional data that is going to mirror data that corresponds to whatever the neighbouring processes have in their  $2 \times 2$  squares.

Diffusion is a very universal phenomenon that is encountered almost everywhere, beginning with microscopic world of quantum systems, and going all the way up to atmospheric circulation, pollution, and hurricanes. In the middle you encounter diffusion of nutrients through various membranes in cells or diffusion of oil through geological strata, or diffusion of heat through a combustion engine.

Mathematics of diffusion in all these systems is very similar and, at its simplest, it is described by the following partial differential equation:

$$\frac{\partial T(x, y, z)}{\partial t} = D \nabla^2 T(x, y, z)$$

where  $T(x, y, z)$  represents the diffusing quantity (e.g., it may stand for *temperature*),  $t$  stands for *time*,  $D$  is the *diffusion coefficient* and  $\nabla^2$  is the *Laplace operator*, which looks like this in Cartesian coordinates:

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}$$

Assuming that the problem is static, we get that  $\frac{\partial T(x, y, z)}{\partial t} = 0$  and then

$$\nabla^2 T(x, y, z) = 0 \tag{5.2}$$

too. This is called the *Laplace equation*.

We can simplify the problem further, by assuming that it is only 2-dimensional, so that

$$\nabla^2 T(x, y) = 0$$

This equation has the following discretized equivalent on a regular rectangular lattice with spacing  $h$  for any lattice node  $(x_0, y_0)$ :

$$T(x_0, y_0) = \frac{1}{4} (T(x_0 + h, y_0) + T(x_0 - h, y_0) + T(x_0, y_0 + h) + T(x_0, y_0 - h)) \quad (5.3)$$

In other words, the value of  $T$  at  $(x_0, y_0)$  is the arithmetic average of  $T$  over the neighbours nearest of  $(x_0, y_0)$  (we don't take the diagonal neighbours into account, because they are not the nearest).

This provides us with a simple method of solving equation (5.2) for a situation with fixed boundary conditions, i.e., we know  $T$  everywhere on the boundary of some 2-dimensional region and we want to find  $T$  everywhere *within* the region. The solution is to begin with an initial guess. Then use equation (5.3) to replace each  $T(x_0, y_0)$  with the average over the neighbours of  $(x_0, y_0)$ . We may have to repeat this process over and over, until we'll notice that  $T$  no longer changes from iteration to iteration. This is going to be our numerical solution of equation (5.2). This very simple method is due to a German mathematician Karl Gustav Jacob Jacobi, who was born on the 10th of December 1804 in Potsdam near Berlin, and who died on the 18th of February 1851 in Berlin.

In order to carry out Jacobi iterations on a parallel system, you need to divide the two-dimensional region, in the simplest case a rectangle, into, say, squares, allocating each square to a separate process. For nodes on the boundaries of the squares, we will need to know values that function  $T$  assumes on nodes that belong to other processes, so that we can perform the averaging.

This is where MPI communication comes in. We will have to use MPI in order to transfer these values. We only need to transfer values associated with the boundaries of the squares. If each square comprises  $N \times N$  nodes, then a process responsible for a given square will have to send  $4 \times N$  data items to other nodes for each iteration. The amount of data that needs to be communicated scales like  $N$ , whereas the amount of data that stays within the square and can be computed on without communication scales like  $N^2$ . Assuming that computation is a thousand times faster than communication, we get a "break-even" for  $N^2 = 1000 \times N$ , which yields  $N = 1000$ . For  $N$  much larger than 1000 the cost of communication will be negligible compared to the cost of computation. This means that the squares need to be very large. The faster the CPUs compared to the network, the more nodes you have to pack into each square to make the parallelization worthwhile.

The communication in this case can be facilitated further, if the processes responsible for adjacent squares can be made to run on "adjacent" processors. There are machines, such as, e.g., Cray X1, whose internal network topology lets programmers do just this. Such machines are exceptionally well suited to solving field problems, of which diffusion is the simplest.

In this section you will learn how MPI provides various constructs that are very helpful in capturing this type of problem. But we are not going to solve



the problem itself. Instead we will focus on the MPI mechanisms themselves and we will analyze a simple MPI demonstration program that explores these mechanisms.

In our simple example each process looks after an small  $2 \times 2$  integer matrix, which represents the “square” and the integers represent function  $T$ . We are going to surround each  $2 \times 2$  square with additional rows and columns around its boundary, into which we are going to transfer data from the adjacent squares. There will be a column on the left and on the right of the  $2 \times 2$  region to represent values obtained from adjacent squares on the left and on the right hand side, and rows above and below the  $2 \times 2$  region, to represent values obtained from adjacent squares above and below “our” square. So, in effect, each process will really have to work with a  $4 \times 4$  matrix, but only  $2 \times 2$  interior of this matrix is its own, and the periphery of the matrix will be populated with data obtained from other processes.

Within the example code we are going to:

1. arrange processes into a *virtual* 2-dimensional Cartesian process topology
2. orient processes within this new topology (so that they know who their neighbours are)
3. exchange data with neighbouring processes within the Cartesian topology so as to populate the peripheral rows and columns in the  $4 \times 4$  matrices.

At every stage the master process is going to collect data from all other processes and print it on standard output so as to show the state of the system at one glance.

In order to help you understand the whole procedure, I’m going to show you some of the output of the program first, then I’ll list the program for you, and then discuss it in more detail.

When the program starts, right after the processes have formed the new Cartesian topology, oriented themselves within it, but before they exchanged any data, their state looks as follows:

```

9 9 9 9 10 10 10 10 11 11 11 11
9 9 9 9 10 10 10 10 11 11 11 11
9 9 9 9 10 10 10 10 11 11 11 11
9 9 9 9 10 10 10 10 11 11 11 11

6 6 6 6 7 7 7 7 8 8 8 8
6 6 6 6 7 7 7 7 8 8 8 8
6 6 6 6 7 7 7 7 8 8 8 8
6 6 6 6 7 7 7 7 8 8 8 8

3 3 3 3 4 4 4 4 5 5 5 5
3 3 3 3 4 4 4 4 5 5 5 5
3 3 3 3 4 4 4 4 5 5 5 5
3 3 3 3 4 4 4 4 5 5 5 5

0 0 0 0 1 1 1 1 2 2 2 2
0 0 0 0 1 1 1 1 2 2 2 2
0 0 0 0 1 1 1 1 2 2 2 2
0 0 0 0 1 1 1 1 2 2 2 2
```

Every process initializes its own  $4 \times 4$  matrix to its own rank number. The matrices are displayed by the master process in a way that illustrates the topology of the whole system, i.e., we have a rectangular topology  $3 \times 4$ , process rank 0 is in the lower left corner, and its neighbours are process rank 1 on the right and process rank 3 above. Process rank 4 sits roughly in the middle and its neighbours are process rank 1 below, process rank 7 above, process rank 3 on the left, and process rank 5 on the right.

Now, the processes exchange the content of their inner rows with their neighbours, i.e., process rank 4 sends the content of its row 3 (counted from the bottom) to process number 7, which puts it in its own row 1 (counted from the bottom), and, at the same time receives the content of row 2 from process rank 7, and places it in its own row 4.

So that after this exchange operation has taken place, the matrices look as follows:

```

9 9 9 9 10 10 10 10 11 11 11 11
9 9 9 9 10 10 10 10 11 11 11 11
9 9 9 9 10 10 10 10 11 11 11 11
6 6 6 6 7 7 7 7 8 8 8 8

9 9 9 9 10 10 10 10 11 11 11 11
6 6 6 6 7 7 7 7 8 8 8 8
6 6 6 6 7 7 7 7 8 8 8 8
3 3 3 3 4 4 4 4 5 5 5 5

6 6 6 6 7 7 7 7 8 8 8 8
3 3 3 3 4 4 4 4 5 5 5 5
3 3 3 3 4 4 4 4 5 5 5 5
0 0 0 0 1 1 1 1 2 2 2 2

3 3 3 3 4 4 4 4 5 5 5 5
0 0 0 0 1 1 1 1 2 2 2 2
0 0 0 0 1 1 1 1 2 2 2 2
0 0 0 0 1 1 1 1 2 2 2 2

```

Now we have to repeat this operation, but this time exchanging columns between neighbours. And so process rank 4 will send its column 2 (counted from the left) to process rank 3, which is going to place it in its own column 4 (counted from the left), and, at the same time process rank 3 is going to send its own column 3 to process rank 4, which is going to place it in its own column 1.

And after all this is over, the matrices are going to look as follows:

```

9 9 9 10 9 10 10 11 10 11 11 11
9 9 9 10 9 10 10 11 10 11 11 11
9 9 9 10 9 10 10 11 10 11 11 11
6 6 6 7 6 7 7 8 7 8 8 8

9 9 9 10 9 10 10 11 10 11 11 11
6 6 6 7 6 7 7 8 7 8 8 8
6 6 6 7 6 7 7 8 7 8 8 8
3 3 3 4 3 4 4 5 4 5 5 5

6 6 6 7 6 7 7 8 7 8 8 8
3 3 3 4 3 4 4 5 4 5 5 5

```

```

3 3 3 4 3 4 4 5 4 5 5 5
0 0 0 1 0 1 1 2 1 2 2 2

3 3 3 4 3 4 4 5 4 5 5 5
0 0 0 1 0 1 1 2 1 2 2 2
0 0 0 1 0 1 1 2 1 2 2 2
0 0 0 1 0 1 1 2 1 2 2 2

```

Observe that now not only does every process know what data is harboured by its neighbours on the left, on the right, above, and below, but they even know the data that somehow made it diagonally, i.e., from the upper left, upper right, lower left, and lower right directions - but the latter is not going to last, because that data came from the peripheral rows and columns in the adjacent squares, so it is not truly representative of the processes' internal states.

Now every process can perform its own Jacobi iteration within its own little patch for a diffusion problem, and set new values to its own data, while keeping the mirrored data unchanged.

Before the next iteration can commence, the data has to be exchanged between the neighbours again, in order to refresh the peripheries of the squares.

So now let us have a look at the code:

### The Code

Here is the code itself. Then I show how it's made, installed and how it's run. You have to run it on 12 processes.

```

/*
 * %Id: cart.c,v 1.1 2003/10/06 00:56:48 gustav Exp %
 *
 * %Log: cart.c,v %
 * Revision 1.1 2003/10/06 00:56:48 gustav
 * Initial revision
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define FALSE 0
#define TRUE 1
#define MASTER_RANK 0

#define UPDOWN 0
#define SIDEWAYS 1
#define RIGHT 1
#define UP 1

#define PROCESS_DIMENSIONS 2

#define PROCESS_ROWS 4
#define ROWS 4
#define DISPLAY_ROWS 16 /* must be PROCESS_ROWS * ROWS */

```

```

#define PROCESS_COLUMNS      3
#define COLUMNS              4
#define DISPLAY_COLUMNS      12    /* must be PROCESS_COLUMNS * COLUMNS */

int main ( int argc, char **argv )
{
    int pool_size, my_rank, destination, source;
    MPI_Status status;
    char char_buffer[BUFSIZ];
    int i_am_the_master = FALSE;

    int divisions[PROCESS_DIMENSIONS] = {PROCESS_ROWS, PROCESS_COLUMNS};
    int periods[PROCESS_DIMENSIONS] = {0, 0};
    int reorder = 1;
    MPI_Comm cartesian_communicator;
    int my_cartesian_rank, my_coordinates[PROCESS_DIMENSIONS];
    int left_neighbour, right_neighbour, bottom_neighbour, top_neighbour;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &pool_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank == MASTER_RANK) i_am_the_master = TRUE;

    MPI_Cart_create ( MPI_COMM_WORLD, PROCESS_DIMENSIONS, divisions,
                     periods, reorder, &cartesian_communicator );

    if (cartesian_communicator != MPI_COMM_NULL) {

        int matrix [ROWS] [COLUMNS];
        int i, j;
        MPI_Datatype column_type;

        MPI_Comm_rank ( cartesian_communicator, &my_cartesian_rank );
        MPI_Cart_coords ( cartesian_communicator, my_cartesian_rank,
                         PROCESS_DIMENSIONS, my_coordinates );
        MPI_Cart_shift ( cartesian_communicator, SIDEWAYS, RIGHT,
                        &left_neighbour, &right_neighbour );
        MPI_Cart_shift ( cartesian_communicator, UPDOWN, UP,
                        &bottom_neighbour, &top_neighbour );

        if (! i_am_the_master ) {
            sprintf(char_buffer, "process %2d, cartesian %2d, \
coords (%2d,%2d), left %2d, right %2d, top %2d, bottom %2d",
                  my_rank, my_cartesian_rank, my_coordinates[0],
                  my_coordinates[1], left_neighbour, right_neighbour,
                  top_neighbour, bottom_neighbour);
            MPI_Send(char_buffer, strlen(char_buffer) + 1, MPI_CHAR,
                     MASTER_RANK, 3003, MPI_COMM_WORLD);
        }
        else {

            int number_of_c_procs, count;

            number_of_c_procs = divisions[0] * divisions[1];
            for (count = 0; count < number_of_c_procs - 1; count++) {

```

```

MPI_Recv(char_buffer, BUFSIZ, MPI_CHAR, MPI_ANY_SOURCE, 3003,
MPI_COMM_WORLD, &status);
printf ("%s\n", char_buffer);
}
printf( "process %2d, cartesian %2d, \
coords (%2d,%2d), left %2d, right %2d, top %2d, bottom %2d\n",
my_rank, my_cartesian_rank, my_coordinates[0],
my_coordinates[1], left_neighbour, right_neighbour,
top_neighbour, bottom_neighbour);
}

for ( i = 0; i < ROWS; i++ ) {
for ( j = 0; j < COLUMNS; j++ ) {
matrix [i][j] = my_cartesian_rank;
}
}

if (my_cartesian_rank != MASTER_RANK )
MPI_Send ( matrix, COLUMNS * ROWS, MPI_INT, MASTER_RANK, 3003,
cartesian_communicator );
else
collect_matrices ( cartesian_communicator, my_cartesian_rank,
matrix, 3003 );

MPI_Sendrecv ( &matrix[ROWS - 2][0], COLUMNS, MPI_INT,
top_neighbour, 4004,
&matrix[0][0], COLUMNS, MPI_INT, bottom_neighbour,
4004,
cartesian_communicator, &status );

MPI_Sendrecv ( &matrix[1][0], COLUMNS, MPI_INT, bottom_neighbour,
5005,
&matrix[ROWS - 1][0], COLUMNS, MPI_INT,
top_neighbour, 5005,
cartesian_communicator, &status );

if (my_cartesian_rank != MASTER_RANK )
MPI_Send ( matrix, COLUMNS * ROWS, MPI_INT, MASTER_RANK, 6006,
cartesian_communicator );
else
collect_matrices ( cartesian_communicator, my_cartesian_rank,
matrix, 6006 );

MPI_Type_vector (ROWS, 1, COLUMNS, MPI_INT, &column_type);
MPI_Type_commit (&column_type);

MPI_Sendrecv ( &matrix[0][1], 1, column_type, left_neighbour, 7007,
&matrix[0][COLUMNS - 1], 1, column_type,
right_neighbour, 7007,
cartesian_communicator, &status );

MPI_Sendrecv ( &matrix[0][COLUMNS - 2], 1, column_type,
right_neighbour, 8008,
&matrix[0][0], 1, column_type, left_neighbour, 8008,
cartesian_communicator, &status );

if (my_cartesian_rank != MASTER_RANK )

```

```

        MPI_Send ( matrix, COLUMNS * ROWS, MPI_INT, MASTER_RANK, 9009,
        cartesian_communicator );
        else
            collect_matrices ( cartesian_communicator, my_cartesian_rank,
            matrix, 9009 );
    }

    MPI_Finalize ();
    exit(0);
}

int print_array (int array [DISPLAY_ROWS] [DISPLAY_COLUMNS],
    int vertical_break,
    int horizontal_break)
{
    int k, l;

    printf ("\n");
    for (k = DISPLAY_ROWS - 1; k >= 0; k -- ) {
        for (l = 0; l < DISPLAY_COLUMNS; l ++ ) {
            if (l % horizontal_break == 0) printf (" ");
            printf ("%2d ", array [k][l] );
        }
        printf ( "\n" );
        if (k % vertical_break == 0) printf ( "\n" );
    }
}

int collect_matrices (MPI_Comm cartesian_communicator,
    int my_cartesian_rank,
    int matrix[ROWS][COLUMNS],
    int tag)
{
    int coordinates[PROCESS_DIMENSIONS];
    int client_matrix[ROWS][COLUMNS];
    int display[DISPLAY_ROWS][DISPLAY_COLUMNS];
    int i, j, k, l, source;
    MPI_Status status;

    for ( i = PROCESS_ROWS - 1; i >= 0; i -- ) {
        for ( j = 0; j < PROCESS_COLUMNS; j ++ ) {
            coordinates[0] = i;
            coordinates[1] = j;
            MPI_Cart_rank ( cartesian_communicator, coordinates,
            &source );
            if (source != my_cartesian_rank) {
                MPI_Recv ( client_matrix, BUFSIZ, MPI_INT, source, tag,
                cartesian_communicator, &status );
                for ( k = ROWS - 1; k >= 0; k -- ) {
                    for ( l = 0; l < COLUMNS; l ++ ) {
                        display [i * ROWS + k] [j * COLUMNS + l] =
                        client_matrix[k][l];
                    }
                }
            }
            else {
                for ( k = ROWS - 1; k >= 0; k -- ) {

```

```

    for ( l = 0; l < COLUMNS; l ++ ) {
        display [i * ROWS + k] [j * COLUMNS + l]
            = matrix[k][l];
    }
}
    }
}
}

print_array (display, ROWS, COLUMNS);
}

```

The Makefile for this program looks the same as the Makefile for the previous program. It simply calls `mpicc` to compile and link it, then `install` to install it:

```

gustav@bh1 $ pwd
/N/B/gustav/src/I590/cart
gustav@bh1 $ make install
co RCS/Makefile,v Makefile
RCS/Makefile,v --> Makefile
revision 1.1
done
co RCS/cart.c,v cart.c
RCS/cart.c,v --> cart.c
revision 1.1
done
mpicc -o cart cart.c
install cart /N/B/gustav/bin
gustav@bh1 $ make clobber
rm -f cart.o cart
rcsclean
rm -f Makefile
rm -f cart.c
gustav@bh1 $

```

As I have mentioned above, this program *must* be run on 12 processes. Here's how it's done:

```

gustav@bh1 $ mpdboot
gustav@bh1 $ mpdtrace | wc
    13    13    64
gustav@bh1 $ mpdtrace
bh1
bc29
bc30
bc31
bc32
bc33
bc36
bc35
bc34
bc40
bc38
bc37
bc39
gustav@bh1 $ mpiexec -n 12 cart
process 1, cartesian 1, coords ( 0, 1), left 0, right 2, top 4, bottom -1

```

```

process 11, cartesian 11, coords ( 3, 2), left 10, right -1, top -1, bottom 8
process 10, cartesian 10, coords ( 3, 1), left 9, right 11, top -1, bottom 7
process 8, cartesian 8, coords ( 2, 2), left 7, right -1, top 11, bottom 5
process 9, cartesian 9, coords ( 3, 0), left -1, right 10, top -1, bottom 6
process 7, cartesian 7, coords ( 2, 1), left 6, right 8, top 10, bottom 4
process 6, cartesian 6, coords ( 2, 0), left -1, right 7, top 9, bottom 3
process 5, cartesian 5, coords ( 1, 2), left 4, right -1, top 8, bottom 2
process 4, cartesian 4, coords ( 1, 1), left 3, right 5, top 7, bottom 1
process 3, cartesian 3, coords ( 1, 0), left -1, right 4, top 6, bottom 0
process 2, cartesian 2, coords ( 0, 2), left 1, right -1, top 5, bottom -1
process 0, cartesian 0, coords ( 0, 0), left -1, right 1, top 3, bottom -1

```

```

9 9 9 9 10 10 10 10 11 11 11 11
9 9 9 9 10 10 10 10 11 11 11 11
9 9 9 9 10 10 10 10 11 11 11 11
9 9 9 9 10 10 10 10 11 11 11 11

```

```

6 6 6 6 7 7 7 7 8 8 8 8
6 6 6 6 7 7 7 7 8 8 8 8
6 6 6 6 7 7 7 7 8 8 8 8
6 6 6 6 7 7 7 7 8 8 8 8

```

```

3 3 3 3 4 4 4 4 5 5 5 5
3 3 3 3 4 4 4 4 5 5 5 5
3 3 3 3 4 4 4 4 5 5 5 5
3 3 3 3 4 4 4 4 5 5 5 5

```

```

0 0 0 0 1 1 1 1 2 2 2 2
0 0 0 0 1 1 1 1 2 2 2 2
0 0 0 0 1 1 1 1 2 2 2 2
0 0 0 0 1 1 1 1 2 2 2 2

```

```

9 9 9 9 10 10 10 10 11 11 11 11
9 9 9 9 10 10 10 10 11 11 11 11
9 9 9 9 10 10 10 10 11 11 11 11
6 6 6 6 7 7 7 7 8 8 8 8

```

```

9 9 9 9 10 10 10 10 11 11 11 11
6 6 6 6 7 7 7 7 8 8 8 8
6 6 6 6 7 7 7 7 8 8 8 8
3 3 3 3 4 4 4 4 5 5 5 5

```

```

6 6 6 6 7 7 7 7 8 8 8 8
3 3 3 3 4 4 4 4 5 5 5 5
3 3 3 3 4 4 4 4 5 5 5 5
0 0 0 0 1 1 1 1 2 2 2 2

```

```

3 3 3 3 4 4 4 4 5 5 5 5
0 0 0 0 1 1 1 1 2 2 2 2
0 0 0 0 1 1 1 1 2 2 2 2
0 0 0 0 1 1 1 1 2 2 2 2

```

```

9 9 9 10 9 10 10 11 10 11 11 11
9 9 9 10 9 10 10 11 10 11 11 11
9 9 9 10 9 10 10 11 10 11 11 11

```



```

6 6 6 7 6 7 7 8 7 8 8 8
9 9 9 10 9 10 10 11 10 11 11 11
6 6 6 7 6 7 7 8 7 8 8 8
6 6 6 7 6 7 7 8 7 8 8 8
3 3 3 4 3 4 4 5 4 5 5 5

6 6 6 7 6 7 7 8 7 8 8 8
3 3 3 4 3 4 4 5 4 5 5 5
3 3 3 4 3 4 4 5 4 5 5 5
0 0 0 1 0 1 1 2 1 2 2 2

3 3 3 4 3 4 4 5 4 5 5 5
0 0 0 1 0 1 1 2 1 2 2 2
0 0 0 1 0 1 1 2 1 2 2 2
0 0 0 1 0 1 1 2 1 2 2 2

```

```

gustav@bh1 $ mpdallexit
gustav@bh1 $

```

### The Discussion

This program contains numerous new elements and MPI function calls, so this section may be a little heavy. But it also illustrates where MPI differs from earlier, much simpler, but also much more primitive message passing systems. MPI is very powerful. It is not just *send* and *receive*.

The code begins innocently enough like all other MPI codes that we have seen so far: MPI itself is initialised, then every process finds out about the size of the process pool, and its own rank within that pool.

```

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &pool_size);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

```

And then, as usual, one of the processes assumes the role of the master:

```

if (my_rank == MASTER_RANK) i_am_the_master = TRUE;

```

But the next operation is entirely new:

```

MPI_Cart_create ( MPI_COMM_WORLD, PROCESS_DIMENSIONS, divisions,
                  periods, reorder, &cartesian_communicator );

```

What happens here is as follows. The processes create a new communicator, which, unlike the `MPI_COMM_WORLD`, has a special topology imposed on it. The new communicator, the place for which is passed to `MPI_Cart_create` as the last argument, is formed out of the old one, which is passed to `MPI_Cart_create` in the first argument. The four parameters passed in the middle: `PROCESS_DIMENSIONS`, `divisions`, `periods`, and `reorder`, specify the topology of the new communicator. When we say *topology* it means that there is some sort of a connectivity between the processes within this new communicator. In this case they will acquire left, right, bottom, and top *neighbours*.

The value of `PROCESS_DIMENSIONS` is 2, which means that the processes are to be organised into a 2 dimensional grid. `divisions` is going to be an array of rank 1, whose entries specify lengths of the process grid in those two dimensions, in this case:

```

#define PROCESS_ROWS      4
...
#define PROCESS_COLUMNS   3
...
int divisions[PROCESS_DIMENSIONS] = {PROCESS_ROWS, PROCESS_COLUMNS};

```

that is, the processes are going to be organised into a rectangular grid with 4 rows and 3 columns.

The parameter `periods` specifies if the topology is going to be a wrap-around topology or an open ended topology, i.e., if the processes that are on the borders of the region should have as their over the border neighbours the guys on the other side, or nobody. In this case we simply say that

```
int periods[PROCESS_DIMENSIONS] = {0, 0};
```

which means that we don't want any wrapping.

The last parameter in this group, `reorder`, specifies if processes should be reordered for efficiency. This implies that processes may be moved around between processors, or just renumbered, so as to utilize better any hardware architecture that a given machine may have. Examples of such machines are Cray T3E, Cray X1, Fujitsu AP-3000, and NEC Cenju-3.

If you have asked for a  $4 \times 3$  Cartesian communicator and started with, say, 15 processes, then 3 of those 15 processes would have to be rejected from the new communicator, because there is only going to be enough space in it for 12 processes. The rejected processes will find that after `MPI_Cart_create` returns, the returned value of `cartesian_communicator` is `MPI_COMM_NULL`. What this means is that those processes didn't get the job.

They may hang about, if they choose, or they may go right for `MPI_Finalize` – this is up to the programmer. You may create more than one communicator with various properties, so that the processes that didn't make it into the `cartesian_communicator` will get jobs elsewhere.

For this reason the remainder of the program is in the form of one large if statement:

```

if (cartesian_communicator != MPI_COMM_NULL) {
    blah... blah... blah...
}

MPI_Finalize ();
exit (0);
}

```

Now, assuming that you did get a job, as a process, with the `cartesian_communicator`, the first thing that you do is to find your new rank number within that new communicator:

```
MPI_Comm_rank ( cartesian_communicator, &my_cartesian_rank );
```

And the reason for this is that your rank number in the `cartesian_communicator` may be entirely different from your old number in the `MPI_COMM_WORLD` communicator. They're like a Tax File Number in Australia and a Social Security Number in the US.

But now, since the `cartesian_communicator` is a communicator with a topology, you can make inquiries about your neighbourhood:

```

MPI_Cart_coords ( cartesian_communicator, my_cartesian_rank,
                 PROCESS_DIMENSIONS, my_coordinates );
MPI_Cart_shift ( cartesian_communicator, SIDEWAYS, RIGHT,
                 &left_neighbour, &right_neighbour );
MPI_Cart_shift ( cartesian_communicator, UPDOWN, UP,
                 &bottom_neighbour, &top_neighbour );

```

The first call to `MPI_Cart_coords` returns your Cartesian coordinates in an array `my_coordinates` of length `PROCESS_DIMENSIONS`. Naturally you have to tell `MPI_Cart_coords` who you are, in order to obtain that information. You have to pass on your new Cartesian rank number in the second slot of the function.

The following two calls to `MPI_Cart_shift` tell you about the rank numbers of your left and right neighbours and of your bottom and top neighbours. So the function `MPI_Cart_shift` doesn't really shift anything. It's more like looking up who your neighbours are. But the designers of MPI thought of it in terms of shifting rank numbers of your neighbours from the left and from the right and from above and from below into those containers, which you call `left_neighbour`, `right_neighbour`, `bottom_neighbour`, and `top_neighbour`.

This stuff is worth displaying, so what we do now is to pack all that information into a message and send it to the master process.

```

if (! i_am_the_master ) {
    sprintf(char_buffer, "process %2d, cartesian %2d, \
coords (%2d,%2d), left %2d, right %2d, top %2d, bottom %2d",
           my_rank, my_cartesian_rank, my_coordinates[0],
           my_coordinates[1], left_neighbour, right_neighbour,
           top_neighbour, bottom_neighbour);
    MPI_Send(char_buffer, strlen(char_buffer) + 1, MPI_CHAR,
             MASTER_RANK, 3003, MPI_COMM_WORLD);
}
else {

    int number_of_c_procs, count;

    number_of_c_procs = divisions[0] * divisions[1];
    for (count = 0; count < number_of_c_procs - 1; count++) {
        MPI_Recv(char_buffer, BUFSIZ, MPI_CHAR, MPI_ANY_SOURCE, 3003,
                MPI_COMM_WORLD, &status);
        printf ("%s\n", char_buffer);
    }
    printf( "process %2d, cartesian %2d, \
coords (%2d,%2d), left %2d, right %2d, top %2d, bottom %2d\n",
           my_rank, my_cartesian_rank, my_coordinates[0],
           my_coordinates[1], left_neighbour, right_neighbour,
           top_neighbour, bottom_neighbour);
}

```

Observe that this communication takes place within the `MPI_COMM_WORLD`, not within the `cartesian_communicator`. The old communicator didn't go away, when the new one was created, so it can be still used for various things.

Observe a subtle bug here: we are assuming that the master process is still going to hang around! In other words that the master process was included into the `cartesian_communicator`.

What will happen if it is not included?

In our case we are going to run this job requesting exactly the right number of processes, so the master process is guaranteed to be included into the `cartesian_communicator`. Having said that you may consider modifying the logic of the program so as to get rid of the bug.

Here is what the output looks like, once the master process gets down to it:

```
process 1, cartesian 1, coords ( 0, 1), left 0, right 2, top 4, bottom -1
process 10, cartesian 10, coords ( 3, 1), left 9, right 11, top -1, bottom 7
process 11, cartesian 11, coords ( 3, 2), left 10, right -1, top -1, bottom 8
process 9, cartesian 9, coords ( 3, 0), left -1, right 10, top -1, bottom 6
process 8, cartesian 8, coords ( 2, 2), left 7, right -1, top 11, bottom 5
process 7, cartesian 7, coords ( 2, 1), left 6, right 8, top 10, bottom 4
process 6, cartesian 6, coords ( 2, 0), left -1, right 7, top 9, bottom 3
process 5, cartesian 5, coords ( 1, 2), left 4, right -1, top 8, bottom 2
process 2, cartesian 2, coords ( 0, 2), left 1, right -1, top 5, bottom -1
process 3, cartesian 3, coords ( 1, 0), left -1, right 4, top 6, bottom 0
process 4, cartesian 4, coords ( 1, 1), left 3, right 5, top 7, bottom 1
process 0, cartesian 0, coords ( 0, 0), left -1, right 1, top 3, bottom -1
```

Observe that the cartesian rank numbers here are the same as the world rank numbers. But you must not count on it. It just happens to be so here, on the AVIDD, for this particular program, and under this particular version of MPICH2. This is not an MPI requirement.

What happens next is this simple piece of code:

```
for ( i = 0; i < ROWS; i++ ) {
    for ( j = 0; j < COLUMNS; j++ ) {
        matrix [i][j] = my_cartesian_rank;
    }
}
```

Every process simply initializes its own little matrix with its own cartesian rank number. The whole matrix is initialized, including the parts that are going to be dedicated to mirroring, along the borders of the matrix.

Once the matrices have been initialized, the processes send them to the master process, who dilligently collects them and displays on standard output:

```
if (my_cartesian_rank != MASTER_RANK )
    MPI_Send ( matrix, COLUMNS * ROWS, MPI_INT, MASTER_RANK, 6006,
              cartesian_communicator );
else
    collect_matrices ( cartesian_communicator, my_cartesian_rank,
                     matrix, 6006 );
```

Function `collect_matrices` hides the tedium of collecting, arranging and displaying data received from other nodes.

This time the communication takes place entirely within the `cartesian_communicator`, and the process that collects all matrices does not have to be the same process as the master process in the `MPI_COMM_WORLD` communicator.

Now the processes in the `cartesian_communicator` have to exchange information with their neighbours:

```
MPI_Sendrecv ( &matrix[ROWS - 2][0], COLUMNS, MPI_INT,
              top_neighbour, 4004,
              &matrix[0][0], COLUMNS, MPI_INT, bottom_neighbour,
```

```

        4004,
        cartesian_communicator, &status );

MPI_Sendrecv ( &matrix[1][0], COLUMNS, MPI_INT, bottom_neighbour,
              5005,
              &matrix[ROWS - 1][0], COLUMNS, MPI_INT,
              top_neighbour, 5005,
              cartesian_communicator, &status );

```

While sending this stuff up and down with the `MPI_Sendrecv` operation, we're making use of the fact that matrices in C are stored in the row-major fashion, i.e., the other way to Fortran. The first statement sends the whole second row from the top (I have numbered these matrices upside down, so as to stick to the usual  $x \times y$  convention) to the top neighbour. The send buffer begins at `&matrix[ROWS - 2][0]` and is `COLUMNS` items long. The items are of type `MPI_INT` and the corresponding message is going to have tag 4004.

At the same time, every process receives into its bottom row data sent by its bottom neighbour. The beginning of the receive buffer is `&matrix[0][0]`, the buffer contains items of type `MPI_INT`, and the number of those items is `COLUMNS`.

Of course this communication must take place within the `cartesian_communicator`, because it is only within this communicator that the notion of a bottom and top neighbour makes sense.

The second `MPI_Sendrecv` operation does the same, but in the opposite direction, i.e., the data is now sent down, whereas previously it went up.

Having done that we display the state of the whole system again:

```

if (my_cartesian_rank != MASTER_RANK )
    MPI_Send ( matrix, COLUMNS * ROWS, MPI_INT, MASTER_RANK, 6006,
              cartesian_communicator );
else
    collect_matrices ( cartesian_communicator, my_cartesian_rank,
                      matrix, 6006 );

```

The last part of the code is a little tricky. Here we have to exchange columns between neighbours, but columns are not stored in C contiguously.

MPI provides a very powerful apparatus for situations like this one.

First we define a stridden data type:

```

MPI_Type_vector (ROWS, 1, COLUMNS, MPI_INT, &column_type);
MPI_Type_commit (&column_type);

```

Function `MPI_Type_vector` constructs this special stridden data type as follows: the data type comprises `ROWS` items of type `MPI_INT`. The items are collected by picking up 1 item every `COLUMNS` steps from a contiguous storage. The description of this peculiar data type is placed into `column_type`. Once the type has been defined it must be committed by calling function `MPI_Type_commit`. Here we give a parallel machine an opportunity to adjust its hardware or software logic in order to prepare for manipulating this new data type.

Now we can exchange columns between neighbours in a much the same way we did rows before:

```

MPI_Sendrecv ( &matrix[0][1], 1, column_type, left_neighbour, 7007,
               &matrix[0][COLUMNS - 1], 1, column_type,
               right_neighbour, 7007,
               cartesian_communicator, &status );

MPI_Sendrecv ( &matrix[0][COLUMNS - 2], 1, column_type,
               right_neighbour, 8008,
               &matrix[0][0], 1, column_type, left_neighbour, 8008,
               cartesian_communicator, &status );

```

The first operation sends one item of type `column_type`, stored at `&matrix[0][1]` to the `left_neighbour`. The corresponding message has tag 7007. At the same time another single item of `column_type` is received from the `right_neighbour` and placed in `&matrix[0][COLUMNS - 1]`. Again, observe that we are not sending the border columns: we are sending internal columns, and we're receiving them into the border columns. The second `MPI_Sendrecv` call does the same in the opposite direction, i.e., from left to right.

After the data has been exchanged, the state of the system is displayed again by calling:

```

if (my_cartesian_rank != MASTER_RANK )
    MPI_Send ( matrix, COLUMNS * ROWS, MPI_INT, MASTER_RANK, 9009,
              cartesian_communicator );
else
    collect_matrices ( cartesian_communicator, my_cartesian_rank,
                      matrix, 9009 );

```

So, this is what the program does.

In spite of MPI's powerful auxiliary functions this is still messy, clumsy and, worst of all, error prone, compared to data parallel environments such as High Performance Fortran that do it all automatically. IBM and PGI HPF compilers, as a matter of fact, compile your HPF program to an MPI code that does very much what this simple example illustrates. To use HPF in place of MPI, wherever applicable, will save you a lot of hard work.

HPF has recently been installed on the AVIDD cluster and it lives in `/opt/pgi/linux86/bin`.

## Exercises

- 1 The program discussed in section 5.2.5 initializes all entries in the  $4 \times 4$  matrices managed by the worker processes to the rank number of the processes. But it is only the  $2 \times 2$  interior of these matrices that is really managed by the processes, whereas the boundary rows and columns are filled by obtaining data from the neighbouring processes. Rewrite the initialization procedure to initialize only the  $2 \times 2$  interior to the process rank number and the remainder of each matrix should be initialized to *minus* the rank number.

**Hint** You will have to revise the function that displays the matrices, because you will need more space for the minus sign.

- 2** The program transfers whole  $1 \times 4$  rows or  $4 \times 1$  columns between processes. But it is only the  $1 \times 2$  or  $2 \times 1$  interior of the columns and rows that contains the valid data. Rewrite the program to transfer only what is really needed between the neighbouring processes.
- 3** Revise the program to allow for an arbitrary number of start-up processes. Fix the bug that does not account for the fact that some processes may not make it into the newly formed Cartesian communicator and that the master process may be amongst them.

### 5.2.6 Interacting Particles

In this section we are going to take a shot at the problem of interacting particles. A typical application of this problem is to the motion of charged plasma particles that interact electrostatically with each other. Plasmas are actually even more complex, because accelerated particles radiate and some of this radiation may be captured by other particles, and this adds enormous complexity to the models. Other applications include molecular dynamics and, right at the other end of the spectrum, galactic dynamics.

As far as MPI is concerned, we are going to take another look at defining MPI types and at collective communications. We are going to arrange all particles into an array and the way that this array is going to be partitioned between processes of the computational pool is very reminiscent of how MPI-IO partitions files. So this section may be thought of as a preparation to MPI-IO as well.

Let me begin by reminding you about the Coulomb law. If you have two electrically charged particles, with charges  $q_1$  and  $q_2$ , then the force with which particle 2 acts on particle 1 is given by

$$\begin{aligned} \mathbf{F}_{12} &= -\frac{1}{4\pi\epsilon_0} \frac{q_1 q_2}{r_{12}^2} \frac{\mathbf{r}_{12}}{r_{12}} \quad \text{where} & (5.4) \\ \mathbf{r}_{12} &= \mathbf{r}_2 - \mathbf{r}_1 \quad \text{and} \\ r_{12} &= \sqrt{(r_{12}^x)^2 + (r_{12}^y)^2 + (r_{12}^z)^2} \end{aligned}$$

where  $\epsilon_0$  is the dielectric constant of vacuum and vector  $\mathbf{r}_{12}$  points from particle 1 to particle 2 (this is easy to see – simply assume that particle 1 sits in the middle of the world so that  $\mathbf{r}_1 = 0$ ). If the particle charges are of the opposite sign, then  $q_1 q_2$  is negative. This cancels the minus in front of  $1/(4\pi\epsilon_0)$  and particle 1 ends up being accelerated towards particle 2. The particles attract each other. If the particles have charges of the same sign,  $q_1 q_2 > 0$  and then particle 1 is accelerated in the direction that is opposite to where particle 2 is. The particles repel each other.

The force with which particle 1 acts on particle 2 is the opposite to  $\mathbf{F}_{12}$ , i.e.,

$$\mathbf{F}_{21} = -\mathbf{F}_{12}$$

Sometimes equation (5.4) is printed without the minus in front of  $1/(4\pi\epsilon_0)$ . Whether the minus should or should not be there depends on the definitions of  $\mathbf{F}_{12}$  and  $\mathbf{r}_{12}$ . If  $\mathbf{r}_{12}$  is the vector that points from particle 1 to particle 2, i.e.,  $\mathbf{r}_2 - \mathbf{r}_1$ , and if  $\mathbf{F}_{12}$  is the force with which particle 2 acts on particle 1, then the minus should be there. Physics is one of these somewhat troubling disciplines, where you need to understand what happens and where you need to be very precise about what is what, otherwise it's easy to get confused.

Consider  $N$  particles of various charges scattered at random in some region of space. In order to evaluate the total force that acts on particle number  $i$  we have to evaluate the sum:

$$\mathbf{F}_i = -\frac{1}{4\pi\epsilon_0} \sum_{\substack{j=N \\ (j \neq i)}} \frac{q_i q_j}{r_{ij}^2} \frac{\mathbf{r}_{ij}}{r_{ij}} \quad (5.5)$$

This sum has  $N - 1$  terms in it, because we assume that particle  $i$  does not act on itself. In order to evaluate forces that act on all  $N$  particles we have to evaluate  $N(N - 1)$  such terms. We say that this computation is going to scale like  $\mathcal{O}(N^2)$ . This is going to turn very expensive if  $N$  gets very large.

We are going to distribute the computation over  $n$  processes by making each process evaluate the force for  $N/n$  particles. Each process is therefore going to evaluate  $N/n(N - 1)$  terms, which may result in considerable speed up of the computation for a sufficiently large value of  $n$ . In particular, if we could make  $n = N$ , the computation would scale as  $\mathcal{O}(N)$  instead of  $\mathcal{O}(N^2)$ . This is how nature goes about “parallelizing” this problem. You can think of each particle as being a separate processor that calculates its own evolution.

Within this example program we are going to push all  $N$  particles for 20 short time steps using a very simplistic numerical procedure that corresponds to the following equations of motion for particle  $i$ :

$$m_i \frac{d\mathbf{v}_i}{dt} = -\frac{q_i}{4\pi\epsilon_0} \sum_{\substack{j=N \\ (j \neq i)}} \frac{q_j}{r_{ij}^2} \frac{\mathbf{r}_{ij}}{r_{ij}}$$

$$\frac{d\mathbf{r}_i}{dt} = \mathbf{v}_i$$

We can always choose to use such time units in our computation that

$$\frac{1}{4\pi\epsilon_0} = 1$$

We are going to discretize the equations of motion by replacing  $dt$  with a final time interval  $\Delta t$ , so that:

$$\mathbf{v}_i(t + \Delta t) = \mathbf{v}_i(t) - \frac{q_i \Delta t}{m_i} \sum_{\substack{j=N \\ (j \neq i)}} \frac{q_j}{r_{ij}^2(t)} \frac{\mathbf{r}_{ij}(t)}{r_{ij}(t)}$$

$$\mathbf{r}_i(t + \Delta t) = \mathbf{r}_i(t) + \frac{\mathbf{v}_i(t) + \mathbf{v}_i(t + \Delta t)}{2} \Delta t$$



## The Code

So here is the code in full. Towards the end of this section I show you how the code is compiled and how it runs. The code is discussed in the next section.

```

/*
 * %Id: particles.c,v 1.7 2003/10/12 01:38:12 gustav Exp %
 *
 * %Log: particles.c,v %
 * Revision 1.7 2003/10/12 01:38:12 gustav
 * Fixed the position advance bug.
 *
 * Revision 1.6 2003/10/10 23:15:32 gustav
 * Made the master process write the iteration number.
 * Increased the number of iterations to 20.
 *
 * Revision 1.5 2003/10/10 23:08:59 gustav
 * Modified the whole example, making it into a true computation.
 * Changed the structure to incorporate velocity, accelerations and charge.
 * Changed number of particles back to 10000
 *
 * Revision 1.4 2003/10/10 21:10:04 gustav
 * Increased the number of particles to 100,000
 *
 * Revision 1.3 2003/10/10 19:15:23 gustav
 * More niceties.
 *
 * Revision 1.2 2003/10/10 19:07:26 gustav
 * Increased the number of particles to 10,000 added some niceties.
 *
 * Revision 1.1 2003/10/10 18:55:23 gustav
 * Initial revision
 *
 */

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <mpi.h>

#define FALSE          0
#define TRUE           1
#define MASTER_RANK   0

#define MAX_PARTICLES 10000
#define MAX_PROCS     128
#define EPSILON        1.0E-10
#define DT              0.01
#define N_OF_ITERATIONS 20

int main ( int argc, char **argv )
{
    int pool_size, my_rank;
    int i_am_the_master = FALSE;
    extern double drand48();
    extern void srand48();

```

```

typedef struct {
    double x, y, z, vx, vy, vz, ax, ay, az, mass, charge;
} Particle;

Particle particles[MAX_PARTICLES]; /* Particles on all nodes */
int counts[MAX_PROCS]; /* Number of ptcls on each proc */
int displacements[MAX_PROCS]; /* Offsets into particles */
int offsets[MAX_PROCS]; /* Offsets used by the master */
int particle_number, i, j, my_offset, true_i;
int total_particles; /* Total number of particles */
int count; /* Count time steps */

MPI_Datatype particle_type;

double dt = DT; /* Integration time step */
double comm_time, start_comm_time, end_comm_time, start_time, end_time;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &pool_size);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

if (my_rank == MASTER_RANK) i_am_the_master = TRUE;

particle_number = MAX_PARTICLES / pool_size;

if (i_am_the_master)
    printf ("%d particles per processor\n", particle_number);

MPI_Type_contiguous ( 11, MPI_DOUBLE, &particle_type );
MPI_Type_commit ( &particle_type );

MPI_Allgather ( &particle_number, 1, MPI_INT, counts, 1, MPI_INT,
               MPI_COMM_WORLD );

displacements[0] = 0;
for (i = 1; i < pool_size; i++)
    displacements[i] = displacements[i-1] + counts[i-1];
total_particles = displacements[pool_size - 1]
    + counts[pool_size - 1];

if (i_am_the_master)
    printf ("total number of particles = %d\n", total_particles);

my_offset = displacements[my_rank];

MPI_Gather ( &my_offset, 1, MPI_INT, offsets, 1, MPI_INT, MASTER_RANK,
            MPI_COMM_WORLD );

if (i_am_the_master) {
    printf ("offsets: ");
    for (i = 0; i < pool_size; i++)
        printf ("%d ", offsets[i]);
    printf("\n");
}

srand48((long) (my_rank + 28));

```

```

/* Here each process initializes its own particles. */

for (i = 0; i < particle_number; i++) {
    particles[my_offset + i].x = drand48();
    particles[my_offset + i].y = drand48();
    particles[my_offset + i].z = drand48();
    particles[my_offset + i].vx = 0.0;
    particles[my_offset + i].vy = 0.0;
    particles[my_offset + i].vz = 0.0;
    particles[my_offset + i].ax = 0.0;
    particles[my_offset + i].ay = 0.0;
    particles[my_offset + i].az = 0.0;
    particles[my_offset + i].mass = 1.0;
    particles[my_offset + i].charge = 1.0 - 2.0 * (i % 2);
}

start_time = MPI_Wtime();
comm_time = 0.0;

for (count = 0; count < N_OF_ITERATIONS; count++) {

    if (i_am_the_master) printf("Iteration %d.\n", count + 1);

    /* Here processes exchange their particles with each other. */

    start_comm_time = MPI_Wtime();

    MPI_Allgatherv ( particles + my_offset, particle_number,
                    particle_type,
                    particles, counts, displacements, particle_type,
                    MPI_COMM_WORLD );

    end_comm_time = MPI_Wtime();
    comm_time += end_comm_time - start_comm_time;

    for (i = 0; i < particle_number; i++) {

        true_i = i + my_offset;

        /* initialize accelerations to zero */

        particles[true_i].ax = 0.0;
        particles[true_i].ay = 0.0;
        particles[true_i].az = 0.0;

        for (j = 0; j < total_particles; j++) {

            /* Do not evaluate interaction with yourself. */

            if (j != true_i) {

                /* Evaluate forces that j-particles exert on the i-particle. */

                double dx, dy, dz, r2, r, qj_by_r3;

                /* Here we absorb the minus sign by changing the order

```

```

        of i and j. */

        dx = particles[true_i].x - particles[j].x;
        dy = particles[true_i].y - particles[j].y;
        dz = particles[true_i].z - particles[j].z;

        r2 = dx * dx + dy * dy + dz * dz; r = sqrt(r2);

        /* Quench the force if the particles are too close. */

        if (r < EPSILON) qj_by_r3 = 0.0;
        else qj_by_r3 = particles[j].charge / (r2 * r);

        /* accumulate the contribution from particle j */

        particles[true_i].ax += qj_by_r3 * dx;
        particles[true_i].ay += qj_by_r3 * dy;
        particles[true_i].az += qj_by_r3 * dz;
    }
}

/*
 * We advance particle positions and velocities only *after*
 * we have evaluated all accelerations using the *old* positions.
 */

for (i = 0; i < particle_number; i++) {

    double qidt_by_m, dt_by_2, vx0, vy0, vz0;

    true_i = i + my_offset;

    /* Save old velocities */

    vx0 = particles[true_i].vx;
    vy0 = particles[true_i].vy;
    vz0 = particles[true_i].vz;

    /* Now advance the velocity of particle i */

    qidt_by_m = particles[true_i].charge * dt / particles[true_i].mass;
    particles[true_i].vx += particles[true_i].ax * qidt_by_m;
    particles[true_i].vy += particles[true_i].ay * qidt_by_m;
    particles[true_i].vz += particles[true_i].az * qidt_by_m;

    /* Use average velocity in the interval to advance the particles'
       positions */

    dt_by_2 = 0.5 * dt;
    particles[true_i].x += (vx0 + particles[true_i].vx) * dt_by_2;
    particles[true_i].y += (vy0 + particles[true_i].vy) * dt_by_2;
    particles[true_i].z += (vz0 + particles[true_i].vz) * dt_by_2;
}
}

MPI_Barrier(MPI_COMM_WORLD);

```

```

end_time = MPI_Wtime();

if (i_am_the_master) {
    printf ("Communication time %8.5f seconds\n", comm_time);
    printf ("Computation time %8.5f seconds\n",
            end_time - start_time - comm_time);
    printf ("\tEvaluated %d interactions\n",
            N_OF_ITERATIONS * total_particles * (total_particles - 1));
}

MPI_Finalize ();

exit(0);
}

```

The code is compiled using `mpicc` and linked with the mathematics library `-lm`, which contains the definition of square root:

```

gustav@bh1 $ pwd
/N/B/gustav/src/I590/particles
gustav@bh1 $ make install
co RCS/Makefile,v Makefile
RCS/Makefile,v --> Makefile
revision 1.2
done
co RCS/particles.c,v particles.c
RCS/particles.c,v --> particles.c
revision 1.7
done
mpicc -o particles particles.c -lm
install particles /N/B/gustav/bin
gustav@bh1 $

```

I run it on 32 CPUs:

```

gustav@bh1 $ cd
gustav@bh1 $ mpdtrace | wc
    32    32    159
gustav@bh1 $ mpiexec -n 32 particles
312 particles per processor
total number of particles = 9984
offsets: 0 312 624 936 1248 1560 1872 2184 2496 2808 3120 \
3432 3744 4056 4368 4680 4992 5304 5616 5928 6240 6552 6864 \
7176 7488 7800 8112 8424 8736 9048 9360 9672
Iteration 1.
Iteration 2.
Iteration 3.
...
Iteration 18.
Iteration 19.
Iteration 20.
Communication time 1.72091 seconds
Computation time 14.97211 seconds
    Evaluated 1993405440 interactions
gustav@bh1 $

```

Observe that the communication time is quite short compared to the computation time. This type of computation benefits from the CPU farm model.

### The Discussion

The program begins with the usual MPI incantations that result in every process finding about its place within the pool and about the size of the pool of processes. A process whose rank is `MASTER_RANK` assumes the “leadership” of the pool, which in this program only means that it’s going to write stuff on standard output.

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &pool_size);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
if (my_rank == MASTER_RANK) i_am_the_master = TRUE;
```

Now we find out how many particles each process will have to deal with. This is simply going to be `MAX_PARTICLES/pool_size`. If `MAX_PARTICLES` does not divide by `pool_size` exactly, we’re going to work with fewer particles in this program. In more elaborate codes, you may wish to give some processes more particles. This code uses tools that allow for non-uniform particle distribution amongst processes.

```
particle_number = MAX_PARTICLES / pool_size;
if (i_am_the_master)
    printf ("%d particles per processor\n", particle_number);
```

The next step is to *define* a new MPI data type, so that we send and receive the *whole particle* as an individual data item. A particle in this code is defined as a structure:

```
typedef struct {
    double x, y, z, vx, vy, vz, ax, ay, az, mass, charge;
} Particle;
```

whose slots are position coordinates,  $x, y, z$ , velocity in  $x, y$  and  $z$  direction,  $vx, vy, vz$ , accelerations in  $x, y$  and  $z$  directions,  $ax, ay, az$ , mass and charge. All particles are arranged into an array:

```
Particle particles[MAX_PARTICLES]; /* Particles on all nodes */
```

How are we to tell MPI what a `Particle` is? In order to do this we must know how the C-language is going to store the structure. It would be good if we could simply pass `Particle` to MPI as an argument, but `Particle` is not a variable. It is a type, and types cannot be passed to functions as variables.

In this program we assume that C is going to store all components of structure `Particle` contiguously (if it doesn’t, the program is going to crash – this is the problem with MPI: it’s too low level). The components are all of the same type `double` and there are 11 of them. So we tell MPI, by calling function `MPI_Type_contiguous`, that we are going to operate on chunks of 11 contiguously laid out doubles and we want MPI to store its own type description of a chunk like this on a variable called `particle_type`, which is of type `MPI_Datatype`:

```
MPI_Type_contiguous ( 11, MPI_DOUBLE, &particle_type );
MPI_Type_commit ( &particle_type );
```

Once an MPI type has been defined, it must be *committed* by calling `MPI_Type_commit`. This call gives the MPI system a chance to *compile* this information on-the-fly, so that it can be used efficiently in communication calls that follow.

Now we encounter a new fancy communication between processes. We call function `MPI_Allgather` to tell each process how many particles each other process is going to look after.

```
MPI_Allgather ( &particle_number, 1, MPI_INT, counts, 1, MPI_INT,
               MPI_COMM_WORLD );
```

This call works as follows. Every process contributes an array given by the first argument. The array comprises a number of elements given by the second argument and of type given by the third argument. These arrays are then put together to form a new array given by the fourth argument. In this operation the types of individual data items in the contributed arrays *may be* reinterpreted (cast) and this, in turn, may also change the number of items that are received from each process. The number of received items is therefore given in the fifth argument and the type of the received items is given in the sixth argument. I don't know how much call there is for such re-interpretations of transmitted data and I wouldn't use this trick in my own programs, but MPI provides for it, if the programmer wants to play with fire. Observe that `MPI_Allgather` fills the array `counts` (in this case) for *all participating processes*. There is another similar function called `MPI_Gather` which performs a *gather* operation like `MPI_Allgather` but leaves the result on a *root* process only. Function `MPI_Gather` has one more argument, the rank of the root process, which is placed just before the communicator.

In this specific program every process ends up looking after the same number of particles, so it is not strictly necessary to call `MPI_Allgather`, but remember that I promised that the infrastructure of the program would let you assign different numbers of particles to processes. If these numbers were to be arrived at by some tricky algorithm that each process would perform for itself and that would yield different numbers of particles for each process, we would need `MPI_Allgather` to send this information to all other processes.

In the following section of the code we find about the portions of array `particles` that each process is going to be responsible for. We construct an array called `displacements`. For process of rank 0 its first particle is going to be `particles[displacements[0]]`. For process of rank 1 its first particle is going to be `particles[displacements[1]]` and so on. The specific displacement for "me" is going to be called `my_offset` and it is simply `displacements[my_rank]`:

```
displacements[0] = 0;
for (i = 1; i < pool_size; i++)
    displacements[i] = displacements[i-1] + counts[i-1];
total_particles = displacements[pool_size - 1]
    + counts[pool_size - 1];

if (i_am_the_master)
    printf ("total number of particles = %d\n", total_particles);

my_offset = displacements[my_rank];
```

Here is the example of calling function `MPI_Gather`. Every process sends a one-item long array to the root process. The array contains simply `my_offset`. The root process assembles these contributions into an array `offsets`, which is then printed on standard output.

```
MPI_Gather ( &my_offset, 1, MPI_INT, offsets, 1, MPI_INT, MASTER_RANK,
            MPI_COMM_WORLD );

if (i_am_the_master) {
    printf ("offsets: ");
    for (i = 0; i < pool_size; i++)
        printf ("%d ", offsets[i]);
    printf("\n");
}
```

This part of the code doesn't really do anything important, other than illustrating the use of `MPI_Gather`.

Now each process initializes its own particles. It seeds the random number generator with its rank number and then initializes  $x$ ,  $y$  and  $z$  coordinates of its particles to random double precision numbers between 0 and 1.

```
srand48((long) (my_rank + 28));

/* Here each process initializes its own particles. */

for (i = 0; i < particle_number; i++) {
    particles[my_offset + i].x = drand48();
    particles[my_offset + i].y = drand48();
    particles[my_offset + i].z = drand48();
    particles[my_offset + i].vx = 0.0;
    particles[my_offset + i].vy = 0.0;
    particles[my_offset + i].vz = 0.0;
    particles[my_offset + i].ax = 0.0;
    particles[my_offset + i].ay = 0.0;
    particles[my_offset + i].az = 0.0;
    particles[my_offset + i].mass = 1.0;
    particles[my_offset + i].charge = 1.0 - 2.0 * (i % 2);
}
```

Now the real computation begins. The first thing we do is to initialize timers. We are going to measure the total wall clock time used by the program to perform its `N_OF_ITERATIONS` time steps and, separately, time spent communicating, i.e., time spent within the MPI function calls.

```
start_time = MPI_Wtime();
comm_time = 0.0;
```

The iteration loop begins with a timed call to function `MPI_Allgatherv`.

```
for (count = 0; count < N_OF_ITERATIONS; count++) {

    if (i_am_the_master) printf("Iteration %d.\n", count + 1);

    /* Here processes exchange their particles with each other. */

    start_comm_time = MPI_Wtime();

    MPI_Allgatherv ( particles + my_offset, particle_number,
```



```

        particle_type,
        particles, counts, displacements, particle_type,
        MPI_COMM_WORLD );

    end_comm_time = MPI_Wtime();
    comm_time += end_comm_time - start_comm_time;

```

This is a yet another variant of a *gather* operation. It differs from `MPI_Allgather` by allowing processes to contribute arrays of *different length*. Remember that all contributions in `MPI_Allgather` had to be of the same length. But here every process contributes information about its particles to the pool and the number of particles the processes look after *may differ* from a process to a process (although in this simple program they do not). Hence the need to use `MPI_Allgather_v` rather than `MPI_Allgather`.

Each process contributes particles from the location in the array given by `particles + my_offset`. This is an example of pointer arithmetic. The meaning of this hideous, dangerous and potentially misleading expression is `&(particles[my_offset])`, and you could use the latter expression in this context too. The length of the contributed array is `particle_number` (and this number *may* be now different for each process) and the type of the items in the array is `particle_type`. So far this looks much the same as what we have seen in `MPI_Allgather`. But now we have to assemble these contributions into array `particles` keeping in mind that we may have received different numbers of particles from participating processes. The two arrays that follow `particles` in the arguments list specify how many items (`count`) of type `particle_type` should be placed at the location given by an entry in the array `displacements`. And so, `count[0]` items of `particle_type` should go into the location pointed by `displacements[0]` in the array `particles`, `count[1]` items of `particle_type` should go into the location pointed by `displacements[1]` in the array `particles`, and so on.

After the operation completes, every process is going to have data pertaining to *all* particles in its local copy of the array `particles`.

Now we can finally commence the computation. The  $i$  particles are the ones  $I$  am responsible for (“ $I$ ” as a process, this is).

```

    for (i = 0; i < particle_number; i++) {

        true_i = i + my_offset;

        /* initialize accelerations to zero */

        particles[true_i].ax = 0.0;
        particles[true_i].ay = 0.0;
        particles[true_i].az = 0.0;
    }

```

We begin by setting accelerations for the  $i$  particle to zero, i.e.,

$$\mathbf{a}_i = 0 \quad \text{for each } i$$

Now we are going to calculate how the  $i$  particle is tugged by all the  $j$  particles it interacts with. Remember that the  $i$  particle does not interact with itself, but it *does* interact with every other particle.

```

for (j = 0; j < total_particles; j++) {
    /* Do not evaluate interaction with yourself. */

    if (j != true_i) {

        /* Evaluate forces that j-particles exert on the i-particle. */

        double dx, dy, dz, r2, r, qj_by_r3;

        /* Here we absorb the minus sign by changing the order
           of i and j. */

        dx = particles[true_i].x - particles[j].x;
        dy = particles[true_i].y - particles[j].y;
        dz = particles[true_i].z - particles[j].z;

        r2 = dx * dx + dy * dy + dz * dz; r = sqrt(r2);

```

Here we have evaluated  $\mathbf{r}_i - \mathbf{r}_j = \mathbf{r}_{ji} = -\mathbf{r}_{ij}$ , this is what sits in dx, dy, dz. We have also evaluated  $r_{ji}^2 = r_{ij}^2$ , this is what sits in r2 and  $r_{ji} = r_{ij}$ , this is what sits in r.

It may happen that  $r_{ji}$  is so small that  $q_j/r_{ji}^3$  overflows. Here we implement a very primitive quenching mechanism. If the particles are closer to each other than  $\epsilon$ , we switch the force between them off. This is not physical, but it is safe. Real plasma codes implement more elaborate means of dealing with this problem. If the particles are not too close we evaluate

$$\frac{q_j}{r_{ji}^3}$$

```

/* Quench the force if the particles are too close. */

if (r < EPSILON) qj_by_r3 = 0.0;
else qj_by_r3 = particles[j].charge / (r2 * r);

```

Now we can finally sum up the contributions from  $j$  particles to the acceleration that acts on the  $i$  particle:

$$\sum_{\substack{j=1 \\ (j \neq i)}}^{j=N} \frac{q_j}{r_{ji}^3} (\mathbf{r}_i - \mathbf{r}_j)$$

```

particles[true_i].ax += qj_by_r3 * dx;
particles[true_i].ay += qj_by_r3 * dy;
particles[true_i].az += qj_by_r3 * dz;
}
}
}

```

Having evaluated the accelerations using the *old* positions of the  $i$  and  $j$  particles, we can now advance velocities of the  $i$  particles, and then advance their positions as well. We do this within the next for(i loop:

```

/*
 * We advance particle positions and velocities only *after*
 * we have evaluated all accelerations using the *old* positions.
 */

for (i = 0; i < particle_number; i++) {

    double qidt_by_m, dt_by_2, vx0, vy0, vz0;

    true_i = i + my_offset;

```

But first we save current velocities of the particles on auxiliary variables, because we will need them to advance particle positions:

```

/* Save old velocities */

vx0 = particles[true_i].vx;
vy0 = particles[true_i].vy;
vz0 = particles[true_i].vz;

```

And now we implement the formula:

$$\mathbf{v}_i(t + \Delta t) = \mathbf{v}_i(t) + \frac{q_i \Delta t}{m_i} \sum_{\substack{j=1 \\ j \neq i}}^{j=N} \frac{q_j}{r_{ji}(t)^3} (\mathbf{r}_i(t) - \mathbf{r}_j(t))$$

```

/* Now advance the velocity of particle i */

qidt_by_m = particles[true_i].charge * dt / particles[true_i].mass;
particles[true_i].vx += particles[true_i].ax * qidt_by_m;
particles[true_i].vy += particles[true_i].ay * qidt_by_m;
particles[true_i].vz += particles[true_i].az * qidt_by_m;

```

Finally, we *push* the  $i$  particle, i.e., advance their positions using the formula:

$$\mathbf{r}_i(t + \Delta t) = \mathbf{r}_i(t) + \frac{\mathbf{v}_i(t) + \mathbf{v}_i(t + \Delta t)}{2} \Delta t$$

```

/* Use average velocity in the interval to advance the particles'
   positions */

dt_by_2 = 0.5 * dt;
particles[true_i].x += (vx0 + particles[true_i].vx) * dt_by_2;
particles[true_i].y += (vy0 + particles[true_i].vy) * dt_by_2;
particles[true_i].z += (vz0 + particles[true_i].vz) * dt_by_2;
}

```

This ends the computation within the top level `for(count` loop. We repeat this loop for `N_OF_ITERATIONS` times – thus performing `N_OF_ITERATIONS` time steps.

Having completed the computation we meet at the barrier and the master process prints on standard output how much time it spent within MPI communications and how much time it spent computing.

```

MPI_Barrier(MPI_COMM_WORLD);

```

```

    end_time = MPI_Wtime();

    if (i_am_the_master) {
        printf ("Communication time %8.5f seconds\n", comm_time);
        printf ("Computation time   %8.5f seconds\n",
                end_time - start_time - comm_time);
        printf ("\tEvaluated %d interactions\n",
                N_OF_ITERATIONS * total_particles * (total_particles - 1));
    }

    MPI_Finalize ();

    exit(0);
}

```

Then all processes meet again on `MPI_Finalize` and `exit`.

### Exercises

- 1 Experiment with the program discussed in section 5.2.6 changing the number of processes the program runs on and the total number of particles as well. Observe how the ratio of communication to computation changes dramatically as the particle number grows. This is a very computation-intensive problem and it parallelizes quite well for a large number of particles.
- 2 Rewrite the program so that processes handle different numbers of particles.

### 5.2.7 The Random Pie

In this section we are going to evaluate  $\pi$ , yet again, using a Monte Carlo method. Monte Carlo methods are quite popular because they are usually rather easy to code and to parallelize, but they can be also inaccurate and slow.

The way we're going to employ Monte Carlo in this code is as follows. The area of a circle is given by  $\mathcal{A}_o = \pi r^2$ , where  $r$  is the radius of the circle. For  $r = 1$  we have that  $\mathcal{A}_o = \pi$ . The area of a square this circle is fitted in is  $\mathcal{A}_s = 2 \times 2 = 4$ . So the ratio of  $\mathcal{A}_o/\mathcal{A}_s = \pi/4$ .

Now imagine that you throw grains of sand, at random, at the square. We expect that  $\pi/4$  of all grains will end up within the circle, the remaining grains landing outside.

Within this program we are going to have worker processes throw sand grains at random at the square and check if the grains have landed within or without the circle. Every now and then the processes will collect the results from each other in order to improve their estimate of  $\pi$ . We are going to have one process, set aside, whose only job will be to generate random numbers. The process will send the numbers to the workers on request. This guarantees that each worker is going to get *different* random numbers. In our other programs we dealt with this problem by using the process rank number to seed the random number generator differently. So this is a new strategy.

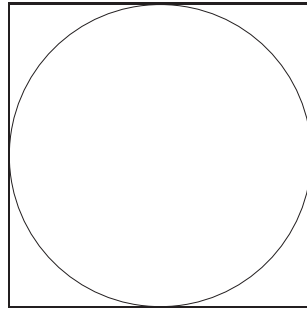


Figure 5.1: Dividing the number of grains that have landed within the circle by the total number of sand grains thrown at the square should give us  $\pi/4$ .

The process that is set aside will be excluded from the communicator, which the worker processes will use to exchange their estimates of  $\pi$  with each other. So we are going to learn on this occasion how to construct such a communicator too.

The computation goes on until the required accuracy is reached, and this accuracy is passed through the command line, the parameter is called  $\epsilon$ , or until the maximum number of sand grains has been thrown on the square.

### The Code

```

/*
 * %Id: randompie.c,v 1.1 2003/10/12 20:22:50 gustav Exp %
 *
 * %Log: randompie.c,v %
 * Revision 1.1 2003/10/12 20:22:50 gustav
 * Initial revision
 *
 */

#include <stdio.h>
#include <stdlib.h> /* function random is defined here */
#include <limits.h> /* INT_MAX is defined here */
#include <mpi.h>

#define CHUNKSIZE 1000
#define REQUEST 1
#define REPLY 2
#define TOTAL 5000000

main(argc, argv)
    int argc;
    char *argv[];
{
    int iter;
    int in, out, i, iters, max, ix, iy, ranks[1], done, temp;

```

```

double x, y, Pi, error, epsilon;
int numprocs, myid, server, totalin, totalout, workerid;
int rands[CHUNKSIZE], request;
MPI_Comm world, workers;
MPI_Group world_group, worker_group;
MPI_Status stat;

MPI_Init(&argc, &argv);
world = MPI_COMM_WORLD;
MPI_Comm_size(world, &numprocs);
MPI_Comm_rank(world, &myid);
server = numprocs - 1;
if (myid == 0)
    sscanf( argv[1], "%lf", &epsilon);
MPI_Bcast(&epsilon, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Comm_group( world, &world_group);
rands[0] = server;
MPI_Group_excl(world_group, 1, rands, &worker_group);
MPI_Comm_create(world, worker_group, &workers);
MPI_Group_free(&worker_group);

if(myid == server) {
    do {
        MPI_Recv(&request, 1, MPI_INT, MPI_ANY_SOURCE, REQUEST, world, &stat);
        if (request) {
            for (i = 0; i < CHUNKSIZE; i++)
                rands[i] = random();
            MPI_Send(rands, CHUNKSIZE, MPI_INT, stat.MPI_SOURCE, REPLY, world);
        }
    }
    while (request > 0);
}
else {
    request = 1;
    done = in = out = 0;
    max = INT_MAX;
    MPI_Send(&request, 1, MPI_INT, server, REQUEST, world);
    MPI_Comm_rank(workers, &workerid);
    iter = 0;
    while(!done) {
        iter++;
        request = 1;
        MPI_Recv(rands, CHUNKSIZE, MPI_INT, server, REPLY, world, &stat);
        for (i=0; i < CHUNKSIZE; ) {
            x = (((double) rands[i++])/max) * 2 - 1;
            y = (((double) rands[i++])/max) * 2 - 1;
            if (x*x + y*y < 1.0)
                in++;
            else
                out++;
        }
        MPI_Allreduce(&in, &totalin, 1, MPI_INT, MPI_SUM, workers);
        MPI_Allreduce(&out, &totalout, 1, MPI_INT, MPI_SUM, workers);
        Pi = (4.0*totalin)/(totalin + totalout);
        error = fabs( Pi-3.141592653589793238462643);
        done = ((error < epsilon) || ((totalin+totalout) > TOTAL));
        request = (done) ? 0 : 1;
    }
}

```

```

    if (myid == 0) {
        printf( "\rpi = %23.201f", Pi);
        MPI_Send(&request, 1, MPI_INT, server, REQUEST, world);
    }
    else {
        if (request)
            MPI_Send(&request, 1, MPI_INT, server, REQUEST, world);
    }
}
MPI_Comm_free(&workers);
}
if (myid == 0)
    printf( "\npoints: %d\nin: %d, out: %d\n",
            totalin+totalout, totalin, totalout);
MPI_Finalize();
exit (0);
}

```

Here's how the code is compiled and installed:

```

gustav@bh1 $ pwd
/N/B/gustav/src/I590/randompie
gustav@bh1 $ make
co RCS/Makefile,v Makefile
RCS/Makefile,v --> Makefile
revision 1.1
done
co RCS/randompie.c,v randompie.c
RCS/randompie.c,v --> randompie.c
revision 1.1
done
mpicc -o randompie randompie.c
gustav@bh1 $ make install
install randompie /N/B/gustav/bin
gustav@bh1 $

```

And here's how the code is run. Observe that the MPI program takes one argument from the command line, namely the  $\epsilon$ . It is simply typed on the command line following the name of the MPI program, `randompie`.

```

gustav@bh1 $ mpdtrace | wc
    32    32   159
gustav@bh1 $ mpiexec -n 16 randompie 0.0001
pi = 3.14164761904761924427
points: 420000
in: 329873, out: 90127
gustav@bh1 $

```

## The Discussion

Like all other MPI programs, this program also begins with the usual incantations. But here we are going to create a new communicator by manipulating the default one, so we copy the default communicator, which is an MPI constant, onto a variable of type `MPI_Comm`, `world`:

```

MPI_Init(&argc, &argv);
world = MPI_COMM_WORLD;
MPI_Comm_size(world, &numprocs);
MPI_Comm_rank(world, &myid);

```

We are going to exclude one process from the pool, and this process will be employed to generate random numbers and then send them to other workers. We are going to use a process with the highest rank number for this:

```
server = numprocs - 1;
```

In the next line we employ process of rank 0 to read the value of  $\epsilon$  from the command line and to broadcast it to all other processes:

```
if (myid == 0)
    sscanf( argv[1], "%lf", &epsilon);
MPI_Bcast(&epsilon, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Now we commence the communicator manipulations. The communicator comprises a number of elements. First, there are the processes themselves. The processes when abstracted from the communicator form a *group*. But there is more to a communicator than just a group of processes that enter it. A communicator may order the processes in a special way, e.g., to form a Cartesian grid, as we have seen before, and a communicator arranges for separate communication channels between its processes too. Messages are differentiated by the means of tag numbers, recipient numbers, sender numbers and the communicator within which they are transmitted. But here it is the group of processes that we want to reform. And so we begin by calling `MPI_Comm_group`, which is going to extract the group of processes from the existing communicator `world` and deposit it on a variable of type `MPI_Group` here called `worker_group`:

```
MPI_Comm_group( world, &world_group);
```

Type `MPI_Group` is an opaque type. It is not an array, which you could manipulate directly. In order to manipulate variables of this type, you have to call special MPI functions that deal with groups of processes. Here we want to *exclude* the random number server process from the group. And here is how it's done:

```
ranks[0] = server;
MPI_Group_excl(world_group, 1, ranks, &worker_group);
```

Function `MPI_Group_excl` takes an existing group, here it is `world_group`. Then it takes an array of ranks, here called `ranks` of length specified by the second argument, here it is 1. And finally it creates a new group of processes, here it is called `worker_group`, by excluding processes listed in the array `ranks` from the `world_group` group.

So this is an example of how you can manipulate process groups. Now we need to make this new group, `worker_group` into a communicator. This is done by calling `MPI_Comm_create`. This function takes an existing communicator from which processes that form the `worker_group` have been extracted and then creates a new communicator, here called `workers`, that contains processes of the `worker_group` *only*.

```
MPI_Comm_create(world, worker_group, &workers);
```

Once the communicator has been created, we have no more need for the group `worker_group` itself. So we can free the storage that's been allocated for the `worker_group` variable by calling function `MPI_Group_free`

```
MPI_Group_free(&worker_group);
```



We could free right here and now the other group `world_group` too, since the program does not make any more use of it.

Processes within the new communicator called `workers` may have different rank numbers, but the variable `myid` has been established within the original `MPI_COMM_WORLD` and only one process is going to have `myid == server` and this is exactly the process that has been excluded from `workers`. Whenever `workers` will communicate with the `server` they will have to send messages within the `world` communicator. Whenever they want to communicate just amongst themselves, they will have to send messages within the `workers` communicator.

Now the program splits into two parts. The first part that corresponds to the first clause of the `if` statement describes the `server`'s activities, the second part describes the workers' activities.

The server enters a loop, within which it waits for a request, i.e., a message that may arrive from `MPI_ANY_SOURCE`, with the tag number set to `REQUEST` sent to it within the `world` communicator. The message will contain just one integer. If the integer is greater than zero (this means `TRUE` in the C-language) the server process is going to generate `CHUNKSIZE` of random integers between 0 and `INT_MAX` (which is defined on `limits.h`). The integers will be written on `rand`, which is then going to be sent back to the requesting process of rank `stat.MPI_SOURCE` with the tag number set to `REPLY`. The communication still takes place within the `world` communicator, because the `server` does not belong to any other communicator.

The server is going to loop until someone sends it `request == 0`. Sending `request == 0` to the server terminates the server, which then goes right to `MPI_Finalize` to wait for other processes.

```

if(myid == server) {
  do {
    MPI_Recv(&request, 1, MPI_INT, MPI_ANY_SOURCE, REQUEST, world, &stat);
    if (request) {
      for (i = 0; i < CHUNKSIZE; i++)
        rands[i] = random();
      MPI_Send(rands, CHUNKSIZE, MPI_INT, stat.MPI_SOURCE, REPLY, world);
    }
  }
  while (request > 0);
}

```

The worker's life is a little more complicated. The workers are the ones who have to make various decisions in this program. They begin their lives by sending a request to the `server` for a string of random integers.

```

else {
  request = 1;
  done = in = out = 0;
  max = INT_MAX;
  MPI_Send(&request, 1, MPI_INT, server, REQUEST, world);
}

```

Then they find about their rank numbers within the `workers` communicator. But in this program they are not going to make any use of it, so we could just as well skip this call to `MPI_Comm_rank`:

```
MPI_Comm_rank(workers, &workerid);
```

Now the iteration process begins. The iterations are going to be repeated until the job is `done`, i.e., until  $\pi$  has been evaluated with precision better than  $\epsilon$  or until the total number of sand grains thrown at the square has exceeded `TOTAL`. Each iteration begins by receiving a string of random integers from the `server`:

```
iter = 0;
while(!done) {
    iter++;
    request = 1;
    MPI_Recv(rands, CHUNKSIZE, MPI_INT, server, REPLY, world, &stat);
```

Having received the numbers, the worker process “throws” them at the square. We take two random integers from the square. Then we renormalize them to reals that fit between -1 and 1 and we call them  $x$  and  $y$ . Then the process checks if the sand grain so produced hits the circle or the part of the square that’s outside by checking if  $x^2 + y^2 < 1$ . If we have scored the hit, we increment the `in` counter, if we have missed we increment the `out` counter.

```
for (i=0; i < CHUNKSIZE; ) {
    x = (((double) rands[i++])/max) * 2 - 1;
    y = (((double) rands[i++])/max) * 2 - 1;
    if (x*x + y*y < 1.0)
        in++;
    else
        out++;
}
```

Each process could evaluate `pi` on its own, but the evaluation will be more accurate if they can add all their scores. This is done by calling the `MPI_Allreduce` function, which is like `MPI_Reduce`, which we have encountered already, but with the difference that the result of the reduction operation is distributed to *all* participating processes. Remember that in `MPI_Reduce` only the *root* process had the number.

```
MPI_Allreduce(&in, &totalin, 1, MPI_INT, MPI_SUM, workers);
MPI_Allreduce(&out, &totalout, 1, MPI_INT, MPI_SUM, workers);
```

This communication takes place within the `workers` communicator. This is the only place where the workers communicate with each other. Because the communication is always collective, they don’t need to know their rank numbers within the `workers` communicator.

Now we can evaluate our estimate of  $\pi$ . `totalin + totalout` is the total number of sand grains thrown at the square, whereas `totalin` is the total number of sand grains that have landed within the circle:

```
Pi = (4.0*totalin)/(totalin + totalout);
```

Since we know from elsewhere what’s the value of  $\pi$ , we can easily evaluate the accuracy of our Monte Carlo experiment:

```
error = fabs( Pi-3.141592653589793238462643);
```

and use this result in checking if the job is done:

```
done = ((error < epsilon) || ((totalin+totalout) > TOTAL));
```

Now we use this funny C-language construct, which encrypts `if(done) request=0; else request=1;` and process of rank zero sends the `request` to the `server` after it has printed the value of  $\pi$  on standard output. This value is printed in such a way that it overwrites the previously printed value: the message begins with the carriage return and there is no new-line.

The reason why process of rank zero sends the request to the `server` separately from other processes is because the request may be a termination request, in which case other processes should not send their termination requests. There wouldn't be anyone at the `server`'s end to receive them. The other processes send the request to the `server` only if there is more work to be done:

```

request = (done) ? 0 : 1;
if (myid == 0) {
    printf( "\rpi = %23.20lf", Pi);
    MPI_Send(&request, 1, MPI_INT, server, REQUEST, world);
}
else {
    if (request)
        MPI_Send(&request, 1, MPI_INT, server, REQUEST, world);
}
}

```

And this ends the `while(!done)` loop. There is one more thing that the worker processes are going to do before they quit. They terminate their communicator by calling `MPI_Comm_free`

```

MPI_Comm_free(&workers);
}

```

Having released themselves from the `workers` communicator they go to `MPI_Finalize` to wait for each other. In the meantime process of rank 0 (within the `world` communicator) prints the total number of sand grains thrown at the square on standard output.

```

if (myid == 0)
    printf( "\npoints: %d\nin: %d, out: %d\n",
            totalin+totalout, totalin, totalout);
MPI_Finalize();
exit (0);
}

```

## Exercises

- 1 Program `randompie` of section 5.2.7 does not check for the correctness of the parameter passed on the command line. It does not check if anything is passed on the command line to begin with. Try running this program without passing the value of `epsilon` on the command line and see what happens.
- 2 Rewrite program `randompie` so that process of rank zero checks the command line for any parameters, then checks if the parameters are correct, then sends the OK message to other processes if it has found the value of  $\epsilon$  there. Otherwise process rank zero should write an error message on standard error and send a termination message to other processes that would make them go directly to `MPI_Finalize`.

- 3 Try varying `epsilon` and `TOTAL`. Observe that the accuracy of  $\pi$  does not improve beyond a certain `TOTAL` number of sand grains used in the computation. Why?

### 5.2.8 Error Handling

In this section we are going to look at how to handle errors in MPI programs.

There are two categories of errors we are going to encounter. UNIX errors and MPI errors.

MPI errors can arise when messages are incorrectly constructed, addressed, sent or received, or when they get lost because of network problems or because some nodes that participate in the communication may have crashed. The latter falls into the category of largely intractable problems. Even if we could diagnose such problems from within a program, there is little we could do about them. Consequently, MPI does not really provide mechanisms for dealing with failures in the communication system. Instead, the MPI-2 standard assumes that MPI implementors are going to “insulate the user from this unreliability” by developing systems that are sufficiently robust and fault tolerant to the extent allowed by the present day technology.

But the former, i.e., MPI program errors, ought to be tractable, and it is here that MPI provides mechanisms for handling recoverable errors. But this requires explicitly changing the default behaviour of an MPI program, which is to *abort all parallel computation* on having detected an MPI program error.

We are going to talk more about it in the second part of this section.

In the first part we are going to look at how we can use MPI itself (with the assumption that MPI, at least, does not fail) in order to process UNIX errors that may arise during program execution.

Here the difficulty is caused by the fact that on capturing a UNIX error a process should not just *exit*. Well, it may, but this would take down the whole parallel program and we may be none the wiser as to why it happened. The trick is to capture any UNIX problems that may arise and then to *build the logic of the parallel program around them*.

#### Handling UNIX Errors

In this section we are going to discuss a parallel version of program `mkrandfile` introduced in section 3.4.3. The parallel version of this program is called `mkrandfiles` (plural) and it tries to capture and process possible errors just as diligently as its sequential version. This program will also prepare us for the issues of MPI-IO. You can think of it as a prelude to MPI-IO.

Here is the program itself:

```
/*
 * %Id: mkrandfiles.c,v 1.6 2003/10/19 19:02:09 gustav Exp %
 *
 * %Log: mkrandfiles.c,v %
 * Revision 1.6 2003/10/19 19:02:09 gustav
 * Forgot to initialize basename to NULL.
```

```

*
* Revision 1.5  2003/10/19 18:58:59  gustav
* Corrected reading the command line.
*
* Revision 1.4  2003/10/13 22:49:45  gustav
* Moved the debug messages into the if clauses.
*
* Revision 1.3  2003/10/13 22:46:36  gustav
* Added more debug messages.
*
* Revision 1.2  2003/10/13 22:41:36  gustav
* Finished.
*
* Revision 1.1  2003/10/13 21:18:00  gustav
* Initial revision
*
*
*/

#include <stdio.h>  /* all IO stuff lives here */
#include <stdlib.h> /* exit lives here */
#include <unistd.h> /* getopt lives here */
#include <errno.h>  /* UNIX error handling lives here */
#include <string.h> /* strcpy lives here */
#include <mpi.h>    /* MPI and MPI-IO live here */

#define MASTER_RANK 0
#define TRUE 1
#define FALSE 0
#define BOOLEAN int
#define BLOCK_SIZE 1048576
#define SYNOPSIS printf ("synopsis: %s -f <file> -l <blocks>\n", argv[0])

int main(argc, argv)
    int argc;
    char *argv[];
{
    /* my variables */

    int my_rank, pool_size, number_of_blocks = 0, block, i;
    BOOLEAN i_am_the_master = FALSE, input_error = FALSE,
        my_file_open_error = FALSE, file_open_error = FALSE,
        my_write_error = FALSE, write_error = FALSE;
    char *basename = NULL, file_name[BUFSIZ], message[BUFSIZ];
    int basename_length, junk[BLOCK_SIZE];
    FILE *fp;
    double start, finish, io_time = 0.0;

    /* getopt variables */

    extern char *optarg;
    int c;

    /* error handling variables */

    extern int errno;

```

```

/* ACTION */

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &pool_size);
if (my_rank == MASTER_RANK) i_am_the_master = TRUE;

if (i_am_the_master) {

    /* read the command line */

    while ((c = getopt(argc, argv, "f:l:h")) != EOF)
        switch(c) {
            case 'f':
                basename = optarg;
break;
            case 'l':
if ((sscanf (optarg, "%d", &number_of_blocks) != 1) ||
    (number_of_blocks < 1))
    input_error = TRUE;
break;
            case 'h':
input_error = TRUE;
break;
            case '?':
input_error = TRUE;
break;
        }

    /* Check if the command line has initialized basename and
     * number_of_blocks.
     */

    if ((basename == NULL) || (number_of_blocks == 0)) input_error = TRUE;

    if (input_error)
        SYNOPSIS;
    else {
        basename_length = strlen(basename) + 1;
#ifdef DEBUG
        printf("basename          = %s\n", basename);
        printf("basename_length = %d\n", basename_length);
        printf("number_of_blocks = %d\n", number_of_blocks);
#endif
    }
}

/* Transmit the effect of reading the command line to other
processes. */

MPI_Bcast(&input_error, 1, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);

if (! input_error) {
    MPI_Bcast(&number_of_blocks, 1, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);
    MPI_Bcast(&basename_length, 1, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);
    if (! i_am_the_master) basename = (char*) malloc(basename_length);
    MPI_Bcast(basename, basename_length, MPI_CHAR, MASTER_RANK, MPI_COMM_WORLD);
}

```

```

#ifdef DEBUG
    printf("%3d: basename = %s, number_of_blocks = %d\n",
           my_rank, basename, number_of_blocks);
#endif

    /* Now every process creates its own file name and attempts
       to open the file. */

    sprintf(file_name, "%s.%d", basename, my_rank);

#ifdef DEBUG
    printf("%3d: opening file %s\n", my_rank, file_name);
#endif

    if (! (fp = fopen(file_name, "w"))) {
        sprintf(message, "%3d: %s", my_rank, file_name);
        perror(message);
        my_file_open_error = TRUE;
    }

    /* Now we must ALL check that NOBODY had problems
       with opening the file. */

    MPI_Allreduce (&my_file_open_error, &file_open_error, 1, MPI_INT,
                  MPI_LOR, MPI_COMM_WORLD);

#ifdef DEBUG
    if (i_am_the_master)
        if (file_open_error)
            fprintf(stderr, "problem opening output files\n");
#endif

    /* If all files are open for writing, write to them */

    if (! file_open_error) {
        srand(28 + my_rank);
        for (block = 0; (block < number_of_blocks) || my_write_error;
             block++) {
            for (i = 0; i < BLOCK_SIZE; junk[i++] = rand());
                start = MPI_Wtime();
            if (fwrite(junk, sizeof(int), BLOCK_SIZE, fp) != BLOCK_SIZE) {
                sprintf(message, "%3d: %s", my_rank, file_name);
                perror(message);
                my_write_error = TRUE;
            }
            finish = MPI_Wtime();
            io_time += finish - start;
        }

        /* Check if anybody had problems writing on the file */

        MPI_Allreduce (&my_write_error, &write_error, 1, MPI_INT,
                      MPI_LOR, MPI_COMM_WORLD);

#ifdef DEBUG
        if (i_am_the_master)

```

```

if (write_error)
    fprintf(stderr, "problem writing on files\n");
#endif
    if (i_am_the_master)
        if (!write_error)
            printf("io_time = %f\n", io_time);
    }

    /* Only processes that were successful opening the files
       need do close them here */

    if (!my_file_open_error) {
        fclose(fp);
#ifdef DEBUG
        printf ("%3d: closed %s\n", my_rank, file_name);
#endif
    }

    /* If we have either write errors or file open errors,
       then processes that managed to open their files
       are requested to throw them away */

    if ((write_error || file_open_error) && !my_file_open_error) {
        unlink(file_name);
#ifdef DEBUG
        printf("%3d: unlinked %s\n", my_rank, file_name);
#endif
    }

    /* We don't try to capture unlink or fclose errors here,
       because there is little we could do about them. */

}

MPI_Finalize();
exit(0);
}

```

Now let us discuss what the program does and how it goes about it.

The program begins the same way all other MPI programs do. All processes find about the size of the pool and their own rank within it. Then it befalls to the master process, i.e., process of rank zero, to read input from the command line using function `getopt`. The reading goes much the same as in `mkrandfile`, but this time we don't call `exit` whenever we encounter an error. Instead we have a boolean variable `input_error`, which is set by default to `FALSE`, and whenever the master process encounters a problem on reading the command line, instead of exiting it sets `input_error` to `TRUE`.

The command line expects the same options as before: `-f` followed by the name of the file, `-l` followed by the number of blocks that will be written on the file and `-h` if the user asks for quick help. But the file name is treated a little differently because it is a parallel program and without MPI-IO we cannot make all processes write on the same file. The processes will append their rank



number to the name obtained from the command line and each process will open a different file. For example if the program is invoked with

```
-f test
```

then the files that will be opened will be called `test.0`, `test.1`, `test.2` and so on.

The master process checks the correctness of input on the command line, checks if everything has been properly specified and sets `input_error` accordingly. The value of `input_error` is then broadcast to all processes:

```
MPI_Bcast(&input_error, 1, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);
```

and the rest of the program is one large

```
if (! input_error) {
    blah... blah... blah...
}
```

clause. What follows the clause is `MPI_Finalize`. In other words, if any input errors are encountered by the master process, nobody does anything. All processes go directly to `MPI_Finalize`, and the whole program exits cleanly.

Now, if there have been no input errors on the command line, we may still encounter various other errors and the logic of the program has to flow around them.

The `if(! input_error) {` clause begins with some broadcasts and a `malloc` that transmit `number_of_blocks` and `basename` (i.e., the word from which the file names will be constructed by appending the rank numbers to it) to all processes. On having received this data, the processes attempt to construct their file names, and then they try to open the files for writing. This is how they go about it:

```
if (! (fp = fopen(file_name, "w"))) {
    sprintf(message, "%3d: %s", my_rank, file_name);
    perror(message);
    my_file_open_error = TRUE;
}
```

A problem may arise at this stage, but if it does the processes affected by it should not just exit. Instead the process sets its own boolean variable `my_file_open_error` to `TRUE`. The default value of this variable is `FALSE`.

Now the logic of the program handles such error as follows. We check if *any* of the processes failed to open the file by calling `MPI_Allreduce` with the reduction operation set to logical `OR`:

```
/* Now we must ALL check that NOBODY had problems
   with opening the file. */

MPI_Allreduce (&my_file_open_error, &file_open_error, 1, MPI_INT,
               MPI_LOR, MPI_COMM_WORLD);
```

If any instance of `my_file_open_error` is `TRUE`, everybody's instance of `file_open_error` will be `TRUE` too. But by default `file_open_error` is `FALSE`.

If any of the processes failed to open its file, we don't compute anything and we don't write anything either. Instead processes that opened files successfully, close them and then delete them:

```

if (! file_open_error) {
    blah... blah... blah...
}
/* Only processes that were successful opening the files
   need do close them here */

if (!my_file_open_error) {
    fclose(fp);
#ifdef DEBUG
    printf ("%3d: closed %s\n", my_rank, file_name);
#endif
}

/* If we have either write errors or file open errors,
   then processes that managed to open their files
   are requested to throw them away */

if ((write_error || file_open_error) && !my_file_open_error) {
    unlink(file_name);
#ifdef DEBUG
    printf("%3d: unlinked %s\n", my_rank, file_name);
#endif
}

```

But if all processes managed to open their files without problems, we commence the computation, which works the same way it did in the sequential version of `mkrandfile...`

```

if (! file_open_error) {
    srand(28 + my_rank);
    for (block = 0; (block < number_of_blocks) || my_write_error;
        block++) {
        for (i = 0; i < BLOCK_SIZE; junk[i++] = rand());
        start = MPI_Wtime();
        if (fwrite(junk, sizeof(int), BLOCK_SIZE, fp) != BLOCK_SIZE) {
            sprintf(message, "%3d: %s", my_rank, file_name);
            perror(message);
            my_write_error = TRUE;
        }
        finish = MPI_Wtime();
        io_time += finish - start;
    }
}

```

... with one difference. If a process fails to write on its file, it sets the variable `my_write_error` to `TRUE`. the `for` statement checks the value of `my_write_error` at the top of the loop:

```
for (block = 0; (block < number_of_blocks) || my_write_error; block++)
```

and will not execute the next iteration if error has been detected.

Now, if there has been an error, then this information needs to be exchanged with other processes. So we call `MPI_Allreduce` again:

```

/* Check if anybody had problems writing on the file */

MPI_Allreduce (&my_write_error, &write_error, 1, MPI_INT,
               MPI_LOR, MPI_COMM_WORLD);

```

If any instance of `my_write_error` is `TRUE`, then every instance of `write_error` is going to be `TRUE` too.

The way the program handles this condition is to discard all data. After the files have been closed, they are unlinked by this statement:

```

    if ((write_error || file_open_error) && !my_file_open_error) {
        unlink(file_name);
#ifdef DEBUG
        printf("%3d: unlinked %s\n", my_rank, file_name);
#endif
    }

```

After all these travails, whether successful or unsuccessful, the processes meet at `MPI_Finalize` and exit cleanly. The error condition is printed on standard error by the `perror` statements. We could rewrite the program so that an appropriate exit code would be set up too, depending on the type of error encountered.

The program is compiled with the `-DDEBUG CFLAG` so that we can see how it goes about detecting and handling errors. Without this flag the program execution is silent and only the specific error messages are printed on standard error.

So here is the compilation and installation:

```

gustav@bh1 $ pwd
/N/B/gustav/src/I590/mkrandfiles
gustav@bh1 $ make install
co RCS/Makefile,v Makefile
RCS/Makefile,v --> Makefile
revision 1.2
done
co RCS/mkrandfiles.c,v mkrandfiles.c
RCS/mkrandfiles.c,v --> mkrandfiles.c
revision 1.4
done
mpicc -DDEBUG -o mkrandfiles mkrandfiles.c
install mkrandfiles /N/B/gustav/bin
gustav@bh1 $

```

Now let us run a few tests to see how the program is going to behave. First we are going to test for incorrect input on the command line:

```

gustav@bh1 $ mpdboot
gustav@bh1 $ mpdtrace | wc
    32    32   159
gustav@bh1 $ mpiexec -n 16 mkrandfiles
synopsis: mkrandfiles -f <file> -l <blocks>
gustav@bh1 $ mpiexec -n 16 mkrandfiles -f test -l -7
output files basename: test
synopsis: mkrandfiles -f <file> -l <blocks>
gustav@bh1 $

```

Now we are going to check the behaviour of the program if one of the processes fails to open its output file. To do so I am going to create a file `test.7` in my working directory and will set permissions on this file to `-r--r--r--`. Then I'll invoke the program with the `-f test` option:

```

gustav@bh1 $ touch test.7
gustav@bh1 $ chmod 444 test.7
gustav@bh1 $ mpiexec -n 8 mkrandfiles -f test -l 7
output files basename: test
each process will write 7 blocks of integers
 0: basename = test, number_of_blocks = 7
 0: opening file test.0
 4: basename = test, number_of_blocks = 7
 4: opening file test.4
 2: basename = test, number_of_blocks = 7
 2: opening file test.2
 5: basename = test, number_of_blocks = 7
 5: opening file test.5
 3: basename = test, number_of_blocks = 7
 3: opening file test.3
 7: basename = test, number_of_blocks = 7
 7: opening file test.7
 1: basename = test, number_of_blocks = 7
 1: opening file test.1
 7: test.7: Permission denied
 6: basename = test, number_of_blocks = 7
 6: opening file test.6
problem opening output files
 0: closed test.0
 3: closed test.3
 4: closed test.4
 0: unlinked test.0
 4: unlinked test.4
 3: unlinked test.3
 1: closed test.1
 1: unlinked test.1
 6: closed test.6
 2: closed test.2
 5: closed test.5
 6: unlinked test.6
 5: unlinked test.5
 2: unlinked test.2
gustav@bh1 $

```

Every process write quite rich diagnostics on standard output. You can see that they all received the `basename` and `number_of_blocks`. They all managed to construct their file names without problems, but process number 7 failed to open the file, wrote the diagnostics on standard error and passed this information to all other processes, so that the master process could write `problem opening output files` on standard output:

```

 7: opening file test.7
...
 7: test.7: Permission denied
...
problem opening output files

```

Then all processes that managed to open files, i.e., all but process rank 7, close them and then unlink them. After the run there are no files called `test.?` left in the directory with the exception of `test.7`:

```

gustav@bh1 $ ls
PBS bin man mpd.hosts src test.7 tmp
gustav@bh1 $

```

Now let me recompile the program without debugging and we can use it to test IO on the AVIDD GPFS:

```
gustav@bh1 $ make install
co RCS/Makefile,v Makefile
RCS/Makefile,v --> Makefile
revision 1.3
done
co RCS/mkrandfiles.c,v mkrandfiles.c
RCS/mkrandfiles.c,v --> mkrandfiles.c
revision 1.4
done
mpicc -o mkrandfiles mkrandfiles.c
install mkrandfiles /N/B/gustav/bin
gustav@bh1 $
```

This time I am going to run the program on GPFS:

```
gustav@bh1 $ cd /N/gpfs/gustav/mkrandfiles
gustav@bh1 $ mpiexec -n 32 mkrandfiles -f test -l 100
io_time = 202.484912
gustav@bh1 $ ls -l
total 13107200
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:05 test.0
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:05 test.1
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:05 test.10
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:05 test.11
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:05 test.12
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:04 test.13
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:05 test.14
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:05 test.15
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:05 test.16
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:05 test.17
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:05 test.18
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:05 test.19
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:05 test.2
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:05 test.20
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:05 test.21
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:05 test.22
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:05 test.23
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:05 test.24
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:05 test.25
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:05 test.26
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:05 test.27
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:05 test.28
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:05 test.29
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:04 test.3
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:05 test.30
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:05 test.31
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:04 test.4
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:05 test.5
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:05 test.6
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:04 test.7
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:05 test.8
-rw-rw-rw- 1 gustav ucs 419430400 Oct 13 19:05 test.9
gustav@bh1 $
```

The program wrote, in parallel, 32 files, each 400MB long. This is 12.5GB

altogether. The IO itself took 202 seconds, which yields data transfer rate on writes to GPFS of 63MB/s.

Would we be better off using fewer processes perhaps? This is easy to check:

```
gustav@bh1 $ ls
test.0 test.12 test.16 test.2 test.23 test.27 test.30 test.6
test.1 test.13 test.17 test.20 test.24 test.28 test.31 test.7
test.10 test.14 test.18 test.21 test.25 test.29 test.4 test.8
test.11 test.15 test.19 test.22 test.26 test.3 test.5 test.9
gustav@bh1 $ rm *
gustav@bh1 $ mpiexec -n 16 mkrandfiles -f test -l 200
io_time = 226.157398
gustav@bh1 $
```

Well, this is the same amount of data, but it took a little longer, and we got data transfer rate on writes of about 57MB/s this time. But observe the following:

```
gustav@bh1 $ rm -f *
gustav@bh1 $ mpiexec -n 32 mkrandfiles -f test -l 4
io_time = 2.663388
gustav@bh1 $
```

Here we created 32 files, each 16MB long, which is 512MB total, and it took us only 2.663388 seconds to write them. This yields data transfer rate on this parallel write to GPFS of 190MB/s, sic! How would you explain this result?

When we get to study MPI-IO, you will learn how to write data from 32 processes to a single file, shared amongst them all.

## Handling MPI Errors

I have emphasized many times before that an MPI *communicator* is more than just a group of processes that belong to it. The latter is simply a group. But communications do not take place within the group. They take place within the communicator, because one needs more for a communication than just a list of participating processes. Amongst the items that the communicator hides inside its bulbous body is an *error handler*. The error handler is called every time an MPI error is detected within the communicator.

The predefined default error handler, which is called `MPI_ERRORS_ARE_FATAL`, for a newly created communicator or for `MPI_COMM_WORLD` is to *abort the whole parallel program* as soon as any MPI error is detected. Whether an error message is printed or not, and what the error message is, depends on the implementation.

There is another predefined error handler, which is called `MPI_ERRORS_RETURN`. The default error handler can be replaced with this one by calling function `MPIErrhandler_set`, for example:

```
MPIErrhandler_set(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
```

Once you've done this in your MPI code, the program will not longer abort on having detected an MPI error, instead the error will be returned and you will have to handle it.

The returned error code is implementation specific. The only error code that MPI standard itself defines is `MPI_SUCCESS`, i.e., no error. But the meaning of an error code can be extracted by calling function `MPIError_string`.

For example, consider the following code fragment:

```

MPI_Errhandler_set(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
error_code = MPI_Send(send_buffer, strlen(send_buffer) + 1, MPI_CHAR,
                      addressee, tag, MPI_COMM_WORLD);
if (error_code != MPI_SUCCESS) {

    char error_string[BUFSIZ];
    int length_of_error_string;

    MPI_Error_string(error_code, error_string, &length_of_error_string);
    fprintf(stderr, "%3d: %s\n", my_rank, error_string);
    send_error = TRUE;
}

```

On top of the above MPI standard defines the so called *error classes*. Every error code, even the one that is implementation specific, which is every one with the exception of `MPI_SUCCESS`, must belong to some error class, and the error class for a given error code can be obtained by calling function `MPI_Error_class`. Error classes can be converted to comprehensible error messages by calling the same function that does it for error codes, i.e., `MPI_Error_string`. The reason for this is that error classes are implemented as a subset of error codes. Here is the example:

```

MPI_Errhandler_set(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
error_code = MPI_Send(send_buffer, strlen(send_buffer) + 1, MPI_CHAR,
                      addressee, tag, MPI_COMM_WORLD);
if (error_code != MPI_SUCCESS) {

    char error_string[BUFSIZ];
    int length_of_error_string, error_class;

    MPI_Error_class(error_code, &error_class);
    MPI_Error_string(error_class, error_string, &length_of_error_string);
    fprintf(stderr, "%3d: %s\n", my_rank, error_string);
    MPI_Error_string(error_code, error_string, &length_of_error_string);
    fprintf(stderr, "%3d: %s\n", my_rank, error_string);
    send_error = TRUE;
}

```

The idea here is that the error class should give you a general description of the problem, yet it should be precise enough for most debugging purposes, and the error code can then give you an even more precise, implementation specific, diagnostic.

If you have found an MPI error like this in your code, it may be very difficult to recover gracefully. Other than printing the message on standard error and then exiting, or, at best, going right to `MPI_Finalize`, there isn't much that you can do. Sometimes if the problem is, e.g., a receive buffer that is too small, you may be able to allocate a larger buffer dynamically. Your program has to anticipate such events though, and if it does, there are other means of finding how large a buffer you need and avoiding the error altogether.

Perhaps the best use of activating the non-aborting error handler is when you debug the program and try to find where exactly it fails.

Once you have detected the error and are desperate to exit in a controllable way, you can call MPI function `MPI_Abort`, for example:

```

MPI_Errhandler_set(MPI_COMM_WORLD, MPI_ERRORS_RETURN);

```

```

error_code = MPI_Send(send_buffer, strlen(send_buffer) + 1, MPI_CHAR,
                      addressee, tag, MPI_COMM_WORLD);
if (error_code != MPI_SUCCESS) {

    char error_string[BUFSIZ];
    int length_of_error_string, error_class;

    MPI_Error_class(error_code, &error_class);
    MPI_Error_string(error_class, error_string, &length_of_error_string);
    fprintf(stderr, "%3d: %s\n", my_rank, error_string);
    MPI_Error_string(error_code, error_string, &length_of_error_string);
    fprintf(stderr, "%3d: %s\n", my_rank, error_string);
    MPI_Abort(MPI_COMM_WORLD, error_code);
}

```

Each MPI file, which is always associated with a communicator and about which we are going to learn in the next section, has its own separate file handler, which can be altered with the call to function `MPI_File_set_errhandler`. The predefined values for an MPI file error handler are the same as the values for an MPI communicator error handler, i.e., `MPI_ERRORS_ARE_FATAL` and `MPI_ERRORS_RETURN`. However, since file manipulation errors are very common, in this case `MPI_ERRORS_RETURN` is the default.

Apart from communicators and files MPI also supports the so called *windows*. These are *windows of existing memory* that each process *exposes* to direct memory accesses by processes within the communicator. Like MPI files, MPI windows are also associated with MPI communicators. Each MPI window has its own error handler associated with it too and these can be altered by calling function `MPI_Win_set_errhandler`. The predefined values for the windows error handlers are the same as for communicators and files.

### 5.3 MPI IO

MPI-IO was developed in 1994 in the IBM's Watson Laboratory in order to provide parallel I/O support for MPI. NASA adopted MPI-IO for its own research computing projects in 1996 and in the same year MPI Forum decided to incorporate MPI-IO in MPI-2. And so, when MPI-2 was published in 1997, MPI-IO was already in it.

The reason why NASA and MPI Forum embraced MPI-IO so quickly was because it was really very nice – especially if you looked at it from MPI. All MPI-IO function calls are very reminiscent of MPI calls and very much in the spirit of MPI too. Writing MPI files is similar to sending MPI messages and reading MPI files is similar to receiving MPI messages. Furthermore MPI-IO fully embraces the versatility and flexibility of MPI data types – and then takes this concept one step further in defining the so called MPI file views.

Sending and receiving messages can be blocking or non-blocking. In this course, which is introductory, we haven't worked with non-blocking sends and receives, because they are quite difficult to use. But if you want to optimize your parallel program and expect a communication bottleneck, it may help



to use non-blocking communications. The general idea here is that you can start a message send, and then immediately return to computations, while the message is being sent in the background. Similarly you can keep computing while receiving a message in the background. This is going to work best on systems where there are processors that are dedicated to IO and do not in general participate in computations. The IBM BlueGene/L is an example of such a machine.

MPI-IO also lets you write and read files in a normal, i.e., blocking mode, and then in the non-blocking mode – asynchronously – so that you can carry out computations, while the file is being read or written in the background.

MPI-IO supports the concept of collective operations too. Processes can access MPI files each on its own, or all together at the same time. The latter allows for read and write optimizations that can be implemented on various levels.

MPI-IO semantics are so nice that people use MPI-IO even to write normal sequential files associated with individual processes.

In this section we are going to explore MPI-IO beginning with simple parallel writes and reads and then gradually moving to more complex features.

### 5.3.1 Writing on MPI Files

In this section we are going to discuss a program that works much like our previous program `mkrandfiles` discussed in section 5.2.8. But there are a few important changes. First, rather than writing separate files, we are going to open and write the same single file, which will be accessed in parallel by all processes of the MPI process pool. Second, instead of writing the data in small chunks of `BLOCK_SIZE` integers, every process is going to lump all the chunks into a single array and then flush it into the MPI file in one large write.

Once we have concocted and analyzed our example program, we are going to run it and illustrate some of the elementary concepts of MPI-IO. We will use it also to test performance of the AVIDD GPFS.

#### Program `mkrandpfile`

So, here is the listing of the program in full glory.

```
/*
 * %Id: mkrandpfile.c,v 1.13 2003/10/19 19:29:59 gustav Exp %
 *
 * %Log: mkrandpfile.c,v %
 * Revision 1.13 2003/10/19 19:29:59 gustav
 * Indented the file with Emacs.
 *
 * Revision 1.12 2003/10/19 19:26:09 gustav
 * Truncated the log.
 *
 */
```

```

#include <stdio.h> /* all IO stuff lives here */
#include <stdlib.h> /* exit lives here */
#include <unistd.h> /* getopt lives here */
#include <string.h> /* strcpy lives here */
#include <mpi.h> /* MPI and MPI-IO live here */

#define MASTER_RANK 0
#define TRUE 1
#define FALSE 0
#define BOOLEAN int
#define BLOCK_SIZE 1048576
#define MBYTE 1048576
#define SYNOPSIS printf ("synopsis: %s -f <file> -l <blocks>\n", argv[0])

int main(argc, argv)
    int argc;
    char *argv[];
{
    /* my variables */

    int my_rank, pool_size, number_of_blocks = 0, i, count;
    BOOLEAN i_am_the_master = FALSE, input_error = FALSE;
    char *filename = NULL;
    int filename_length;
    int *junk;
    int number_of_integers, number_of_bytes;
    long long total_number_of_integers, total_number_of_bytes;
    MPI_Offset my_offset, my_current_offset;
    MPI_File fh;
    MPI_Status status;
    double start, finish, io_time, longest_io_time;

    /* getopt variables */

    extern char *optarg;
    int c;

    /* ACTION */

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &pool_size);
    if (my_rank == MASTER_RANK) i_am_the_master = TRUE;

    if (i_am_the_master) {

        /* read the command line */

        while ((c = getopt(argc, argv, "f:l:h")) != EOF)
            switch(c) {
                case 'f':
                    filename = optarg;
#ifdef DEBUG
                    printf("output file: %s\n", filename);
#endif
                    break;
                case 'l':

```

```

if ((sscanf (optarg, "%d", &number_of_blocks) != 1) ||
    (number_of_blocks < 1)) {
    SYNOPSIS;
    input_error = TRUE;
}
#ifdef DEBUG
else
    printf("each process will write %d blocks of integers\n",
        number_of_blocks);
#endif
break;
    case 'h':
SYNOPSIS;
input_error = TRUE;
break;
    case '?':
SYNOPSIS;
input_error = TRUE;
break;
    }

/* Check if the command line has initialized filename and
 * number_of_blocks.
 */

if ((filename == NULL) || (number_of_blocks == 0)) {
    SYNOPSIS;
    input_error = TRUE;
}

if (input_error) MPI_Abort(MPI_COMM_WORLD, 1);
/* This is another way of exiting, but it can be done only
   if no files have been opened yet. */

filename_length = strlen(filename) + 1;

} /* end of "if (i_am_the_master)"; reading the command line */

/* If we got this far, the data read from the command line
   should be OK. */

MPI_Bcast(&number_of_blocks, 1, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);
MPI_Bcast(&filename_length, 1, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);
if (! i_am_the_master) filename = (char*) malloc(filename_length);
MPI_Bcast(filename, filename_length, MPI_CHAR, MASTER_RANK, MPI_COMM_WORLD);
#ifdef DEBUG
printf("%3d: received broadcast\n", my_rank);
printf("%3d: filename = %s\n", my_rank, filename);
#endif

number_of_integers = number_of_blocks * BLOCK_SIZE;
number_of_bytes = sizeof(int) * number_of_integers;

/* number_of_bytes must be just plain integer, because we are
   going to use it in malloc */

total_number_of_integers =

```

```

    (long long) pool_size * (long long) number_of_integers;
total_number_of_bytes =
    (long long) pool_size * (long long) number_of_bytes;
my_offset = (long long) my_rank * (long long) number_of_bytes;

#ifdef DEBUG
    if (i_am_the_master) {
        printf("number_of_bytes      = %d/process\n", number_of_bytes);
        printf("total_number_of_bytes = %lld\n", total_number_of_bytes);
        printf("size of offset          = %d bytes\n", sizeof(MPI_Offset));
    }
#endif

    MPI_File_open(MPI_COMM_WORLD, filename,
MPI_MODE_CREATE | MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
    MPI_File_seek(fh, my_offset, MPI_SEEK_SET);
    MPI_File_get_position(fh, &my_current_offset);
#ifdef DEBUG
    printf ("%3d: my current offset is %lld\n", my_rank, my_current_offset);
#endif

    /* generate random integers */

    junk = (int*) malloc(number_of_bytes);
    srand(28 + my_rank);
    for (i = 0; i < number_of_integers; i++) *(junk + i) = rand();

    /* write the stuff out */

    start = MPI_Wtime();
    MPI_File_write(fh, junk, number_of_integers, MPI_INT, &status);
    finish = MPI_Wtime();
    io_time = finish - start;
    MPI_Get_count(&status, MPI_INT, &count);
#ifdef DEBUG
    printf("%3d: wrote %d integers\n", my_rank, count);
#endif
    MPI_File_get_position(fh, &my_current_offset);
#ifdef DEBUG
    printf ("%3d: my current offset is %lld\n", my_rank, my_current_offset);
#endif
    MPI_File_close(&fh);

    MPI_Allreduce(&io_time, &longest_io_time, 1, MPI_DOUBLE, MPI_MAX,
MPI_COMM_WORLD);

    if (i_am_the_master) {
        printf("longest_io_time      = %f seconds\n", longest_io_time);
        printf("total_number_of_bytes = %lld\n", total_number_of_bytes);
        printf("transfer rate         = %f MB/s\n",
total_number_of_bytes / longest_io_time / MBYTE);
    }

    MPI_Finalize();
    exit(0);
}

```

Here is how the program is installed and made:

```

gustav@bh1 $ pwd
/N/B/gustav/src/I590/mkrandpfile
gustav@bh1 $ make install
co RCS/Makefile,v Makefile
RCS/Makefile,v --> Makefile
revision 1.2
done
co RCS/mkrandpfile.c,v mkrandpfile.c
RCS/mkrandpfile.c,v --> mkrandpfile.c
revision 1.10
done
mpicc -DDEBUG -D_FILE_OFFSET_BITS=64 -D_LARGEFILE64_SOURCE -o mkrandpfile mkrandpfile.c
install mkrandpfile /N/B/gustav/bin
gustav@bh1 $

```

Observe the defines, which are added to the CFLAGS:

```
-D_FILE_OFFSET_BITS=64 -D_LARGEFILE64_SOURCE
```

These defines inform the Linux C compiler that we are going to work with files of size that exceeds the ancient UNIX limitation of 2 GB maximum. This limitation is related to the 32-bit architecture of IA32. I have it on explicit advice of ROMIO authors that I should use these switches for all MPI-IO compilations on IA32 systems under Linux, yet they are not MPI-IO specific. They are mentioned, instead, in various system files in `/usr/include`.

And here is how the program is run. The program generates quite a lot of output when run in the DEBUG mode, of course, this apart from the file it writes, which can be very large indeed.

First I want to show you how the program captures error on the command line:

```

gustav@bh1 $ pwd
/N/gpfs/gustav/mkrandpfile
gustav@bh1 $ ls
gustav@bh1 $ mpiexec -n 8 mkrandpfile -l 10
each process will write 10 blocks of integers
synopsis: mkrandpfile -f <file> -l <blocks>
ABORT - process 0: application called MPI_ABORT
rank 0 in job 40 bh1_46074 caused collective abort of all ranks
  exit status of rank 0: return code 1
gustav@bh1 $

```

The ABORT message is triggered by the call to `MPI_Abort`. In this case process of rank 0 is the only one that has called `MPI_Abort`, but this is enough to take all other processes down too. The return code of the master process, i.e., process of rank 0, is returned in the ABORT message.

Now let us create a relatively small 640 MB file with `mkrandpfile`:

```

gustav@bh1 $ mpiexec -n 8 mkrandpfile -f test -l 20
output file: test
each process will write 20 blocks of integers
  0: received broadcast
  0: filename = test
number_of_bytes      = 83886080/process
total_number_of_bytes = 671088640

```

```

size of offset          = 8 bytes
4: received broadcast
4: filename = test
6: received broadcast
6: filename = test
7: received broadcast
7: filename = test
1: received broadcast
1: filename = test
2: received broadcast
2: filename = test
5: received broadcast
5: filename = test
3: received broadcast
3: filename = test
0: my current offset is 0
3: my current offset is 251658240
2: my current offset is 167772160
4: my current offset is 335544320
7: my current offset is 587202560
5: my current offset is 419430400
6: my current offset is 503316480
1: my current offset is 83886080
2: wrote 20971520 integers
2: my current offset is 251658240
5: wrote 20971520 integers
5: my current offset is 503316480
4: wrote 20971520 integers
4: my current offset is 419430400
1: wrote 20971520 integers
1: my current offset is 167772160
6: wrote 20971520 integers
6: my current offset is 587202560
3: wrote 20971520 integers
3: my current offset is 335544320
7: wrote 20971520 integers
7: my current offset is 671088640
0: wrote 20971520 integers
0: my current offset is 83886080
longest_io_time        = 5.781979 seconds
total_number_of_bytes = 671088640
transfer rate          = 110.688746 MB/s
gustav@bh1 $

```

First observe a rather nice IO bandwidth of 110MB/s. The bandwidth is so nice, because the file is small and we end up writing it to memory caches. This program does not flush the file. We will discuss the other messages, about offsets and who wrote how many integers, in the next section.

And now we are going to create a somewhat larger 32GB file. This is going to take a little longer, the program will run on 32 nodes, and I am going to shorten the rather verbose output a little.

```

gustav@bh1 $ pwd
/N/gpfs/gustav/mkrandpfile
gustav@bh1 $ rm test
gustav@bh1 $ mpiexec -n 32 mkrandpfile -f test -l 256

```

```

output file: test
each process will write 256 blocks of integers
number_of_bytes      = 1073741824/process
total_number_of_bytes = 34359738368
size of offset       = 8 bytes

```

```
[...]
```

```

0: my current offset is 0
1: my current offset is 1073741824
2: my current offset is 2147483648
3: my current offset is 3221225472
4: my current offset is 4294967296

```

```
[...]
```

```

0: wrote 268435456 integers
0: my current offset is 1073741824
1: wrote 268435456 integers
1: my current offset is 2147483648
2: wrote 268435456 integers
2: my current offset is 3221225472
3: wrote 268435456 integers
3: my current offset is 4294967296
4: wrote 268435456 integers
4: my current offset is 5368709120

```

```
[...]
```

```

longest_io_time      = 509.006609 seconds
total_number_of_bytes = 34359738368
transfer rate        = 64.376374 MB/s
gustav@bh1 $

```

This time IO has dropped to 64 MB/s, which is what we saw when we had written a lot of data with `mkrandfiles`. This transfer rate is disk array limited. The amount of data, 1GB/node, is too large to just hide in memory buffers.

### The Discussion

This program is very similar to `mkrandfiles`. It begins in the usual MPI-ish way, i.e., all processes find about the size of the pool and their own rank number within it. Then process of rank 0 assumes the mastership (which usually just means more work and not much more pay) and reads the command line.

On having detected a command line input error, the master process calls `MPI_Abort`:

```
if (input_error) MPI_Abort(MPI_COMM_WORLD, 1);
```

and this takes every other process down. We can quit the program in such an abrupt manner in this place because we haven't opened any files yet. This is important. Otherwise, we should really postpone aborting and clean up the mess first. But in this first MPI-IO example of ours we are not going to be particularly fastidious about error handling. This will come later.

If there are no problems with the command line, the master process broadcasts (1) the number of blocks of random integers each process is going to contribute to the file, (2) the length of the file name, which includes also the space for the string termination character.

Having received the latter each process, with the exception of the master process, calls `malloc` to allocate enough space for the string. Observe that the master process never had to `malloc` space for the string explicitly. This was done by function `getopt` internally, when it created the string `optarg`. Then the master process merely made its own instance of `filename` *point* to the same location to which `optarg` pointed.

Finally, the master process broadcasts (3) the name of the file to other processes.

In the next part of the code:

```
number_of_integers = number_of_blocks * BLOCK_SIZE;
number_of_bytes = sizeof(int) * number_of_integers;
```

each process calculates the number of random integers it is going to write and the number of bytes it will need to store *all* these integers in its memory. Here this program differs a little from `mkrandfiles`. Instead of writing in numerous small chunks, we are going to prepare all number in memory first, and then write the whole lot in a single operation. We will have to allocate sufficient space for the numbers, and so `number_of_bytes` will become an argument to `malloc`. This argument *must* be an integer. You cannot `malloc` a `long long` of bytes. UNIX does not support memory above `MAX_INT`.

But the next three numbers that are computed here:

```
total_number_of_integers =
    (long long) pool_size * (long long) number_of_integers;
total_number_of_bytes =
    (long long) pool_size * (long long) number_of_bytes;
my_offset = (long long) my_rank * (long long) number_of_bytes;
```

are all of type `long long`, which on the IA32 is a 64-bit integer. This is because the total number of bytes written on the file may well exceed the total amount of memory available to a single process, on account of there being many processes in the pool.

The last number, `my_offset`, will be used to point to a location in our MPI file, which, in general, is going to be longer than 2 GB. The variables `my_offset` and its sibling `my_current_offset` are of type:

```
MPI_Offset my_offset, my_current_offset;
```

You have to look up the meaning of this type in `/N/hpc/mpich2/include/mpi.h` in order to find out that it is `long long` on the AVIDD system. It doesn't always have to be `long long` though. It may well be `long` or just `int`, depending on how MPI was compiled and what machine it runs on.

Usually you should be able to just refer to this type knowing only that it is an integer of some opaque length. And so we could write:

```
total_number_of_integers =
    (MPI_Offset) pool_size * (MPI_Offset) number_of_integers;
```



```
total_number_of_bytes =
    (MPI_Offset) pool_size * (MPI_Offset) number_of_bytes;
my_offset = (MPI_Offset) my_rank * (MPI_Offset) number_of_bytes;
```

But if `MPI_Offset` is not long enough, you will not be able to generate truly large MPI files. And if you don't know what it is, you may have problems writing values of file pointer offsets on standard output, be it for debugging or for other purposes, although there is a macro defined on `mpio.h`, which is included in `mpi.h`:

```
#define LL %lld
```

and you may be able to use it in calls to `printf`.

Now we encounter the first MPI-IO function, `MPIFile_open`:

```
MPI_File_open(MPI_COMM_WORLD, filename,
              MPI_MODE_CREATE | MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
```

This function opens a file identified by `filename` on all processes in the `MPI_COMM_WORLD` communicator. This is a *collective* function meaning that all processes must do it together and the values of all parameters passed to it must be identical on all processes too.

The third parameter specifies how the file should be opened, e.g., for writing, reading or both, and whether it should be created if it doesn't exist. The modes supported by MPI-2 are as follows:

**MPI\_MODE\_RDONLY** open the file for reading from it only

**MPI\_MODE\_RDWR** open the file for reading and writing

**MPI\_MODE\_WRONLY** open the file for writing to it only

**MPI\_MODE\_CREATE** create the file if it does not exist

**MPI\_MODE\_EXCL** throw an error if you try to create a file that exists already

**MPI\_MODE\_DELETE\_ON\_CLOSE** delete the file on close – such files are called scratch files and they are used for auxiliary data storage during computations

**MPI\_MODE\_UNIQUE\_OPEN** ensure that the file is not going to be opened concurrently elsewhere (e.g., by another communicator)

**MPI\_MODE\_SEQUENTIAL** the file will be accessed sequentially only

**MPI\_MODE\_APPEND** set initial position of all file pointers to the end of the file

The options can be combined with the boolean “or”, `|`, operator.

The fourth parameter can be used to give the operating system additional hints about how and where the file should be opened. For example, our current version of HPSS lets SP users open HPSS files using MPI-IO. If you wanted

to tell HPSS which class of service the file should be associated with, what annotation string it should have attached to its HPSS data base record, and what ACLs it should have, you would use the `info` structure to pass all this information. In this case though we don't pass any such data to GPFS and so we use one of the predefined `infos`, which is `MPI_INFO_NULL`.

On successful completion `MPI_Open_file` returns a *file handle* on `fh`. This is *not* the same as a file pointer. It is used a little differently and you always have to check meticulously, whether an MPI-IO function you call wants a file handle (i.e., the value of) or a pointer to it. Another thing that happens is that *every* process in the `MPI_COMM_WORLD` communicator gets its own local pointer to the file and all those pointers point to the beginning of the file, unless the `MPI_MODE_APPEND` option has been used.

Having opened the file, collectively, each process is now going to advance to its own position within it by calling `MPI_File_seek`

```
MPI_File_seek(fh, my_offset, MPI_SEEK_SET);
```

This is where we use the `my_offset` variable, which is of type `MPI_Offset`. The value of this variable is different for each process, so that when they get to write the data, they'll write it on different portions of the file and without overwriting each-other's territory.

The seek can be performed in one of three ways:

**MPI\_SEEK\_SET** the pointer is set exactly to `my_offset`

**MPI\_SEEK\_CUR** the pointer is *advanced* by `my_offset` from its current position

**MPI\_SEEK\_END** the pointer is *advanced* by `my_offset` from the end of the file

In general `my_offset` can be positive and negative, i.e., you can use it to move in both directions within the file.

In this program we set the pointer explicitly. There are other ways to do this and we'll learn about them later. Whenever you manipulate a pointer explicitly, especially in a parallel program, you must exercise great caution and make sure that you point exactly where you want to point. Otherwise you may end up overwriting your own data. You can check where your pointers are by calling function `MPI_File_get_position`:

```
MPI_File_get_position(fh, &my_current_offset);
```

In this program we are going to call this function before and after the write, in order to see how each of the individual file pointer has advanced.

Having opened the file and positioned themselves within it, the processes allocate space for the random integers they are about to write and then generate them:

```
junk = (int*) malloc(number_of_bytes);
srand(28 + my_rank);
for (i = 0; i < number_of_integers; i++) *(junk + i) = rand();
```

Now we are ready to perform the write itself, and we time it too:

```

start = MPI_Wtime();
MPI_File_write(fh, junk, number_of_integers, MPI_INT, &status);
finish = MPI_Wtime();
io_time = finish - start;

```

Function `MPI_File_write` writes a `number_of_integers` of objects of type `MPI_INT` taken from the buffer pointed to by `junk` on a file whose file handle is `fh`. The write, for a given process, commences at the place where its file pointer points and the file pointer is advanced as the writing proceeds.

We can check how many items have indeed been written by inspecting `status` with function `MPI_Get_count`.

```

MPI_Get_count(&status, MPI_INT, &count);

```

The variable `status` is of type `MPI_Status`, and it is the same *status* that is returned, e.g., by `MPI_Recv`. You can see here how nicely MPI-IO fits with MPI.

After the write we call `MPI_File_get_position` again:

```

MPI_File_get_position(fh, &my_current_offset);

```

and the new positions are printed on standard output. Recall the following:

```

0: my current offset is 0
1: my current offset is 1073741824
2: my current offset is 2147483648
3: my current offset is 3221225472
4: my current offset is 4294967296

```

[...]

```

0: wrote 268435456 integers
0: my current offset is 1073741824
1: wrote 268435456 integers
1: my current offset is 2147483648
2: wrote 268435456 integers
2: my current offset is 3221225472
3: wrote 268435456 integers
3: my current offset is 4294967296
4: wrote 268435456 integers
4: my current offset is 5368709120

```

You can see that the initial offset of process 0 was 0 and after the write it is 1073741824. But this was the initial offset of process 1. Whereas after the write process 1 advanced to 2147483648. But this was the initial offset of process 2, which after the write advanced to 3221225472... and so on. The file has indeed been written rather tightly. Each process wrote on its own portion of it.

Now we can close the file by calling `MPI_File_close`

```

MPI_File_close(&fh);

```

This file merely disposes of the file handle. Once you have closed the file, you can no longer refer to it by using `fh`.

The last part of the program calls `MPI_Allreduce` to find the longest IO time that any process spent writing the data, and this time is then used by the master process to estimate the data transfer rate:

```

MPI_Allreduce(&io_time, &longest_io_time, 1, MPI_DOUBLE, MPI_MAX,
              MPI_COMM_WORLD);

if (i_am_the_master) {
    printf("longest_io_time      = %f seconds\n", longest_io_time);
    printf("total_number_of_bytes = %lld\n", total_number_of_bytes);
    printf("transfer rate        = %f MB/s\n",
           total_number_of_bytes / longest_io_time / MBYTE);
}

```

whereupon all processes meet at `MPI_Finalize` and exit.

### Exercises

- 1 Rewrite the program that simulated interactions between charged particles, discussed in section 5.2.6 so that it dumps states of all particles after every time step. Make each process write only the states of its own particles on the common MPI file.

### 5.3.2 Reading from MPI Files

In this section I am going to show you how to read MPI files. This can be a little tricky. You will also see how MPI-IO errors can be handled

#### Program `xrandpfile`

This is the example program. It opens a file whose name has been provided on the command line. Then it finds about its length. Every process evaluates the amount of data it's going to read by dividing the size of the file by the number of processes in the pool, and allocates sufficient memory for the data. The reading itself is timed.

If any process has problems opening the file, the program aborts.

```

/*
 * %Id: xrandpfile.c,v 1.4 2003/10/18 21:43:05 gustav Exp %
 *
 * %Log: xrandpfile.c,v %
 * Revision 1.4 2003/10/18 21:43:05 gustav
 * integers -> bytes
 *
 * Revision 1.3 2003/10/18 21:33:35 gustav
 * Indented the program (used emacs).
 *
 * Revision 1.2 2003/10/18 20:43:24 gustav
 * Added reading of status with MPI_Get_count
 *
 * Revision 1.1 2003/10/18 19:52:44 gustav
 * Initial revision
 *
 */

#include <stdio.h> /* all IO stuff lives here */

```

```

#include <stdlib.h> /* exit lives here */
#include <unistd.h> /* getopt lives here */
#include <string.h> /* strcpy lives here */
#include <limits.h> /* INT_MAX lives here */
#include <mpi.h> /* MPI and MPI-IO live here */

#define MASTER_RANK 0
#define TRUE 1
#define FALSE 0
#define BOOLEAN int
#define MBYTE 1048576
#define SYNOPSIS printf ("synopsis: %s -f <file>\n", argv[0])

int main(argc, argv)
    int argc;
    char *argv[];
{
    /* my variables */

    int my_rank, pool_size, last_guy, i, count;
    BOOLEAN i_am_the_master = FALSE, input_error = FALSE;
    char *filename = NULL, *read_buffer;
    int filename_length;
    int *junk;
    int file_open_error, number_of_bytes;

    /* MPI_Offset is long long */

    MPI_Offset my_offset, my_current_offset, total_number_of_bytes,
        number_of_bytes_ll, max_number_of_bytes_ll;
    MPI_File fh;
    MPI_Status status;
    double start, finish, io_time, longest_io_time;

    /* getopt variables */

    extern char *optarg;
    int c;

    /* ACTION */

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &pool_size);
    last_guy = pool_size - 1;
    if (my_rank == MASTER_RANK) i_am_the_master = TRUE;

    if (i_am_the_master) {

        /* read the command line */

        while ((c = getopt(argc, argv, "f:h")) != EOF)
            switch(c) {
                case 'f':
filename = optarg;
#ifdef DEBUG
printf("input file: %s\n", filename);

```

```

#endif
break;
    case 'h':
SYNOPSIS;
input_error = TRUE;
break;
    case '?':
SYNOPSIS;
input_error = TRUE;
break;
} /* end of switch(c) */

/* Check if the command line has initialized filename and
 * number_of_blocks.
 */

if (filename == NULL) {
    SYNOPSIS;
    input_error = TRUE;
}

if (input_error) MPI_Abort(MPI_COMM_WORLD, 1);

filename_length = strlen(filename) + 1;

/* This is another way of exiting, but it can be done only
 * if no files have been opened yet. */

} /* end of "if (i_am_the_master)"; reading the command line */

/* If we got this far, the data read from the command line
 * should be OK. */

MPI_Bcast(&filename_length, 1, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);
if (! i_am_the_master) filename = (char*) malloc(filename_length);
#ifdef DEBUG
    printf("%3d: allocated space for filename\n", my_rank);
#endif
MPI_Bcast(filename, filename_length, MPI_CHAR, MASTER_RANK, MPI_COMM_WORLD);
#ifdef DEBUG
    printf("%3d: received broadcast\n", my_rank);
    printf("%3d: filename = %s\n", my_rank, filename);
#endif

MPI_Barrier(MPI_COMM_WORLD);

/* Default I/O error handling is MPI_ERRORS_RETURN */

file_open_error = MPI_File_open(MPI_COMM_WORLD, filename,
                                MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);

if (file_open_error != MPI_SUCCESS) {

    char error_string[BUFSIZ];
    int length_of_error_string, error_class;

    MPI_Error_class(file_open_error, &error_class);

```

```

    MPI_Error_string(error_class, error_string, &length_of_error_string);
    printf("%3d: %s\n", my_rank, error_string);

    MPI_Error_string(file_open_error, error_string, &length_of_error_string);
    printf("%3d: %s\n", my_rank, error_string);

    MPI_Abort(MPI_COMM_WORLD, file_open_error);
}

MPI_File_get_size(fh, &total_number_of_bytes);
#ifdef DEBUG
    printf("%3d: total_number_of_bytes = %lld\n", my_rank, total_number_of_bytes);
#endif

number_of_bytes_ll = total_number_of_bytes / pool_size;

/* If pool_size does not divide total_number_of_bytes evenly,
   the last process will have to read more data, i.e., to the
   end of the file. */

max_number_of_bytes_ll =
    number_of_bytes_ll + total_number_of_bytes % pool_size;

if (max_number_of_bytes_ll < INT_MAX) {

    if (my_rank == last_guy)
        number_of_bytes = (int) max_number_of_bytes_ll;
    else
        number_of_bytes = (int) number_of_bytes_ll;

    read_buffer = (char*) malloc(number_of_bytes);
#ifdef DEBUG
    printf("%3d: allocated %d bytes\n", my_rank, number_of_bytes);
#endif

    my_offset = (MPI_Offset) my_rank * number_of_bytes_ll;
#ifdef DEBUG
    printf("%3d: my offset = %lld\n", my_rank, my_offset);
#endif
    MPI_File_seek(fh, my_offset, MPI_SEEK_SET);

    MPI_Barrier(MPI_COMM_WORLD);

    start = MPI_Wtime();
    MPI_File_read(fh, read_buffer, number_of_bytes, MPI_BYTE, &status);
    finish = MPI_Wtime();
    MPI_Get_count(&status, MPI_BYTE, &count);
#ifdef DEBUG
    printf("%3d: read %d bytes\n", my_rank, count);
#endif
    MPI_File_get_position(fh, &my_offset);
#ifdef DEBUG
    printf("%3d: my offset = %lld\n", my_rank, my_offset);
#endif

    io_time = finish - start;
    MPI_Allreduce(&io_time, &longest_io_time, 1, MPI_DOUBLE, MPI_MAX,

```

```

MPI_COMM_WORLD);
    if (i_am_the_master) {
        printf("longest_io_time      = %f seconds\n", longest_io_time);
        printf("total_number_of_bytes = %lld\n", total_number_of_bytes);
        printf("transfer rate        = %f MB/s\n",
              total_number_of_bytes / longest_io_time / MBYTE);
    }
}
else {
    if (i_am_the_master) {
        printf("Not enough memory to read the file.\n");
        printf("Consider running on more nodes.\n");
    }
} /* of if(max_number_of_bytes_ll < INT_MAX) */

MPI_File_close(&fh);

MPI_Finalize();
exit(0);
}

```

The program is made much the same as its sibling `mkrandpfile`:

```

/N/B/gustav/src/I590/xrandpfile
gustav@bh1 $ make install
co RCS/Makefile,v Makefile
RCS/Makefile,v --> Makefile
revision 1.1
done
co RCS/xrandpfile.c,v xrandpfile.c
RCS/xrandpfile.c,v --> xrandpfile.c
revision 1.3
done
mpicc -DDEBUG -D_FILE_OFFSET_BITS=64 -D_LARGEFILE64_SOURCE -o xrandpfile xrandpfile.c
install xrandpfile /N/B/gustav/bin
gustav@bh1 $

```

Now let us run this program. But first I want to show you how capturing MPI file errors works. I will ask the program to open a non-existent file for reading.

```

gustav@bh1 $ pwd
/N/gpfs/gustav/mkrandpfile
gustav@bh1 $ ls
test
gustav@bh1 $ mpiexec -n 8 xrandpfile -f junk
input file: junk
0: allocated space for filename
0: received broadcast
0: filename = junk
1: allocated space for filename
1: received broadcast
1: filename = junk
2: allocated space for filename
6: allocated space for filename
2: received broadcast
2: filename = junk
5: allocated space for filename

```



```

6: received broadcast
6: filename = junk
4: allocated space for filename
3: allocated space for filename
4: received broadcast
4: filename = junk
7: allocated space for filename
3: received broadcast
3: filename = junk
7: received broadcast
7: filename = junk
5: received broadcast
5: filename = junk
ABORT - process 0: application called MPI_ABORT
ABORT - process 1: application called MPI_ABORT
ABORT - process 3: application called MPI_ABORT
ABORT - process 7: application called MPI_ABORT
ABORT - process 4: application called MPI_ABORT
0: Other I/O error
0: Other I/O error No such file or directory
1: Other I/O error
1: Other I/O error No such file or directory
3: Other I/O error
3: Other I/O error No such file or directory
ABORT - process 2: application called MPI_ABORT
ABORT - process 5: application called MPI_ABORT
ABORT - process 6: application called MPI_ABORT
2: Other I/O error
2: Other I/O error No such file or directory
5: Other I/O error
5: Other I/O error No such file or directory
6: Other I/O error
6: Other I/O error No such file or directory
4: Other I/O error
4: Other I/O error No such file or directory
7: Other I/O error
7: Other I/O error No such file or directory
rank 7 in job 43 bh1_46074 caused collective abort of all ranks
exit status of rank 7: return code 32
rank 6 in job 43 bh1_46074 caused collective abort of all ranks
exit status of rank 6: return code 32
rank 5 in job 43 bh1_46074 caused collective abort of all ranks
exit status of rank 5: return code 32
rank 4 in job 43 bh1_46074 caused collective abort of all ranks
exit status of rank 4: return code 32
rank 3 in job 43 bh1_46074 caused collective abort of all ranks
exit status of rank 3: return code 32
rank 2 in job 43 bh1_46074 caused collective abort of all ranks
exit status of rank 2: return code 32
rank 1 in job 43 bh1_46074 caused collective abort of all ranks
exit status of rank 1: return code 32
rank 0 in job 43 bh1_46074 caused collective abort of all ranks
exit status of rank 0: return code 32
gustav@bh1 $

```

Observe that since *every* process captures the open error, every process issues the `MPI_Abort` call. The error class corresponds to the message:

```
Other I/O error
```

and the specific error is:

```
Other I/O error No such file or directory
```

And now let us read file `test` created with `mkrandpfile` with `xrandpfile`. This file is 32GB long!

```
gustav@bh1 $ pwd
/N/gpfs/gustav/mkrandpfile
gustav@bh1 $ ls -l
total 33554432
-rw-rw-rw-  1 gustav  ucs      34359738368 Oct 18 16:12 test
gustav@bh1 $ mpiexec -n 32 xrandpfile -f test
```

[... there some simple diagnostic output here pertaining to the file name ...]

```
0: total_number_of_bytes = 34359738368
0: allocated 1073741824 bytes
0: my offset = 0
1: total_number_of_bytes = 34359738368
1: allocated 1073741824 bytes
1: my offset = 1073741824
2: total_number_of_bytes = 34359738368
2: allocated 1073741824 bytes
2: my offset = 2147483648
3: total_number_of_bytes = 34359738368
3: allocated 1073741824 bytes
3: my offset = 3221225472
4: total_number_of_bytes = 34359738368
4: allocated 1073741824 bytes
4: my offset = 4294967296
```

[...]

```
0: read 1073741824 bytes
0: my offset = 1073741824
1: read 1073741824 bytes
1: my offset = 2147483648
2: read 1073741824 bytes
2: my offset = 3221225472
3: read 1073741824 bytes
3: my offset = 4294967296
4: read 1073741824 bytes
4: my offset = 5368709120
```

[...]

```
longest_io_time      = 193.654002 seconds
total_number_of_bytes = 34359738368
transfer rate        = 169.209000 MB/s
20: read 1073741824 bytes
20: my offset = 22548578304
gustav@bh1 $
```

Here the transfer rate of nearly 170 MB/s is much better than the 64 MB/s we saw when `mkrandpfile` wrote the 32 GB of `test`. Neither of these two numbers is likely to get any better with the current generation and configuration of the

AVIDD disk arrays. *This is it.* This large discrepancy between reads and writes illustrates very aptly how much slower physical writes on the magnetic media are from reads. Optical media, such as CDs and DVDs, are even worse in this respect. But the discrepancy would not be this large on, e.g., magnetic tapes, because a magnetic tape is a streaming medium. What makes writes to disk arrays so slow, apart from the physical process itself, is having to find space for the writes. Once it's been found, the head has to do quite a lot of moving around as data is written on the disk. File `test` is not stored on the AVIDD disk arrays contiguously.

Now let me show you on a smaller example the case when the total length of the file does not divide by the number of processes evenly. In this case one of the processes will have to read a little bit more.

```
gustav@bh1 $ mpiexec -n 4 mkrandpfile -f small_test -l 4
output file: small_test
each process will write 4 blocks of integers
 0: received broadcast
 0: filename = small_test
number_of_bytes      = 16777216/process
total_number_of_bytes = 67108864
size of offset       = 8 bytes
 2: received broadcast
 2: filename = small_test
 1: received broadcast
 1: filename = small_test
 3: received broadcast
 3: filename = small_test
 0: my current offset is 0
 1: my current offset is 16777216
 2: my current offset is 33554432
 3: my current offset is 50331648
 2: wrote 4194304 integers
 2: my current offset is 50331648
 3: wrote 4194304 integers
 3: my current offset is 67108864
 0: wrote 4194304 integers
 0: my current offset is 16777216
 1: wrote 4194304 integers
 1: my current offset is 33554432
longest_io_time      = 1.453121 seconds
total_number_of_bytes = 67108864
transfer rate        = 44.043134 MB/s
gustav@bh1 $ mpiexec -n 3 xrandpfile -f small_test
input file: small_test
 2: allocated space for filename
 0: allocated space for filename
 0: received broadcast
 0: filename = small_test
 1: allocated space for filename
 1: received broadcast
 1: filename = small_test
 2: received broadcast
 2: filename = small_test
 0: total_number_of_bytes = 67108864
 0: allocated 22369621 bytes
 0: my offset = 0
```

```

2: total_number_of_bytes = 67108864
2: allocated 22369622 bytes
2: my offset = 44739242
1: total_number_of_bytes = 67108864
1: allocated 22369621 bytes
1: my offset = 22369621
1: read 22369621 bytes
1: my offset = 44739242
2: read 22369622 bytes
2: my offset = 67108864
0: read 22369621 bytes
0: my offset = 22369621
longest_io_time      = 0.167047 seconds
total_number_of_bytes = 67108864
transfer rate        = 383.125653 MB/s
gustav@bh1 $

```

Observe that process number 2 reads 22369622 bytes, whereas processes 0 and 1 read 22369621 bytes. In this case  $3 \times 22369621 = 67108863$ , which is one byte less than the length of the file, 67108864 bytes. So process number 2 has to stretch just this little byte farther.

### The Discussion

This program begins the same way `mkrandpfile` did, until we get to

```

file_open_error = MPI_File_open(MPI_COMM_WORLD, filename,
                               MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);

if (file_open_error != MPI_SUCCESS) {

    char error_string[BUFSIZ];
    int length_of_error_string, error_class;

    MPI_Error_class(file_open_error, &error_class);
    MPI_Error_string(error_class, error_string, &length_of_error_string);
    printf("%3d: %s\n", my_rank, error_string);

    MPI_Error_string(file_open_error, error_string, &length_of_error_string);
    printf("%3d: %s\n", my_rank, error_string);

    MPI_Abort(MPI_COMM_WORLD, file_open_error);
}

```

This time we open the file for reading only and we check what function `MPI_File_open` has returned. If there is no problem, i.e., `file_open_error == MPI_SUCCESS`, then we go ahead and read the file. But if there is a problem, we convert `file_open_error` to error messages, print them on standard output and `MPI_Abort`.

Assuming that the `MPI_File_open` worked, we need to find out how much data has to be read. So we check the size of the file by calling `MPI_File_get_size`

```
MPI_File_get_size(fh, &total_number_of_bytes);
```

where `total_number_of_bytes` must be of type `MPI_Offset`, i.e., in our case, `long long`.

Now we evaluate how much data every process needs to read:

```

number_of_bytes_ll = total_number_of_bytes / pool_size;

/* If pool_size does not divide total_number_of_bytes evenly,
   the last process will have to read more data, i.e., to the
   end of the file. */

max_number_of_bytes_ll =
    number_of_bytes_ll + total_number_of_bytes % pool_size;

```

Depending on the length of the file and the number of processes, the division of the former by the latter may or may not be exact. If it isn't then `max_number_of_bytes_ll` is going to be a little larger than `number_of_bytes_ll`. We will make the last process read more. Observe that both `number_of_bytes_ll` and `max_number_of_bytes_ll` are long long. At this stage we don't know if they'll fit in `int`.

Now we have the `if` statement:

```

if (max_number_of_bytes_ll < INT_MAX) {
    blah... blah... blah...
}
else {
    if (i_am_the_master) {
        printf("Not enough memory to read the file.\n");
        printf("Consider running on more nodes.\n");
    }
} /* of if(max_number_of_bytes_ll < INT_MAX) */

MPI_File_close(&fh);

```

This statement checks, right at the top, if `max_number_of_bytes_ll` is going to fit into `int`, because we are going to read the data the same way we wrote it, i.e., in one large gasp into a single sufficiently long array. If `max_number_of_bytes_ll` is too large, then we close the file right away.

Now let's see what happens inside the top clause of the `if` statement.

First each process converts `number_of_bytes_ll` to a normal integer suitable for passing to `malloc` with the exception of the last process, which does it to `max_number_of_bytes_ll`, and then they all call `malloc`:

```

if (my_rank == last_guy)
    number_of_bytes = (int) max_number_of_bytes_ll;
else
    number_of_bytes = (int) number_of_bytes_ll;

read_buffer = (char*) malloc(number_of_bytes);

```

Now every process figures out its own offset in the file and goes there:

```

my_offset = (MPI_Offset) my_rank * number_of_bytes_ll;
#ifdef DEBUG
    printf("%3d: my offset = %lld\n", my_rank, my_offset);
#endif
MPI_File_seek(fh, my_offset, MPI_SEEK_SET);

MPI_Barrier(MPI_COMM_WORLD);

```

and then they all meet at the barrier.

Now we are ready to commence the read, to time it, and to find if and how the pointers have advanced as the result of it:

```

    start = MPI_Wtime();
    MPI_File_read(fh, read_buffer, number_of_bytes, MPI_BYTE, &status);
    finish = MPI_Wtime();
    MPI_Get_count(&status, MPI_BYTE, &count);
#ifdef DEBUG
    printf("%3d: read %d bytes\n", my_rank, count);
#endif
    MPI_File_get_position(fh, &my_offset);
#ifdef DEBUG
    printf("%3d: my offset = %lld\n", my_rank, my_offset);
#endif

```

Function `MPI_File_read` read `number_of_bytes` of items of type `MPI_BYTE` into the `read_buffer` from the file given by the file handle `fh`. Every process reads the data beginning from the position it is at as the result of the call to `MPI_File_seek`, and as the reading progresses, its own pointer moves accordingly.

Let us have a look at the positions of the pointers before and after the reading:

```

0: total_number_of_bytes = 34359738368
0: allocated 1073741824 bytes
0: my offset = 0
1: total_number_of_bytes = 34359738368
1: allocated 1073741824 bytes
1: my offset = 1073741824
2: total_number_of_bytes = 34359738368
2: allocated 1073741824 bytes
2: my offset = 2147483648
3: total_number_of_bytes = 34359738368
3: allocated 1073741824 bytes
3: my offset = 3221225472
4: total_number_of_bytes = 34359738368
4: allocated 1073741824 bytes
4: my offset = 4294967296

```

[...]

```

0: read 1073741824 bytes
0: my offset = 1073741824
1: read 1073741824 bytes
1: my offset = 2147483648
2: read 1073741824 bytes
2: my offset = 3221225472
3: read 1073741824 bytes
3: my offset = 4294967296
4: read 1073741824 bytes
4: my offset = 5368709120

```

Observe that process 0 started at offset 0 and progressed to offset 1073741824 having read exactly 1073741824 bytes. Process 1 started at offset 1073741824 and progressed to offset 2147483648, which is exactly where process 2 started from. In short, we have read every byte from the file, not missing anything, not even the last couple of bytes, in case the length of the file does not divide by the number of processes. The last process is going to mop them up.

Now we check what the bandwidth was the same way we did it for `mkrandpfile`:

```

io_time = finish - start;
MPI_Allreduce(&io_time, &longest_io_time, 1, MPI_DOUBLE, MPI_MAX,
             MPI_COMM_WORLD);
if (i_am_the_master) {
    printf("longest_io_time      = %f seconds\n", longest_io_time);
    printf("total_number_of_bytes = %lld\n", total_number_of_bytes);
    printf("transfer rate        = %f MB/s\n",
           total_number_of_bytes / longest_io_time / MBYTE);
}

```

And this is it. The processes all go to `MPI_Finalize` and exit.

### Exercises

- 1 Rewrite the program that simulated interactions between charged particles, discussed in section 5.2.6 so that it reads initial states of particles from an MPI file if requested. Make each process read only the states of its own particles from the MPI file.

### 5.3.3 Writing Sequential Files with MPI-IO

I have already mentioned that the semantics of MPI-IO are so nice, that it is tempting to use them even for ordinary sequential files. This program is almost the same as `mkrandfiles`, but it uses MPI-IO to write the files. It does more error checking than my previous MPI-IO examples, testing for potential errors both on open and on write. Any problems are handled without calling `MPI_Abort`. Where `mkrandfiles` called `unlink` to delete files in case there were problems either with open or with write, this program calls `MPI_File_Delete`.

#### Program `mkrandpfiles`

Here is the program itself:

```

/*
 * %Id: mkrandpfiles.c,v 1.2 2003/10/19 19:20:14 gustav Exp %
 *
 * %Log: mkrandpfiles.c,v %
 * Revision 1.2 2003/10/19 19:20:14 gustav
 * Indented the program with Emacs.
 *
 * Revision 1.1 2003/10/19 18:51:41 gustav
 * Initial revision
 *
 */

#include <stdio.h> /* all IO stuff lives here */
#include <stdlib.h> /* exit lives here */
#include <unistd.h> /* getopt lives here */
#include <errno.h> /* UNIX error handling lives here */
#include <string.h> /* strcpy lives here */
#include <mpi.h> /* MPI and MPI-IO live here */

```

```

#define MASTER_RANK 0
#define TRUE 1
#define FALSE 0
#define BOOLEAN int
#define BLOCK_SIZE 1048576
#define MBYTE 1048576
#define SYNOPSIS printf ("synopsis: %s -f <file> -l <blocks>\n", argv[0])

int main(argc, argv)
    int argc;
    char *argv[];
{
    /* my variables */

    int my_rank, pool_size, number_of_blocks = 0, i;
    int number_of_integers, number_of_bytes;
    long long total_number_of_integers, total_number_of_bytes;
    BOOLEAN i_am_the_master = FALSE, input_error = FALSE,
        my_file_open_error = FALSE, file_open_error = FALSE,
        my_write_error = FALSE, write_error = FALSE;
    char *basename = NULL, file_name[BUFSIZ], message[BUFSIZ];
    int basename_length, *junk;
    MPI_File fh;
    double start, finish, io_time, longest_io_time;
    char error_string[BUFSIZ];
    int length_of_error_string, error_class;
    MPI_Status status;

    /* getopt variables */

    extern char *optarg;
    int c;

    /* error handling variables */

    extern int errno;

    /* ACTION */

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &pool_size);
    if (my_rank == MASTER_RANK) i_am_the_master = TRUE;

    if (i_am_the_master) {

        /* read the command line */

        while ((c = getopt(argc, argv, "f:l:h")) != EOF)
            switch(c) {
                case 'f':
                    basename = optarg;
            break;
                case 'l':
            if ((sscanf(optarg, "%d", &number_of_blocks) != 1) ||
                (number_of_blocks < 1))
                input_error = TRUE;
            }
    }
}

```



```

break;
    case 'h':
input_error = TRUE;
break;
    case '?':
input_error = TRUE;
break;
    }

/* Check if the command line has initialized both the basename and
 * the number_of_blocks.
 */

if ((basename == NULL) || (number_of_blocks == 0)) input_error = TRUE;

if (input_error)
    SYNOPSIS;
else {
    basename_length = strlen(basename) + 1;
#ifdef DEBUG
    printf("basename          = %s\n", basename);
    printf("basename_length   = %d\n", basename_length);
    printf("number_of_blocks = %d\n", number_of_blocks);
#endif
}
} /* end of if(i_am_the_master) { <read the command line> } */

/* Transmit the effect of reading the command line to other
processes. */

MPI_Bcast(&input_error, 1, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);

if (! input_error) {

    /* If we managed to get here, data read from the command line
    is probably OK. */

    MPI_Bcast(&number_of_blocks, 1, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);
    MPI_Bcast(&basename_length, 1, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);
    if (! i_am_the_master) basename = (char*) malloc(basename_length);
    MPI_Bcast(basename, basename_length, MPI_CHAR, MASTER_RANK, MPI_COMM_WORLD);

#ifdef DEBUG
    printf("%3d: basename = %s, number_of_blocks = %d\n",
        my_rank, basename, number_of_blocks);
#endif

    /* Allocate space needed to generate the integers */

    number_of_integers = number_of_blocks * BLOCK_SIZE;
    number_of_bytes = sizeof(int) * number_of_integers;
    total_number_of_integers = (long long) number_of_integers
        * (long long) pool_size;
    total_number_of_bytes = (long long) number_of_bytes
        * (long long) pool_size;
    junk = (int*) malloc(number_of_bytes);

```

```

/* Now every process creates its own file name and attempts
   to open the file. */

sprintf(file_name, "%s.%d", basename, my_rank);

#ifdef DEBUG
    printf("%3d: opening file %s\n", my_rank, file_name);
#endif

my_file_open_error =
    MPI_File_open(MPI_COMM_SELF, file_name,
MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);

if (my_file_open_error != MPI_SUCCESS) {

    MPI_Error_class(my_file_open_error, &error_class);
    MPI_Error_string(error_class, error_string, &length_of_error_string);
    printf("%3d: %s\n", my_rank, error_string);

    MPI_Error_string(my_file_open_error, error_string,
        &length_of_error_string);
    printf("%3d: %s\n", my_rank, error_string);

    my_file_open_error = TRUE;

}

/* Now we must ALL check that NOBODY had problems
   with opening the file. */

MPI_Allreduce (&my_file_open_error, &file_open_error, 1, MPI_INT,
MPI_LOR, MPI_COMM_WORLD);

#ifdef DEBUG
    if (i_am_the_master)
        if (file_open_error)
            fprintf(stderr, "problem opening output files\n");
#endif

/* If all files are open for writing, write to them */

if (! file_open_error) {
    srand(28 + my_rank);
    for (i = 0; i < number_of_integers; i++) *(junk + i) = rand();
    start = MPI_Wtime();
    my_write_error =
MPI_File_write(fh, junk, number_of_integers, MPI_INT, &status);
    if (my_write_error != MPI_SUCCESS) {
MPI_Error_class(my_write_error, &error_class);
MPI_Error_string(error_class, error_string, &length_of_error_string);
printf("%3d: %s\n", my_rank, error_string);
MPI_Error_string(my_write_error, error_string, &length_of_error_string);
printf("%3d: %s\n", my_rank, error_string);
my_write_error = TRUE;
    }
    else {
finish = MPI_Wtime();
}
}

```

```

io_time = finish - start;
printf("%3d: io_time = %f\n", my_rank, io_time);
}

/* Check if anybody had problems writing on the file */

MPI_Allreduce (&my_write_error, &write_error, 1, MPI_INT,
MPI_LOR, MPI_COMM_WORLD);

#ifdef DEBUG
    if (i_am_the_master)
if (write_error)
    fprintf(stderr, "problem writing on files\n");
#endif

} /* of if(! file_open_error) { <write on the file> } */

/* Only processes that were successful opening the files
   need do close them here */

if (!my_file_open_error) {
    MPI_File_close(&fh);
#ifdef DEBUG
    printf ("%3d: closed %s\n", my_rank, file_name);
#endif
}

/* If we have either write errors or file open errors,
   then processes that managed to open their files
   are requested to throw them away */

if (write_error || file_open_error) {
    if (! my_file_open_error) {
        MPI_File_delete(file_name, MPI_INFO_NULL);
#ifdef DEBUG
        printf("%3d: deleted %s\n", my_rank, file_name);
#endif
    }
}
else {
    MPI_Reduce(&io_time, &longest_io_time, 1, MPI_DOUBLE, MPI_MAX,
MASTER_RANK, MPI_COMM_WORLD);
    if (i_am_the_master) {
printf("longest_io_time      = %f seconds\n", longest_io_time);
printf("total_number_of_bytes = %lld\n", total_number_of_bytes);
printf("transfer rate         = %f MB/s\n",
        total_number_of_bytes / longest_io_time / MBYTE);
    }
}

} /* end of if (write_error || file_open_error) {<delete open files>} */

} /* end of if(! input_error) { <open the file and write on it> } */

MPI_Finalize();
exit(0);
}

```

Here is how this program is made:

```

gustav@bh1 $ pwd
/N/B/gustav/src/I590/mkrandpfiles
gustav@bh1 $ make install
co RCS/Makefile,v Makefile
RCS/Makefile,v --> Makefile
revision 1.1
done
co RCS/mkrandpfiles.c,v mkrandpfiles.c
RCS/mkrandpfiles.c,v --> mkrandpfiles.c
revision 1.2
done
mpicc -DDEBUG -D_FILE_OFFSET_BITS=64 -D_LARGEFILE64_SOURCE -o mkrandpfiles mkrandpfiles.c
install mkrandpfiles /N/B/gustav/bin
gustav@bh1 $

```

And now let us have a look at how it runs. We'll check for error handling first.

```

gustav@bh1 $ pwd
/N/gpfs/gustav/mkrandpfiles
gustav@bh1 $ mpiexec -n 8 mkrandpfiles -l 8
synopsis: mkrandpfiles -f <file> -l <blocks>
gustav@bh1 $ mpiexec -n 8 mkrandpfiles -f test
synopsis: mkrandpfiles -f <file> -l <blocks>
gustav@bh1 $ touch test.4
gustav@bh1 $ chmod 444 test.4
gustav@bh1 $ mpiexec -n 8 mkrandpfiles -f test -l 8
basename          = test
basename_length   = 5
number_of_blocks  = 8
 0: basename = test, number_of_blocks = 8
 0: opening file test.0
 2: basename = test, number_of_blocks = 8
 2: opening file test.2
 4: basename = test, number_of_blocks = 8
 4: opening file test.4
 6: basename = test, number_of_blocks = 8
 6: opening file test.6
 7: basename = test, number_of_blocks = 8
 7: opening file test.7
 3: basename = test, number_of_blocks = 8
 3: opening file test.3
 5: basename = test, number_of_blocks = 8
 5: opening file test.5
 1: basename = test, number_of_blocks = 8
 1: opening file test.1
 4: Other I/O error
 4: Other I/O error Permission denied
problem opening output files
 0: closed test.0
 2: closed test.2
 6: closed test.6
 1: closed test.1
 5: closed test.5
 3: closed test.3
 7: closed test.7
 6: deleted test.6
 0: deleted test.0

```

```

2: deleted test.2
3: deleted test.3
7: deleted test.7
5: deleted test.5
1: deleted test.1
gustav@bh1 $

```

And now a real run:

```

gustav@bh1 $ rm -f test.4
gustav@bh1 $ mpiexec -n 8 mkrandpfiles -f test -l 10
basename          = test
basename_length   = 5
number_of_blocks  = 10
0: basename = test, number_of_blocks = 10
0: opening file test.0
1: basename = test, number_of_blocks = 10
1: opening file test.1
2: basename = test, number_of_blocks = 10
2: opening file test.2
6: basename = test, number_of_blocks = 10
6: opening file test.6
4: basename = test, number_of_blocks = 10
4: opening file test.4
7: basename = test, number_of_blocks = 10
7: opening file test.7
3: basename = test, number_of_blocks = 10
3: opening file test.3
5: basename = test, number_of_blocks = 10
5: opening file test.5
7: io_time = 3.383798
3: io_time = 3.461928
5: io_time = 3.508230
2: io_time = 3.523316
4: io_time = 3.783771
1: io_time = 3.955147
6: io_time = 4.276280
0: io_time = 5.050947
1: closed test.1
2: closed test.2
3: closed test.3
4: closed test.4
7: closed test.7
5: closed test.5
6: closed test.6
0: closed test.0
longest_io_time    = 5.050947 seconds
total_number_of_bytes = 335544320
transfer rate      = 63.354455 MB/s
gustav@bh1 $

```

## The Discussion

The beginning of the program is the same as in `mkrandfiles` until we get to the place where processes have to allocate space for data. This is done the same way we did it in `mkrandpfile`, i.e., we allocate enough space to generate *all* the data and then to write it in one shove onto the file:

```

/* Allocate space needed to generate the integers */

number_of_integers = number_of_blocks * BLOCK_SIZE;
number_of_bytes = sizeof(int) * number_of_integers;
total_number_of_integers = (long long) number_of_integers
    * (long long) pool_size;
total_number_of_bytes = (long long) number_of_bytes
    * (long long) pool_size;
junk = (int*) malloc(number_of_bytes);

```

Now every process generates its own file name:

```
sprintf(file_name, "%s.%d", basename, my_rank);
```

```

#ifdef DEBUG
    printf("%3d: opening file %s\n", my_rank, file_name);
#endif

```

and opens the file:

```

my_file_open_error =
    MPI_File_open(MPI_COMM_SELF, file_name,
        MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);

```

Observe the use of the predefined `MPI_COMM_SELF` communicator. This is a communicator that contains one process only, namely the process that uses it. Consequently the file in question will not be shared with other processes. It belongs to this process in entirety.

Now we check for possible errors on open:

```

if (my_file_open_error != MPI_SUCCESS) {

    MPI_Error_class(my_file_open_error, &error_class);
    MPI_Error_string(error_class, error_string, &length_of_error_string);
    printf("%3d: %s\n", my_rank, error_string);

    MPI_Error_string(my_file_open_error, error_string,
        &length_of_error_string);
    printf("%3d: %s\n", my_rank, error_string);

    my_file_open_error = TRUE;

}

```

We don't call `MPI_Abort`. Instead we set `my_file_open_error` to `TRUE`. If `MPI_SUCCESS` was zero, then we could use the value of `my_file_open_error` simply as returned by `MPI_File_open`, but `MPI_SUCCESS` is opaque and we must not make such assumptions.

Now all processes check if any of them experienced problems opening the file:

```

MPI_Allreduce (&my_file_open_error, &file_open_error, 1, MPI_INT,
    MPI_LOR, MPI_COMM_WORLD);

```

```

#ifdef DEBUG
    if (i_am_the_master)
        if (file_open_error)
            fprintf(stderr, "problem opening output files\n");
#endif

```

The remaining part of the program is one large `if` statement followed by cleanup that depends on what happened before:

```

if (! file_open_error) {
    blah... blah... blah...
}

if (!my_file_open_error) MPI_File_close(&fh);

if (write_error || file_open_error) {
    if (! my_file_open_error) MPI_File_delete(file_name, MPI_INFO_NULL);
}
else {
    MPI_Reduce(&io_time, &longest_io_time, 1, MPI_DOUBLE, MPI_MAX,
              MASTER_RANK, MPI_COMM_WORLD);
    if (i_am_the_master) {
        printf("longest_io_time      = %f seconds\n", longest_io_time);
        printf("total_number_of_bytes = %lld\n", total_number_of_bytes);
        printf("transfer rate         = %f MB/s\n",
              total_number_of_bytes / longest_io_time / MBYTE);
    }
}

```

Here you see that in case of a problem all successfully opened files are deleted by calling `MPI_File_delete`.

Now let us have a look at what the `blah... blah... blah...` stands for.

First every process seeds the random number generator differently and generates the integers:

```

srand(28 + my_rank);
for (i = 0; i < number_of_integers; i++) *(junk + i) = rand();

```

Then we execute a timed write on the open file using `MPI_File_write` and capture the error, if there is any to be caught. The error is treated similarly to the way we did it for `MPI_File_open`. We don't `MPI_Abort` if there is a problem. Instead we set `my_write_error` to `TRUE` and then flow with it.

```

start = MPI_Wtime();
my_write_error =
    MPI_File_write(fh, junk, number_of_integers, MPI_INT, &status);
if (my_write_error != MPI_SUCCESS) {
    MPI_Error_class(my_write_error, &error_class);
    MPI_Error_string(error_class, error_string, &length_of_error_string);
    printf("%3d: %s\n", my_rank, error_string);
    MPI_Error_string(my_write_error, error_string, &length_of_error_string);
    printf("%3d: %s\n", my_rank, error_string);
    my_write_error = TRUE;
}
else {
    finish = MPI_Wtime();
    io_time = finish - start;
    printf("%3d: io_time = %f\n", my_rank, io_time);
}

```

Before we exit the `if` clause we still check if any of the processes has encountered writing problems and set the variable `write_error` accordingly.

```

MPI_Allreduce (&my_write_error, &write_error, 1, MPI_INT,

```

```

        MPI_LOR, MPI_COMM_WORLD);

#ifdef DEBUG
    if (i_am_the_master)
        if (write_error)
            fprintf(stderr, "problem writing on files\n");
#endif

```

At the end of all these adventures the processes meet at `MPI_Finalize` and `exit`.

### Exercises

- 1 Write a program for reading files written with `mkrandpfiles`. Test the `status` variable with `MPI_Get_count` to verify the number of data items read.
- 2 We could use something similar to `mkrandpfiles` in order to dump data pertaining to particles (see the program in section 5.2.6) at every time step. Explain why this would make our life difficult if we wanted to restart the program on a number of nodes different from the number used to write the files.

### 5.3.4 File Views

In the three previous sections we wrote and read MPI files in parallel by *manipulating file pointers explicitly* and by moving the pointers in terms of bytes, e.g., 800 bytes forward, or 800 bytes back. This worked, but this is a pretty low level way of doing things and therefore prone to mistakes. There was nothing in the way we accessed the files that would prevent, say, process of rank 3 overwriting data written by process of rank 4. Furthermore data itself was written and read in the simplest way possible, e.g., as a string of integers or a string of bytes.

MPI provides a mechanism for automating this whole process and structuring it in a high level way, so that processes never step on each other's toes – and yet all writes and reads occur in parallel. The same mechanism is used to access data not in terms of bytes or low level types such as integers or doubles, but in terms of whole data structures. In this process file pointers are advanced in terms of the data structures too.

This mechanism is called *file views*. The view that a given process has of an open file is defined in terms of two data types, the *elementary data type* and the *file type*. The elementary data type may be quite complex, it may be a whole large structure that contains some doubles, some integers, some strings and even other structures.

The way the file is then partitioned amongst the processes is defined in terms of the *file type*, which becomes the *template* for accessing the file by the processes. The file type should be constructed from multiple instances of the same *elementary type*. Every process is then expected to define the *file*



*type* differently, which, if such definitions are consistent, guarantees that the processes will not step on each other's toes.

In order to make this concept more palatable to the reader, we have to go back to the very concept of an MPI data type and cover it in some depth. The following section is going to do just this, and then we'll go back to MPI-IO and see how the MPI data construction and manipulation utilities play there.

### Derived Datatypes in MPI

MPI defines the following basic data types, from which all other data types can be derived:

**MPI\_CHAR** This is the traditional ASCII character that is numbered by integers between 0 and 127.

**MPI\_UNSIGNED\_CHAR** This is the extended character numbered by integers between 0 and 255.

**MPI\_BYTE** This is an 8-bit positive integer between 0 and 255, i.e., a byte.

**MPI\_WCHAR\_T** This is a wide character, e.g., a 16-bit character such as a Chinese ideogram.

**MPI\_SHORT** This is a 16-bit integer between -32,768 and 32,767.

**MPI\_UNSIGNED\_SHORT** This is a 16-bit positive integer between 0 and 65,535.

**MPI\_INT** This is a 32-bit integer between -2,147,483,648 and 2,147,483,647.

**MPI\_UNSIGNED** This is a 32-bit unsigned integer, i.e., a number between 0 and 4,294,967,295.

**MPI\_LONG** This is the same as `MPI_INT` on IA32.

**MPI\_UNSIGNED\_LONG** This is the same as `MPI_UNSIGNED` on IA32.

**MPI\_FLOAT** This is a single precision, 32-bit long floating point number.

**MPI\_DOUBLE** This is a double precision, 64-bit long floating point number.

**MPI\_LONG\_DOUBLE** This is a quadruple precision, 128-bit long floating point number.

**MPI\_LONG\_LONG\_INT** This is a 64-bit long signed integer, i.e., an integer number between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807 (this reads: 9 quintillions 223 quadrillions 372 trillions 36 billions 854 millions 775 thousand 8 hundred and seven – not a large sum of money by Microsoft standards).

**MPI\_LONG\_LONG** Same as `MPI_LONG_LONG_INT`.

**MPI\_FLOAT\_INT** This is a pair of a 32-bit floating point number followed by a 32-bit integer.

**MPI\_DOUBLE\_INT** This is a pair of a 64-bit floating point number followed by a 32-bit integer.

**MPI\_LONG\_INT** This is a pair of a long integer (which under IA32 is just a 32-bit integer) followed by a 32-bit integer.

**MPI\_SHORT\_INT** This is a pair of a 16-bit short integer followed by a 32-bit integer.

**MPI\_2INT** This is a pair of two 32-bit integers.

**MPI\_LONG\_DOUBLE\_INT** This is a pair of a quadruple precision 128-bit floating point number and a 32-bit integer.

**MPI\_LB** The lower bound marker.

**MPI\_UB** The upper bound marker.

These basic types can then be used to construct new, more elaborate data structures, by calling MPI type constructors. The type constructors characterize all these new data structures simply in terms of a *type map*, which is a list of pairs, each of which comprises a type and a displacement:

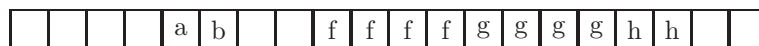
$$\text{type map} = ((\text{type}_0, \text{displacement}_0)(\text{type}_1, \text{displacement}_1) \dots (\text{type}_n, \text{displacement}_n))$$

where the displacement is the number of bytes from the beginning of the structure at which the given data item appears in it.

This is a very low level characterization of the data structure and it is likely to lack portability between, e.g., 32-bit and 64-bit systems – due to, e.g., different meanings of “long” and “long long” on these systems.

The secret to writing portable programs is never to assume any lengths and instead to call `sizeof` whenever you need to use the lengths of various types in your code. You can then help yourself by calling MPI address and extent functions to check on the exact locations of various items in the structures. The locations may not always be what you think, because in some cases the data may have to be aligned at word or half-word boundaries, in which case the compiler may pad the data, i.e., insert empty space between, e.g., a character and the half-word boundary. Following the last typed data item in the structure, located at some displacement, we may still find some padding, if the data structure has to be aligned with the word (half-word) boundary.

Consider the following example:



Here we have a data object that begins with four bytes of padding, i.e., there is nothing there in the first four bytes. Then we have two characters,  $a$  and  $b$ , followed again by two bytes of padding. Then we have 2 32-bit floats  $f$  and  $g$ , which are adjacent, and this is followed immediately by a 16-bit short integer  $h$  and then again 2 bytes of padding.

Let us call this object

```
MPI_Datatype new_type
```

This new MPI type comprises five blocks of the following items.

- The first block contains just the beginning of the structure, which we can mark by inserting here the *lower-bound marker*, i.e., the data item of type `MPI_LB` and zero length, and its displacement from the beginning of the structure is zero bytes.
- The second block contains two characters and its displacement from the beginning of the structure is 4 bytes.
- The third block contains two 32-bit floats and its displacement from the beginning of the structure is 8 bytes.
- The fourth block contains one 16-bit short and its displacement from the beginning of the structure is 16 bytes.
- And the fifth block, yes, there is the fifth block here, contains just the end of the structure. We can mark the end by inserting there the *upper-bound marker*, i.e., the data item of type `MPI_UB` and zero length, and its displacement from the beginning of the structure is 20 bytes.

Now we convert this verbal description into the following call to function `MPL_Type_create_struct` as follows:

```
MPI_Type_create_struct(5, array_of_block_lengths, array_of_displacements,
                      array_of_types, &new_type);
```

where

```
array_of_block_lengths = (1, 2, 2, 1, 1)
array_of_displacements = (0, 4, 8, 16, 20)
array_of_types         = (MPI_LB, MPI_CHAR, MPI_FLOAT, MPI_SHORT, MPI_UB)
```

Observe that if this *MPI structure* corresponds to some C-language structure in your program, it is your responsibility, as the programmer, to ensure that the two are indeed the same.

**Note** Function `MPI_Type_create_struct` is an MPI-2 function. This same function is called `MPL_Type_struct` in MPI-1.

In order to find out how a C-language structure is laid out in the memory of your computer you can use function `MPL_Get_address` in combination with `sizeof`.

Consider, for example, the following C-language (not an MPI) structure:

```
typedef struct {
    char  flavor;
    char  color;
    int   charge;
    double mass;
    double x, y, z;
    double px, py, pz;
} Quark;
```

```
Quark tau;
```

To find how this structure should be described to MPI, you could do something like:

```
int quark_address, flavor_address, color_address, charge_address, mass_address,
    x_address, y_address, z_address, px_address, py_address, pz_address;
int flavor_offset, color_offset, charge_offset, mass_offset, ...
...
MPI_Get_address(&tau, &tau_address);
MPI_Get_address(&tau.flavor, &flavor_address);
MPI_Get_address(&tau.color, &color_address);
MPI_Get_address(&tau.charge, &charge_address);
MPI_Get_address(&tau.mass, &mass_address);
...
flavor_offset = flavor_address - tau_address;
color_offset = color_address - tau_address;
charge_offset = charge_address - tau_address;
mass_offset = mass_address - tau_address;
...

```

Then having these numbers in hand and using `sizeof` to get sizes of `char`, `double`, etc., you would describe this structure to MPI by calling `MPI_Type_create_struct` with appropriate parameters.

**Note** Function `MPI_Get_address` is an MPI-2 function. There is an identical function in MPI-1, which is called `MPI_Address`.

`MPI_Type_create_struct` is one of the more general type constructors in MPI. There are more specific constructors, of which we have already encountered `MPI_Type_contiguous` and `MPI_Type_vector`.

A type constructor, which is somewhere in between `MPI_Type_create_struct` and `MPI_Type_vector` is `MPI_Type_indexed`. It is like `MPI_Type_vector`, meaning that it works with data items of the same type, but as in `MPI_Type_struct` the data can be placed at quite arbitrary locations in memory, not necessarily with a regular stride and regular block length.

### Defining File Views in Terms of MPI Datatypes

How are we going to use data types in order to tell processes how to partition a file?

This is how. Suppose the picture below represents a file. Each little square corresponds to a data item of some *elementary type*, and this type may be quite complex. It is *elementary* not because it is simple, but because this is what the file is made of.

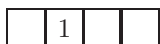


Suppose we have 4 processes in the pool. Let us define the *filetype* for the process of rank 0 to be:



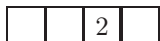
where a 0 in the square means that this particular data item is considered full, and the remaining three squares without a 0 in them are empty. As far as process zero is concerned, they are just padding.

Now, let the *filetype* for process of rank 1 be:



where a 1 in the square means that this particular data item is considered full, and the remaining three squares without a 1 in them are empty. As far as process 1 is concerned, they are just padding.


Similarly the *filetype* for process of rank 2 is:



and for process of rank 3:



Now process of rank 0 is going to call function `MPI_File_set_view` to establish its view of the file as follows:

```
MPI_File_set_view(fh, 0,  ,  , "native", MPI_INFO_NULL)
```

where

- the first argument of `MPI_File_set_view` is the file handle;
- the second argument of `MPI_File_set_view` is the displacement from the beginning of the file in bytes of the place where this file view begins – a file may have different views associated with it in various places;
- the third argument of `MPI_File_set_view` is the *elementary data type*;
- the fourth argument of `MPI_File_set_view` is the *file data type*, which must be defined in terms of elementary data types;
- the fifth argument of `MPI_File_set_view` is a string that defines the *data representation* – in this case it is “native”;
- the last argument of `MPI_File_set_view` is the info structure – in this case it is `MPI_INFO_NULL`.

Let me skip the detailed discussion of this function for the time being, and just show how the remaining processes are going to call it:

```

MPI_File_set_view(fh, 0,  , 1  , "native", MPI_INFO_NULL)
MPI_File_set_view(fh, 0,  ,   2 , "native", MPI_INFO_NULL)
MPI_File_set_view(fh, 0,  ,    3, "native", MPI_INFO_NULL)

```

As you see for every process the only thing that is unique to the process is the file type.

Once all processes have issued the calls this is how the file is going to be partitioned:

0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

When process 1, say, issues `MPI_File_read`, it is going to read its own items only, i.e., the ones labeled with 1. Its own file pointer will be automatically advanced to the required location in the file.

So this is how the file gets partitioned without us having to specify separate file offsets for each process explicitly. But constructing such different file views for each process may not be all this easy either. Luckily MPI-2 provides us with a very powerful function `MPI_Type_create_darray` that can generate process dependent file views automatically. But before I get to explain how this function works, let me go back to `MPI_File_set_view` and explain in more detail the meaning of its various arguments, as well as the behaviour of the function itself.

`MPI_File_set_view` is a collective function. All processes that have opened the file have to participate in this call. The file handle and the data representation strings must be identical for all processes. The *extent* of the elementary type, i.e., the distance between its upper and its lower marker in bytes, must be the same for all processes. But the processes may call this function with different displacements, file types and infos. Note that apart from differentiating the view with a process specific file type, you *may* use different initial displacements too.

The data representation string specifies how the data that is passed to `MPI_File_write` is going to be stored on the file itself. The simplest way to write a file, especially under UNIX, is to copy the bytes from memory to the disk without any further processing. But under other operating systems files may have fancy structures, multiple forks, format records and what not. Even under UNIX Fortran files differ from plain C-language files, because Fortran files may have record markers embedded in them.

MPI defines three data representations and MPI implementations are free to add more. The three basic representations are:

**"native"** Data in this representation is stored in a file exactly as it is in memory. This format is fine for homogeneous farms, but it may fail for heterogeneous farms, because of problems with big-endian versus small-endian conversions and other data representation incompatibilities.

**”internal”** Data in this representation is written in an MPI-implementation dependent and operating system and architecture independent format. The MPI implementation will perform required conversions transparently when data is transferred between computational nodes, possibly of various architectures, and the file.

**”external32”** Data in this representation is written in the “big-endian IEEE” format, which is also operating system and architecture independent. Writing data in this format will let you take the file away from the MPI environment, it’s been written in, for processing on other machines under other operating systems and on other architectures.

When the file gets opened with `MPI_File_open`, you get the default view, which is equivalent to the call:

```
MPI_File_set_view(fh, 0, MPI_BYTE, MPI_BYTE, "native", MPI_INFO_NULL);
```

Now let us get to `MPI_Type_create_darray`, the function that is going to make our task of defining process dependent file views easier.

This function does a lot of very hard work and, at the same time, it is going to save the programmer a lot of very hard work too, but for this very reason it is a little complicated. Its synopsis is as follows:

```
int MPI_Type_create_darray(
    int size,
    int rank,
    int ndims,
    int array_of_gsizes[],
    int array_of_distrib[],
    int array_of_dargs[],
    int array_of_psizes[],
    int order,
    MPI_Datatype oldtype,
    MPI_Datatype *newtype)
```

When called it is going to generate the datatypes corresponding to the distribution of an `ndims`-dimensional array of `oldtype` elements onto an `ndims`-dimensional grid of logical processes.

Remember how we had a 2-dimensional grid of processes in section 5.2.5 that talked about solving a diffusion problem. There we also had a 2-dimensional array of integers, which we have distributed manually amongst the processes of the 2-dimensional grid, so that each process got a small portion of it and then worked on it updating its edges by getting values from its neighbours. Function `MPI_Type_create_darray` is going to deliver us of such partitioning automatically.

The parameters of the function have the following meaning

**size** this is the size of the process group;

**rank** this is the rank of the process in the group;

**ndims** this is the number of the process grid dimensions – here we assume that we have created a Cartesian grid of processes – this is also the number

of the dimensions of the array that is going to be distributed amongst the processes, e.g., if we have a 3-dimensional array, we need to have a 3-dimensional grid of processes to distribute the array over;

**array\_of\_gsizes** this is a one-dimensional array of positive integers of length `ndims`; each entry in the array tells us about the number of elements of type `oldtype` in the corresponding dimension of the global array;

**array\_of\_distribs** this is a one-dimensional array of predefined MPI constants that specifies how a corresponding dimension of the matrix should be distributed; the three constants provided are `MPI_DISTRIBUTE_BLOCK` – which requests block distribution along the corresponding dimension, `MPI_DISTRIBUTE_CYCLIC` – which requests cyclic distribution along the corresponding dimension, and `MPI_DISTRIBUTE_NONE` – which requests no distribution along the corresponding dimension;

**array\_of\_dargs** this is a one-dimensional array of positive integers of length `ndims`; each entry in the array is the argument that further specifies how the distribution of the array should be done – there is one MPI constant provided here, `MPI_DISTRIBUTE_DFLT_DARG`, which lets MPI do default distribution characterized only by `array_of_distribs`;

**array\_of\_psizes** this is a one-dimensional array of positive integers of length `ndims`; each entry in the array tells us about the number of processes in the corresponding dimension of the process grid;

**order** this is the storage order flag: arrays may be stored either FORTRAN-style, i.e., column-major, or C-style, i.e., row-major. There are two predefined MPI constants that may be used here `MPI_ORDER_FORTRAN` and `MPI_ORDER_C`.

**oldtype** this is the MPI type of a single item in the array – because it is an array, every item in it is, of course, of the same type – but the types don't have to be basic, they may be quite complex structures;

**newtype** this is the new MPI data type that is going to be specific to each process and that can be used in the call to `MPI_Set_file_view`.

At this stage I feel that you need a programming example to make sense of all this. So here it is.

### Program `darray.c`

This program exercises a number of new MPI function calls. They are `MPI_Dims_create`, `MPI_Type_create_darray`, `MPI_Type_extent`, `MPI_Type_size`, `MPI_File_set_view`, `MPI_File_write_all`, and `MPI_File_read_all`.

```
/*
 * %Id: darray.c,v 1.2 2003/10/26 22:48:13 gustav Exp %
 *
```



```

* %Log: darray.c,v %
* Revision 1.2  2003/10/26 22:48:13  gustav
* Expunged any references to "info".
*
* Revision 1.1  2003/10/26 22:05:50  gustav
* Initial revision
*
*/

#include <stdio.h>      /* printf and relatives live here */
#include <stdlib.h>     /* exit lives here */
#include <unistd.h>     /* getopt lives here */
#include <string.h>     /* strlen lives here */
#include <sys/types.h> /* chmod needs these two */
#include <sys/stat.h>
#include <mpi.h>       /* all MPI stuff lives here (including MPI-IO) */

#define MASTER_RANK 0
#define TRUE 1
#define FALSE 0
#define BOOLEAN int
#define MBYTE 1048576
#define NDIMS 3
#define SIZE 512
#define SYNOPSIS printf ("synopsis: %s -f <file>\n", argv[0])

int main(argc, argv)
    int argc;
    char *argv[];
{
    /* my variables */

    int my_rank, pool_size, i, ndims, order, file_name_length,
        array_of_gsizes[NDIMS], array_of_distrib[NDIMS],
        array_of_dargs[NDIMS], array_of_psize[NDIMS],
        *write_buffer, write_buffer_size, count,
        *read_buffer, read_buffer_size;
    BOOLEAN i_am_the_master = FALSE, input_error = FALSE,
        file_open_error = FALSE, file_write_error = FALSE, verbose = FALSE,
        my_read_error = FALSE, read_error = FALSE;
    char *file_name = NULL, message[BUFSIZ];

    /* MPI variables */

    MPI_Offset file_size;
    MPI_File fh;
    MPI_Status status;
    MPI_Datatype file_type;
    MPI_Aint file_type_size, file_type_extent;
    int error_string_length;
    char error_string[BUFSIZ];

    /* getopt variables */

    extern char *optarg;
    int c;

```

```

/* error handling variables */

extern int errno;

/* ACTION */

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &pool_size);
if (my_rank == MASTER_RANK) i_am_the_master = TRUE;

if (i_am_the_master) {
  while ((c = getopt(argc, argv, "f:vh")) != EOF)
    switch(c) {
      case 'f':
        file_name = optarg;
        break;
      case 'v':
        verbose = TRUE;
        break;
      case 'h':
        input_error = TRUE;
        break;
      case '?':
        input_error = TRUE;
        break;
    }
  if (file_name == NULL) input_error = TRUE;
  if (input_error)
    SYNOPSIS;
  else {
    file_name_length = strlen(file_name) + 1;
    if (verbose) {
printf("file_name          = %s\n", file_name);
printf("file_name_length    = %d\n", file_name_length);
    }
  }
}

MPI_Bcast(&input_error, 1, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);

if (! input_error) {
  MPI_Bcast(&verbose, 1, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);
  MPI_Bcast(&file_name_length, 1, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);
  if (! i_am_the_master) file_name = (char*) malloc(file_name_length);
  MPI_Bcast(file_name, file_name_length, MPI_CHAR, MASTER_RANK,
    MPI_COMM_WORLD);

  /* Prepare for calling MPI_Type_create_darray */

  ndims = NDIMS;
  for (i = 0; i < ndims; i++) {
    array_of_gsizes[i] = SIZE;
    array_of_distribs[i] = MPI_DISTRIBUTE_BLOCK;
    array_of_dargs[i] = MPI_DISTRIBUTE_DFLT_DARG;
    array_of_psizes[i] = 0;
  }
}

```

```

}
MPI_Dims_create(pool_size, ndims, array_of_psizes);
order = MPI_ORDER_C;

/* Now call MPI_Type_create_darray */

if (verbose) {
    printf ("%3d: calling MPI_Type_create_darray with\n", my_rank);
    printf ("%3d:   pool_size           = %d\n", my_rank, pool_size);
    printf ("%3d:   my_rank            = %d\n", my_rank, my_rank);
    printf ("%3d:   ndims              = %d\n", my_rank, ndims);
    printf ("%3d:   array_of_gsizes    = (%d, %d, %d)\n", my_rank,
        array_of_gsizes[0],
        array_of_gsizes[1],
        array_of_gsizes[2]);
    printf ("%3d:   array_of_distribs = (%d, %d, %d)\n", my_rank,
        array_of_distribs[0],
        array_of_distribs[1],
        array_of_distribs[2]);
    printf ("%3d:   array_of_dargs     = (%d, %d, %d)\n", my_rank,
        array_of_dargs[0],
        array_of_dargs[1],
        array_of_dargs[2]);
    printf ("%3d:   array_of_psizes   = (%d, %d, %d)\n", my_rank,
        array_of_psizes[0],
        array_of_psizes[1],
        array_of_psizes[2]);
    printf ("%3d:   order              = %d\n", my_rank, order);
    printf ("%3d:   type                = %d\n", my_rank, MPI_INT);
}
MPI_Type_create_darray(pool_size, my_rank, ndims,
array_of_gsizes, array_of_distribs,
array_of_dargs, array_of_psizes, order,
MPI_INT, &file_type);
MPI_Type_commit(&file_type);

/* Explore the returned type */

MPI_Type_extent(file_type, &file_type_extent);
MPI_Type_size(file_type, &file_type_size);
if (verbose) {
    printf ("%3d: file_type_size   = %d\n", my_rank, file_type_size);
    printf ("%3d: file_type_extent = %d\n", my_rank, file_type_extent);
}

/* Allocate space for your own write buffer based on the
   return of the MPI_Type_size call. */

write_buffer_size = file_type_size / sizeof(int);
write_buffer = (int*) malloc(write_buffer_size * sizeof(int));

/* We do this in case sizeof(int) does not divide file_type_size
   exactly. But this should not happen if we have called
   MPI_Type_create_darray with MPI_INT as the original data
   type. */

if (! write_buffer) {

```

```

    sprintf(message, "%3d: malloc write_buffer", my_rank);
    perror(message);
    MPI_Abort(MPI_COMM_WORLD, errno);

    /* We can still abort, because we have not opened any
    files yet. Notice that since MPI_Type_create_darray
    will fail if SIZE^3 * sizeof(int) exceeds MAX_INT,
    because MPI_Aint on AVIDD is a 32-bit integer,
    we are rather unlikely to fail on this malloc
    anyway. */
    }

    MPI_Barrier(MPI_COMM_WORLD);
    /* We wait here in case some procs have problems with malloc. */

    /* Initialize the buffer */

    for (i = 0; i < write_buffer_size; i++)
        *(write_buffer + i) = my_rank * SIZE + i;

    file_open_error = MPI_File_open(MPI_COMM_WORLD, file_name,
    MPI_MODE_CREATE | MPI_MODE_WRONLY,
    MPI_INFO_NULL, &fh);
    if (file_open_error != MPI_SUCCESS) {
        MPI_Error_string(file_open_error, error_string,
        &error_string_length);
        fprintf(stderr, "%3d: %s\n", my_rank, error_string);
        MPI_Abort(MPI_COMM_WORLD, file_open_error);

        /* It is still OK to abort, because we have failed to
        open the file. */

    }
    else {

        if (i_am_the_master)
            chmod(file_name, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
        MPI_Barrier(MPI_COMM_WORLD);

        /* We should be able to change permissions on the file by setting
        the "file_perm" hint in the info structure before passing
        it to MPI_File_open, but MPICH2 doesn't support this
        yet. All processes have to meet on the barrier before
        further action. */

        MPI_File_set_view(fh, 0, MPI_INT, file_type, "native", MPI_INFO_NULL);
        file_write_error =
        MPI_File_write_all(fh, write_buffer, write_buffer_size, MPI_INT,
        &status);
        if (file_write_error != MPI_SUCCESS) {
            MPI_Error_string(file_write_error, error_string,
            &error_string_length);
            fprintf(stderr, "%3d: %s\n", my_rank, error_string);
            MPI_File_close(&fh);
            free(write_buffer);
            if (i_am_the_master) MPI_File_delete(file_name, MPI_INFO_NULL);
        }
    }
}

```

```

    else {
MPI_Get_count(&status, MPI_INT, &count);
MPI_File_get_size(fh, &file_size);
if(verbose) {
    printf("%3d: wrote %d integers\n", my_rank, count);
    printf("%3d: file size is %lld bytes\n", my_rank, file_size);
}
MPI_File_close(&fh);

    /* We have managed to open, write on it and close the file.
       Now we're going to read it the same way we wrote it. */

    read_buffer_size = write_buffer_size;
    read_buffer = (int*) malloc(read_buffer_size * sizeof(int));
    if (! read_buffer) {
        sprintf(message, "%3d: malloc read_buffer", my_rank);
        perror(message);
        MPI_Abort(MPI_COMM_WORLD, errno);

        /* We can abort, because the file has been closed and
           we haven't opened it for reading yet. */

    }

    MPI_Barrier(MPI_COMM_WORLD);
    /* We wait here in case some procs have problems with malloc. */

    MPI_File_open(MPI_COMM_WORLD, file_name, MPI_MODE_RDONLY,
        MPI_INFO_NULL, &fh);

    /* We don't check for errors here, because we've just closed
       this file a moment ago, so it should still be there. */

    MPI_File_set_view(fh, 0, MPI_INT, file_type, "native", MPI_INFO_NULL);
    MPI_File_read_all(fh, read_buffer, read_buffer_size, MPI_INT, &status);
    MPI_Get_count(&status, MPI_INT, &count);
    if (verbose)
        printf("%3d: read %d integers\n", my_rank, count);
    MPI_File_close(&fh);

    /* Now check that the integers read are the same as the ones
       we wrote. */

    for (i = 0; i < read_buffer_size; i++) {
        if (*(write_buffer + i) != *(read_buffer + i)) {
            printf("%3d: data read different from data written, i = %d\n",
                my_rank, i);
            my_read_error = TRUE;
        }
    }

    MPI_Reduce (&my_read_error, &read_error, 1, MPI_INT, MPI_LOR,
        MASTER_RANK, MPI_COMM_WORLD);

    if (i_am_the_master)
        if (! read_error)
            printf("--> All data read back is correct.\n");

```

```

    } /* no problem with file write */
  } /* no problem with file open */
} /* no input error */

MPI_Finalize();
exit(0);
}

```

Here is how this program is compiled and installed:

```

gustav@bh1 $ pwd
/N/B/gustav/src/I590/darray
gustav@bh1 $ make install
co RCS/Makefile,v Makefile
RCS/Makefile,v --> Makefile
revision 1.2
done
co RCS/darray.c,v darray.c
RCS/darray.c,v --> darray.c
revision 1.1
done
mpicc -o darray darray.c
install darray /N/B/gustav/bin
gustav@bh1 $

```

And now here is how the program runs. It can be made to run silently, or you can ask for *verbose* output by invoking it with the `-v` option. I'll do this to show you what's going on inside. I run this program under PBS. Not because it is very heavy computationally or in terms of IO, but because I want to run it on 27 processors. The reason for this is that  $27 = 3 \times 3 \times 3$ , so I can have a non-trivial 3-dimensional process grid, over which I am going to distribute a 3-dimensional array ( $512 \times 512 \times 512$ ) of integers. Observe that  $512^3 = 134,217,728$  does not divide by 27:  $512^3/27 = 4971026\frac{26}{27}$ . Function `MPI_Type_create_darray` is going to allocate 5000211 integers to 8 processes each, 4970970 integers to 12 processes each, 4941900 integers to 6 processes each and 4913000 to one process. You can easily check that  $8 \times 5000211 + 12 \times 4970970 + 6 \times 4941900 + 1 \times 4913000 = 134217728 = 512^3$ .

This is my PBS job file:

```

gustav@bh1 $ pwd
/N/B/gustav/PBS
gustav@bh1 $ cat darray.sh
#PBS -S /bin/bash
#PBS -N darray
#PBS -o darray_out
#PBS -e darray_err
#PBS -q bg
#PBS -m a
#PBS -V
#PBS -l nodes=27
NODES=27
HOST='hostname'
echo Local MPD console on $HOST

```

```

cd /N/gpfs/gustav/darray
rm -f test
pwd
# Specify Myrinet interfaces on the hostfile.
grep -v $HOST $PBS_NODEFILE | sed 's/$/-myri0/' > $HOME/mpd.hosts
# Boot the MPI2 engine.
which mpdboot
mpdboot --totalnum=$NODES --file=$HOME/mpd.hosts
sleep 10
# Inspect if all MPI nodes have been activated.
which mpdtrace
mpdtrace
# Execute your MPI program.
which mpiexec
mpiexec -n $NODES darray -f test -v
# Shut down the MPI2 engine and exit the PBS script.
which mpdallexit
mpdallexit
exit 0
gustav@bh1 $

```

After I have submitted it with

```

gustav@bh1 $ qsub darray.sh
46101.bh1.avidd.iu.edu
gustav@bh1 $ qstat | grep gustav
46101.bh1          darray          gustav          O R bg
gustav@bh1 $

```

I got the following output written on darray\_out:

```

gustav@bh1 $ cat darray_out
[ ... various uninteresting junk ...]
0: calling MPI_Type_create_darray with
0:   pool_size      = 27
0:   my_rank       = 0
0:   ndims         = 3
0:   array_of_gsizes = (512, 512, 512)
0:   array_of_distribs = (121, 121, 121)
0:   array_of_dargs = (-49767, -49767, -49767)
0:   array_of_psizes = (3, 3, 3)
0:   order         = 56
0:   type          = 1275069445
0: file_type_size  = 20000844
0: file_type_extent = 536870912
[...]
16: calling MPI_Type_create_darray with
16:   pool_size      = 27
16:   my_rank       = 16
16:   ndims         = 3
16:   array_of_gsizes = (512, 512, 512)
16:   array_of_distribs = (121, 121, 121)
16:   array_of_dargs = (-49767, -49767, -49767)
16:   array_of_psizes = (3, 3, 3)
16:   order         = 56
16:   type          = 1275069445
16: file_type_size  = 19883880
16: file_type_extent = 536870912
[...]

```

```

    0: wrote 5000211 integers
    0: file size is 536870912 bytes
    16: wrote 4970970 integers
    16: file size is 536870912 bytes
    [...]
    0: read 5000211 integers
    1: read 5000211 integers
    2: read 4970970 integers
    [...]
    16: read 4970970 integers
    [...]
    24: read 4941900 integers
    25: read 4941900 integers
    26: read 4913000 integers
--> All data read back is correct.
gustav@bh1 $ ls -l /N/gpfs/gustav/darray
total 524288
-rw-r--r--  1 gustav  ucs      536870912 Oct 26 17:49 test
gustav@bh1 $

```

### The Discussion

The program begins with the usual incantations to MPI, which result in every process learning about its rank number and the size of the pool of processes. The master process, which doesn't really do any mastering in this program, but occasionally speaks for all and reads the command line, learns about being a master.

The command line analysis is done with `getopt` and there is one more option here, `-v`, which activates verbose output, by setting a boolean variable `verbose` to `TRUE`.

Next the master program broadcasts the content of `input_error` to all processes, so that they can all go directly to `MPI_Finalize` if there is such, otherwise the action begins. The master process broadcasts `verbose`, `file_name_length` and `file_name` to other processes.

Having received this information all processes promptly get to prepare themselves for calling `MPI_Type_create_darray`

```

ndims = NDIMS;
for (i = 0; i < ndims; i++) {
    array_of_gsizes[i] = SIZE;
    array_of_distrib[i] = MPI_DISTRIBUTE_BLOCK;
    array_of_dargs[i] = MPI_DISTRIBUTE_DFLT_DARG;
    array_of_psizes[i] = 0;
}
MPI_Dims_create(pool_size, ndims, array_of_psizes);
order = MPI_ORDER_C;

```

The number of dimensions both of the process grid, which is yet to be constructed and of the data array itself is set to `NDIMS`, which is defined to be 3. So all these arrays used in the call to `MPI_Type_create_darray` are going to be of length 3. The first array, `array_of_gsizes`, specifies the sizes of the global data array, and we set it here to be  $512 \times 512 \times 512$ , because this is what `SIZE` is defined to be. All entries in the `array_of_distrib` are set to



`MPI_DISTRIBUTE_BLOCK`, which means that we want block distribution, rather than cyclic distribution of data amongst the processes. Since the block distribution request doesn't take any arguments, all entries in the `array_of_dargs` are set to `MPI_DISTRIBUTE_DFLT_DARG`. Finally we set all entries to the array that specifies the process grid geometry to zeros. This is because this array is going to be configured by the call to `MPI_Dims_create`. This function divides a process pool into an `ndims`-dimensional grid of processes, and returns the number of processes in each direction of the grid on `array_of_psize`s. The total number of processes must be such that they can be indeed organized into a grid. Otherwise this function will return an error. The `order` parameter is set to `MPI_ORDER_C`, which means that the global data matrix is going to be organized in the C-language style, i.e., row-major.

If the program is run in the verbose mode, all processes write the values of all these parameters on standard output, for example:

```
4: calling MPI_Type_create_darray with
4:   pool_size      = 27
4:   my_rank       = 4
4:   ndims         = 3
4:   array_of_gsizes = (512, 512, 512)
4:   array_of_distribs = (121, 121, 121)
4:   array_of_dargs = (-49767, -49767, -49767)
4:   array_of_psize = (3, 3, 3)
4:   order         = 56
4:   type          = 1275069445
```

You have to lookup `/N/hpc/mpich2/include/mpi.h` to see that, e.g., `-49767` is indeed `MPI_DISTRIBUTE_DFLT_DARG` and that `121` is indeed `MPI_DISTRIBUTE_BLOCK` and that `56` is indeed `MPI_ORDER_C` and that `1275069445` is indeed `MPI_INT`.

Now the program finally calls `MPI_Type_create_darray` and commits the new returned MPI data type called `file_type`:

```
MPI_Type_create_darray(pool_size, my_rank, ndims,
                      array_of_gsizes, array_of_distribs,
                      array_of_dargs, array_of_psize, order,
                      MPI_INT, &file_type);
MPI_Type_commit(&file_type);
```

There are two important numbers that characterize this new MPI type, its *extent* and its *size*. The extent of an MPI type is the distance in bytes between the upper and the lower marker of the type. The size of an MPI type is the total amount of non-trivial data contained in the type, also in bytes. In other words, the size is the extent of the type minus the padding. We can extract these two numbers from the newly defined type by calling functions `MPI_Type_extent` and `MPI_Type_size`:

```
MPI_Type_extent(file_type, &file_type_extent);
MPI_Type_size(file_type, &file_type_size);
if (verbose) {
    printf("%3d: file_type_size   = %d\n", my_rank, file_type_size);
    printf("%3d: file_type_extent = %d\n", my_rank, file_type_extent);
}
```

It is instructive to inspect the output of the program in this place:

```
[...]
8: file_type_size   = 19767600
8: file_type_extent = 536870912
4: file_type_size   = 20000844
4: file_type_extent = 536870912
[...]
```

Observe that the *extent* of `file_type` is 536,870,912 bytes, which is the size of the file:

```
gustav@bh1 $ ls -l /N/gpfs/gustav/darray
total 524288
-rw-r--r--  1 gustav  ucs      536870912 Oct 26 17:49 test
gustav@bh1 $
```

but the *size* of `file_type` differs from process to process and corresponds to the amount of data this process is going to write on the file, when the view is established. This is how the processes divide the file amongst themselves. This is done without having to manipulate processes' local pointers explicitly.

But there is a price to this procedure, which makes it somewhat impractical on systems such as the AVIDD cluster. The price is that the second argument in function `MPI_Type_extent` must be of type `MPI_Aint`, and `MPI_Aint` is defined on the 32-bit systems to be simply `int`, because the idea here is that you ought to be able to absorb a data item of this particular type into the memory of your computer. This means that function `MPI_Type_create_darray` as well as `MPI_Type_extent` will fail if the total amount of data you want to partition exceeds `INT_MAX`, i.e., 2,147,483,647 bytes. This, however, is very little data by supercomputer standards. It is in places like this one that the limitations of 32-bit architectures can cause real pain. Luckily, we have a 64-bit system available to us at IU, it is our Research SP. There is also a small component of the AVIDD cluster at IUPUI, which comprises IA64 nodes.

Once data partitioning has been returned to the processes, they can allocate required amount of storage for the data:

```
write_buffer_size = file_type_size / sizeof(int);
write_buffer = (int*) malloc(write_buffer_size * sizeof(int));

/* We do this in case sizeof(int) does not divide file_type_size
   exactly. But this should not happen if we have called
   MPI_Type_create_darray with MPI_INT as the original data
   type. */

if (! write_buffer) {
    sprintf(message, "%3d: malloc write_buffer", my_rank);
    perror(message);
    MPI_Abort(MPI_COMM_WORLD, errno);

    /* We can still abort, because we have not opened any
       files yet. Notice that since MPI_Type_create_darray
       will fail if SIZE^3 * sizeof(int) exceeds MAX_INT,
```

```

        because MPI_Aint on AVIDD is a 32-bit integer,
        we are rather unlikely to fail on this malloc
        anyway. */
    }

    MPI_Barrier(MPI_COMM_WORLD);
    /* We wait here in case some procs have problems with malloc. */

```

Observe that we are not going to proceed with opening the file until all processes meet at the barrier, implying that none had problems with `malloc`. Now they fill their memory buffers with numbers:

```

    for (i = 0; i < write_buffer_size; i++)
        *(write_buffer + i) = my_rank * SIZE + i;

```

Observe that this will result in each process filling its buffer with different numbers. We need this in order to check, towards the end of the program, that data read back from the file is identical to data that has been written on it in the first place.

Now we open the file and immediately check for a possible problem:

```

    file_open_error = MPI_File_open(MPI_COMM_WORLD, file_name,
                                    MPI_MODE_CREATE | MPI_MODE_WRONLY,
                                    MPI_INFO_NULL, &fh);
    if (file_open_error != MPI_SUCCESS) {
        MPI_Error_string(file_open_error, error_string,
                        &error_string_length);
        fprintf(stderr, "%3d: %s\n", my_rank, error_string);
        MPI_Abort(MPI_COMM_WORLD, file_open_error);

        /* It is still OK to abort, because we have failed to
           open the file. */
    }

```

If there is a problem on `open` we can still abort the program without having to worry about clean-up. Function `MPI_File_open` is collective, which means that if anybody has a problem opening the file, error will be returned to all of them – and the file won't be opened.

The rest of the program is contained within the `else` clause of the `if` statement:

```

    else {
        blah... blah... blah...
    } /* no problem with file open */
} /* no problem with input error */
MPI_Finalize();
exit(0);
}

```

The first thing that happens within this clause is that the master process changes permissions on the successfully opened file from `rw-rw-rw-` to `rw-r--r--`, while other processes wait on the barrier:

```

    if (i_am_the_master)
        chmod(file_name, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
    MPI_Barrier(MPI_COMM_WORLD);

```

This shouldn't be necessary, because MPI provides means for changing permissions on a newly created file by the means of `info` hints, but MPICH2 hasn't implemented this part of the standard, so we have to rely on UNIX to do this ourselves.

Now we convert our MPI type `file_type` into an official view of the file:

```
MPI_File_set_view(fh, 0, MPI_INT, file_type, "native", MPI_INFO_NULL);
```

and write the data. This time we call the collective form of `MPI_File_write`, which is `MPI_File_write_all`. It works much like `MPI_File_write`, but this time all processes must do it at the same time, and any local errors become global errors automatically. Consequently, if any of the processes fails to write the data, all will know about it and all will enter the following clause of the `if` statement:

```
file_write_error =
    MPI_File_write_all(fh, write_buffer, write_buffer_size, MPI_INT,
                      &status);
if (file_write_error != MPI_SUCCESS) {
    MPI_Error_string(file_write_error, error_string,
                    &error_string_length);
    fprintf(stderr, "%3d: %s\n", my_rank, error_string);
    MPI_File_close(&fh);
    free(write_buffer);
    if (i_am_the_master) MPI_File_delete(file_name, MPI_INFO_NULL);
}
```

If there is an error, all processes close the file – this is a collective call too, and then the master process deletes it.

The rest of the program is again in the form of the `else` statement:

```
else {
    blah... blah... blah...
} /* no problem with file write */
} /* no problem with file open */
} /* no input error */

MPI_Finalize();
exit(0);
}
```

The first thing that happens here is that every process inspects the `status` returned by `MPI_File_write_all` for the number of data written on the file (it should be the same as intended, but this is a yet another way to check it), and the size of the whole file. These numbers should agree with numbers obtained from the call to `MPI_Type_construct_darray`:

```
MPI_Get_count(&status, MPI_INT, &count);
MPI_File_get_size(fh, &file_size);
if(verbose) {
    printf("%3d: wrote %d integers\n", my_rank, count);
    printf("%3d: file size is %lld bytes\n", my_rank, file_size);
}
```

and afterwards the file gets closed:

```
MPI_File_close(&fh);
```

The last part of the program opens the file again, but this time for reading. We allocate space for the reading buffer first, check if there are no problems with malloc and then open the file for reading only:

```

read_buffer_size = write_buffer_size;
read_buffer = (int*) malloc(read_buffer_size * sizeof(int));
if (! read_buffer) {
    sprintf(message, "%3d: malloc read_buffer", my_rank);
    perror(message);
    MPI_Abort(MPI_COMM_WORLD, errno);

    /* We can abort, because the file has been closed and
       we haven't opened it for reading yet. */
}

MPI_Barrier(MPI_COMM_WORLD);
/* We wait here in case some procs have problems with malloc. */

MPI_File_open(MPI_COMM_WORLD, file_name, MPI_MODE_RDONLY,
              MPI_INFO_NULL, &fh);

```

Next we establish the same view of the file as before, read all data back from it, every process checks how much data has really been read by inspecting the status, and then the file gets closed again:

```

MPI_File_set_view(fh, 0, MPI_INT, file_type, "native", MPI_INFO_NULL);
MPI_File_read_all(fh, read_buffer, read_buffer_size, MPI_INT, &status);
MPI_Get_count(&status, MPI_INT, &count);
if (verbose)
    printf("%3d: read %d integers\n", my_rank, count);
MPI_File_close(&fh);

```

Now every process compares its read\_buffer with its write\_buffer, which we have wisely saved, and the result of the comparison is reduced on the variable read\_error maintained by the master process:

```

for (i = 0; i < read_buffer_size; i++) {
    if (*(write_buffer + i) != *(read_buffer + i)) {
        printf("%3d: data read different from data written, i = %d\n",
              my_rank, i);
        my_read_error = TRUE;
    }
}

MPI_Reduce (&my_read_error, &read_error, 1, MPI_INT, MPI_LOR,
           MASTER_RANK, MPI_COMM_WORLD);

```

If the read and write buffers contain identical data for all processes in the pool, the master process prints the message about it:

```

if (i_am_the_master)
    if (! read_error)
        printf("--> All data read back is correct.\n");

```

And this is where the action ends. All processes then meet at MPI\_Finalize and exit.

### Exercises

- 1 What is the largest size of `SIZE`, for which `MPI_Type_construct_darray` is still going to work? Test it using the program discussed in section 5.3.4.
- 2 Would it help if we replaced `MPI_INT` with a more elaborate structure that would contain more data?
- 3 The restriction on the extent of `file_type` in section 5.3.4 does not imply the restriction on the size of the file. It is only a restriction on the amount of data that can be written on the file in a *single* `MPI_File_write` operation. But this operation can be repeated. Rewrite the program discussed in section 5.3.4 in the following way:
  - Instead of one array `write_buffer` and `read_buffer` define and malloc several, e.g., `write_buffer_1`, `write_buffer_2`, etc., of identical size, but make sure that you have enough memory for them all. Aim for not more than a GB of memory per process.
  - Populate the `write_buffer_?` arrays with random integers, using a different random number generator seed for every process.
  - Write them all on the file in multiple `MPI_File_write_all` operations, one after another. Time the whole data transfer and print MB/s obtained on standard output.
  - Read them back from the file on the `read_buffer_?` arrays and compare every entry with the corresponding entry in the `write_buffer_?` arrays. Time the data transfer and print MB/s obtained on standard output.
  - Replace `MPI_DISTRIBUTE_BLOCK` with `MPI_DISTRIBUTE_CYCLIC` and compare the data transfer rates.
  - Run the program on up to 32 nodes *under PBS*, generating files up to 16GB long. Remember that half of the 1GB memory per process will have to be reserved for `read_buffer_?`.

### 5.3.5 File Hints

Parallel files may be stored and accessed in great many ways that depend on the operating system, particular devices used for storage and any middleware that may live in between. In order to optimize file access the programmer may wish to provide additional information to MPI, in hope that MPI would know what to do with it. Such information is referred to as *hints* and there is a special MPI construct called the *info* object that is supposed to collect all the hints. Once constructed the info object can be passed to `MPI_File_open`, `MPI_File_delete`, `MPI_File_set_view` and `MPI_File_set_info`. It should be understood though that any hints you may wish to give MPI this way are only advisory and what MPI is going to do with them is implementation dependent.

The info object is opaque. MPI provides functions for its manipulation and inspection, but this is the only way you can (or should) deal with it. Information contained in the info object consists of (*key*, *value*) pairs – both are strings. Their maximum length allowed by an MPI implementation is given by constants `MPI_MAX_INFO_KEY` and `MPI_MAX_INFO_VAL`, which can be looked up on `mpi.h`. Their values on the AVIDD cluster are:

```
gustav@bh1 $ grep MAX_INFO mpi.h
#define MPI_MAX_INFO_KEY      255
#define MPI_MAX_INFO_VAL     1024
gustav@bh1 $
```

An empty info object can be created with the call to `MPI_Info_create`, whose synopsis is simply:

```
int MPI_Info_create(MPI_Info *info)
```

Once created, it can be populated with (*key*, *value*) pairs by calling `MPI_Info_set` whose synopsis is

```
int MPI_Info_set(MPI_Info info, char *key, char *value)
```

An info object obtained, e.g., from an MPI file, can be interrogated in various ways. First you can find how many (*key*, *value*) pairs there are in the object by calling function `MPI_Info_get_nkeys` which returns the number of keys currently defined within the object in its second argument:

```
int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)
```

Once you know how many (*key*, *value*) pairs there are in it, you can inspect them by calling first `MPI_Info_get_nthkey` in order to get the string that corresponds to the  $n^{\text{th}}$  key – the synopsis of this function is:

```
int MPI_Info_get_nthkey(MPI_Info info, int n, char *key)
```

and then `MPI_Info_get` to get the value associated with the key. The synopsis of `MPI_Info_get` is:

```
int MPI_Info_get(MPI_Info info, char *key, int valuelen, char *value, int *exists)
```

where `exists` is a Boolean flag that is set to `TRUE` (1) if the corresponding (*key*, *value*) pair exists and to `FALSE` (0) otherwise.

An info object can be duplicated and it can be also erased by freeing it. Specific (*key*, *value*) pairs may be deleted from an existing info object too.

MPI reserves certain hints, which are described in section 9.2.8.1, Reserved File Hints, of the MPI-2 Standard. Unfortunately the current MPICH2 implementation of MPI-2 ignores most of them, including `file_perm`, which would come handy in some of our example programs. The reserved hints cover parameters such as *striping factor*, *striping unit*, *number of IO nodes* (there are systems where not all nodes have IO capability), *list of IO devices* to store the file on, various *chunking* parameters, the size of *buffers* to be used for reading and writing, and the like.

Because the GPFS on the AVIDD system is accessed only through its UFS interface (MPI doesn't really know that GPFS, is GPFS), few of the reserved hints would be of much use anyway, even if MPICH2 did recognize them.

When a file is opened with the `MPI_INFO_NULL` parameter passed in place of an info object, a default info is used by MPI. The following program retrieves this info object and queries it. This shows us some of the info (*key*, *value*) pairs that are implemented by MPICH2 on the UFS at present.

**Program queryinfo.c**

Here is the listing of the program. The program lives in `/N/B/gustav/src/I590/queryinfo.`

```

/*
 * %Id: queryinfo.c,v 1.1 2003/10/27 18:19:10 gustav Exp %
 *
 * %Log: queryinfo.c,v %
 * Revision 1.1 2003/10/27 18:19:10 gustav
 * Initial revision
 *
 */

#include <stdio.h>      /* printf and relatives live here */
#include <stdlib.h>     /* exit lives here */
#include <unistd.h>     /* getopt lives here */
#include <string.h>     /* strlen lives here */
#include <mpi.h>        /* all MPI stuff lives here (including MPI-IO) */

#define MASTER_RANK 0
#define TRUE 1
#define FALSE 0
#define BOOLEAN int
#define SYNOPSIS printf ("synopsis: %s -f <file> [-v] [-h]\n", argv[0])

int main(argc, argv)
    int argc;
    char *argv[];
{
    /* my variables */

    int my_rank, pool_size, i, file_name_length;
    BOOLEAN i_am_the_master = FALSE, input_error = FALSE, verbose = FALSE;
    char *file_name = NULL;

    /* MPI variables */

    MPI_File fh;
    MPI_Info info_used;
    int nkeys;
    BOOLEAN exists;
    char key[MPI_MAX_INFO_KEY], value[MPI_MAX_INFO_VAL];

    /* getopt variables */

    extern char *optarg;
    int c;

    /* ACTION */

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &pool_size);
    if (my_rank == MASTER_RANK) i_am_the_master = TRUE;
    if (i_am_the_master) {
        while ((c = getopt(argc, argv, "f:vh")) != EOF)
            switch(c) {

```



```

    case 'f':
        file_name = optarg;
        break;
    case 'v':
        verbose = TRUE;
        break;
    case 'h':
        input_error = TRUE;
        break;
    case '?':
        input_error = TRUE;
        break;
    }
if (file_name == NULL) input_error = TRUE;
if (input_error)
    SYNOPSIS;
else {
    file_name_length = strlen(file_name) + 1;
    if (verbose) {
        printf("file_name          = %s\n", file_name);
        printf("file_name_length = %d\n", file_name_length);
    }
}
}

MPI_Bcast(&input_error, 1, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);

if (! input_error) {
    MPI_Bcast(&verbose, 1, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);
    MPI_Bcast(&file_name_length, 1, MPI_INT, MASTER_RANK, MPI_COMM_WORLD);
    if (! i_am_the_master) file_name = (char*) malloc(file_name_length);
    MPI_Bcast(file_name, file_name_length, MPI_CHAR, MASTER_RANK,
              MPI_COMM_WORLD);

    MPI_File_open(MPI_COMM_WORLD, file_name, MPI_MODE_CREATE | MPI_MODE_RDWR,
                 MPI_INFO_NULL, &fh);
    MPI_File_get_info(fh, &info_used);
    MPI_Info_get_nkeys(info_used, &nkeys);

    for (i = 0; i < nkeys; i++) {
        MPI_Info_get_nthkey(info_used, i, key);
        MPI_Info_get(info_used, key, MPI_MAX_INFO_VAL, value, &exists);
        if (verbose)
            if (exists)
                printf("%3d: key = %s, value = %s\n", my_rank, key, value);
    }

    MPI_File_close(&fh);
    MPI_Info_free(&info_used);
}

MPI_Finalize();
exit(0);
}

```

The program is made and installed as follows:

```
gustav@bh1 $ pwd
```

```

/N/B/gustav/src/I590/queryinfo
gustav@bh1 $ make install
co RCS/Makefile,v Makefile
RCS/Makefile,v --> Makefile
revision 1.1
done
co RCS/queryinfo.c,v queryinfo.c
RCS/queryinfo.c,v --> queryinfo.c
revision 1.1
done
mpicc -o queryinfo queryinfo.c
install queryinfo /N/B/gustav/bin
gustav@bh1 $

```

It can be run anywhere. It creates a file, but it doesn't write anything on it. Here is a fragment of the output generated by the program when it is run with the `-v` option.

```

gustav@bh1 $ mpiexec -n 4 queryinfo -f test -v
file_name           = test
file_name_length    = 5
 0: key = cb_buffer_size, value = 4194304
 0: key = romio_cb_read, value = automatic
 0: key = romio_cb_write, value = automatic
 0: key = cb_nodes, value = 4
 0: key = romio_no_indep_rw, value = false
 0: key = ind_rd_buffer_size, value = 4194304
 0: key = ind_wr_buffer_size, value = 524288
 0: key = romio_ds_read, value = automatic
 0: key = romio_ds_write, value = automatic
 0: key = cb_config_list, value = *:1
[...]
gustav@bh1 $

```

## The Discussion

The program begins the same way our previous programs have. The master process reads the command line and then passes the options to other processes. Eventually a file is opened with the call to `MPI_File_open`. But we don't write anything on it in this program. Instead we extract the default info object from it by calling function `MPI_File_get_info`:

```
MPI_File_get_info(fh, &info_used);
```

Whatever info has been used on opening the file is returned on `info_used`. Now we ask about the number of (*key*, *value*) pairs inside the info by calling function `MPI_Info_get_nkeys`

```
MPI_Info_get_nkeys(info_used, &nkeys);
```

The number of pairs is returned on `nkeys`. Now we can loop through all the pairs and obtain the key string for each pair first, and then having the key string, we can obtain the value associated with the key. This is done by calling `MPI_Info_get_nthkey` to get the key and then `MPI_Info_get` to get the value.

```

for (i = 0; i < nkeys; i++) {
    MPI_Info_get_nthkey(info_used, i, key);
    MPI_Info_get(info_used, key, MPI_MAX_INFO_VAL, value, &exists);
    if (verbose)

```

```

        if (exists)
            printf("%3d: key = %s, value = %s\n", my_rank, key, value);
    }

```

In this case every key we pass to `MPI_Info_get` *exists*, because we got it from the info object in the first place with the call to `MPI_Info_get_nthkey`, but `MPI_Info_get` may be used in other ways, e.g., just to check whether a particular (*key*, *value*) pair is at all defined within the info, and then the `exists` flag comes handy.

And this is all the program does. The file gets closed, the info gets freed, the processes hit `MPI_Finalize` and exit:

```

        MPI_File_close(&fh);
        MPI_Info_free(&info_used);
    } /* end of the for loop */

    MPI_Finalize();
    exit(0);
}

```

A typical output from a process that has just opened on MPI file on our GPFS is going to look as follows:

```

2: key = cb_buffer_size, value = 4194304
2: key = romio_cb_read, value = automatic
2: key = romio_cb_write, value = automatic
2: key = cb_nodes, value = 4
2: key = romio_no_indep_rw, value = false
2: key = ind_rd_buffer_size, value = 4194304
2: key = ind_wr_buffer_size, value = 524288
2: key = romio_ds_read, value = automatic
2: key = romio_ds_write, value = automatic
2: key = cb_config_list, value = *:1

```

The `cb_buffer_size` and `cb_nodes` are the only hints reserved by the MPI-2 standard. All the other hints here are the implementation hints. In particular we find that the size of the buffer space that can be used for collective buffering on each target node (`cb_buffer_size`) is 4MB. We also find that the default value for `ind_wr_buffer_size` is pretty small, only 0.5MB. You may try to increase this value to 4194304 by calling function `MPI_Info_set`:

```
int MPI_Info_set(MPI_Info info, char *key, char *value)
```

note that the value has to be a character string, not an integer, and then passing the modified info back to the file with `MPI_File_set_info`:

```
int MPI_File_set_info(MPI_File fh, MPI_Info info)
```

## Exercises

- 1 Rewrite program `mkrandpfile.c` discussed in section 5.3.1, to make use of hints. Before you open the file, construct an info object for it taking all the default values discussed in the previous section, but change the value of `ind_wr_buffer_size` to 4194304. After you have opened the file, read its info back and verify that the change has taken effect. Does the change affect the data transfer rate?

### 5.3.6 Asynchronous IO

All message passing operations and file access operations we have discussed in this course so far are blocking. This means that when, e.g., you issue a call to `MPI_Send`, the call returns *only* after all the data in the send buffer has been sent, meaning that it is now safe to perform other operations on the send buffer, e.g., you may write to it again. Similarly when you issue a call to `MPI_Recv`, the call returns *only* after all the data that you expect to receive in the receive buffer has been written on the buffer, meaning that it is now safe to perform other operations on the receive buffer, e.g., it is safe to read it.

The file IO semantics are similar. The blocking IO operations such as `MPI_File_write` and `MPI_File_read` do not return until all the data has been taken out of the write buffer, or all the data has been written onto the read buffer, so that it is now safe to use or re-use either one or the other.

Blocking of these functions calls is *local*, i.e., they block only for as long as the send or write or receive or read buffers are in use by the communication functions. There is another type of blocking, which is markedly more *severe*. It is called *synchronous* or *global* blocking. If you send a message with `MPI_Ssend`, the function will return only *after* a matching receive has been activated on the other side, and the receive process has started reading the data into its receive buffer.

On the other hand we also have totally non-blocking operations such as `MPI_Isend` and `MPI_Irecv`, which merely initiate the send or the receive and return right away, even as their send or receive buffers are still being used by data transfer operations. Of course, while the transfer is under way, you must not touch the buffers.

What do we need such non-blocking operations for? The reason for their existence is that message passing and file access operations are *extremely* slow by computing standards. In the time it take to read data from a file, or to send data to other processes, you may be able to perform thousands, even millions of arithmetic operations. So if every nano-second counts, you want to be able to do just this: compute, while data transfer operations execute in the background.

But how are you going to know that a particular data transfer operation you have initiated has completed?

All non-blocking MPI functions take an additional argument of type `MPI_Request`. It is a yet another opaque MPI data type. Once you have initiated a data transfer you get this `request` back and you can then call `MPI_Test`, which takes your `request` as an argument and checks whether the corresponding communication operation has completed. The synopsis of `MPI_Test` is:

```
int MPI_Test(MPI_Request *request, int *completed, MPI_Status *status)
```

The value of `completed` is set to `TRUE` when the communication operation pointed to by `request` has indeed completed. Otherwise `completed` is `FALSE`. Additionally the `status` variable may be inspected for other details pertaining to the operation, e.g., the rank number of a sender process or the number of items received.

If you have finished all you wanted to do while the data is still being transferred you can instead issue the call to `MPI_Wait`, the synopsis of which is

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

There is no `completed` in `MPI_Wait`. This function returns *only* after the operation pointed to by the `request` has completed.

MPI-IO supports similar non-blocking functions for writing on and reading from files. The functions are `MPI_File_iread` and `MPI_File_iwrite`. Their respective synopses are:

```
int MPI_File_iwrite(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)
int MPI_File_iread(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Request *request)
```

Observe that unlike `MPI_File_read`, function `MPI_File_iread` does not take `status` as an argument. You have to call `MPI_Wait` or `MPI_Status` to get hold of `status` in this case. The reason for this should be obvious. When `MPI_File_iread` returns, there is no status yet to read. It will only come to existence *after* the reading operation has completed.

There are *no* asynchronous versions of collective file access operations like `MPI_File_read_all` and `MPI_File_write_all`.

You can actually express `MPI_File_write` and `MPI_File_read` as combinations of `MPI_File_iwrite` and `MPI_File_iread` with `MPI_Wait`. The following:

```
MPI_File_iwrite(fh, buf, count, datatype, &request);
MPI_Wait(&request, &status);
```

is equivalent to

```
MPI_File_write(fh, buf, count, datatype, &status);
```

and

```
MPI_File_iread(fh, buf, count, datatype, &request);
MPI_Wait(&request, &status);
```

is equivalent to

```
MPI_File_read(fh, buf, count, datatype, &status);
```

### Exercises

- 1 Rewrite program `mkrandpfile.c` discussed in section 5.3.1 to use the non-blocking `MPI_File_iwrite`. Use function `MPI_Test` to test if the writing process has completed every now and then, while calculating square roots of integers from 1 to 1,000,000 in the meantime. Square root is a very expensive, i.e., slow, arithmetic operation.

## 5.4 MPI Graphics and Process Visualization

In this section we are going to learn about an auxiliary library, called MPE, for the analysis and visualization of MPI programs. The library can be also used to visualize data handled by MPI programs – although its graphic capabilities are very simple.

We begin with the discussion of how to “MPE-instrument” MPI programs, and how to view the logs generated thusly. There are various auxiliary tools for

processing the logs and one huge Java program for viewing them in a graphical format.

Then we'll look at the graphics component of MPE and illustrate its use by discussing some simple (and then less simple) example programs.

The MPE library and utilities are not as well documented as MPI itself. But you can download a fairly readable, if terse, MPE manual, <ftp://ftp.mcs.anl.gov/pub/mpi/mpeman.pdf>, from Argonne. The PDF file has a January 2003 time stamp on it, the last I have had a look at it, but it is not entirely up to date with respect to MPICH2. MPE is also discussed in “Using MPI” by Gropp, Lusk and Skjellum, but this discussion may be a little old, because the book itself was published in 1994.

### 5.4.1 Instrumenting MPI Programs with MPE

You have seen a simple and quite effective example of program instrumentation in section 5.2.4 that talked about the Bank Queue paradigm. There we instrumented the program by writing logs of various operations on log files, one per each process.

The MPE library provides a much more sophisticated tools for logging various events and states in your program. These don't have to be MPI events or states only. You can use MPE logging for other events and states too. The logs are created on the run and stored in memory, so as to minimize the impact that logging may have on the program's performance. Only after the logging and the program have been completed, the actual log file is written.

Unlike our simple logging example in section 5.2.4, MPE logging is accompanied by very precise time stamps. This can help solve some intricate communication and synchronization problems as well as optimize communications and computations within the program. So MPE logging can be used not only to debug a parallel program, but also to profile its execution.

MPE logging can be “by default”, in which case just about everything that happens in the program is logged, or it can be customized. The latter is more useful for a programmer, the former is simpler, because the programmer doesn't have to do anything. All that's required is to link the program with the `-lmpe` library.

Suppose you would like to log all MPI events generated by program `cpi.c` discussed in section 5.2.3. This program comes from the MPICH2 examples that are distributed together with the code. To generate the logging version of the program compile it as follows:

```
gustav@bh1 $ pwd
/N/B/gustav/src/I590/cpi
gustav@bh1 $ mpicc -o cpi cpi.c -lmpe -lmpe
gustav@bh1 $
```

When the program is run, it should generate a file called `cpi.clog` in your working directory:

```
gustav@bh1 $ pwd
/N/B/gustav
gustav@bh1 $ mpdtrace | wc
      27      27     134
```

```

gustav@bh1 $ mpiexec -n 8 cpi
Process 0 of 8 is on bh1
Process 1 of 8 is on bc30
Process 2 of 8 is on bc23
Process 6 of 8 is on bc37
Process 4 of 8 is on bc36
Process 3 of 8 is on bc35
Process 5 of 8 is on bc34
Process 7 of 8 is on bc27
pi is approximately 3.1415926544231247, Error is 0.000000008333316
wall clock time = 0.042839
Writing logfile.
Finished writing logfile.
gustav@bh1 $ ls *log*
cpi.clog
gustav@bh1 $

```

This file is written in the “native” format and is not readable without auxiliary tools, but it can be converted to the human readable `alog` format by calling the `clog2alog` utility:

```

gustav@bh1 $ clog2alog cpi
gustav@bh1 $ ls *cpi*
cpi.alog  cpi.clog
gustav@bh1 $

```

You don’t need to supply the `.clog` suffix, because the program adds it automatically to the file basename you provide it with on the command line. The `alog` file contains an ASCII head followed by an ASCII listing of all events and states logged by the program. This is what it may look like:

```

gustav@bh1 $ cat cpi.alog
-1 0 0 0 0 0 Me
-2 0 0 35 0 0
-3 0 0 8 0 0
-4 0 0 1 0 0
-5 0 0 5 0 0
-6 0 0 0 0 0
-7 0 0 0 0 42890
-8 0 0 1 0 0
-11 0 0 0 0 0
-13 0 13 14 0 0 cyan:boxes BCAST
-13 0 25 26 0 0 purple:2x2 REDUCE
-201 0 0 -1 0 0 MPI_PROC_NULL
-201 0 0 -2 0 0 MPI_ANY_SOURCE
-201 0 0 -1 0 1 MPI_ANY_TAG
13 0 0 1 0 55
14 0 0 2 0 83
25 0 0 1 0 108
13 1 0 1 0 4088
14 1 0 2 0 4114
25 1 0 1 0 4138
13 6 0 1 0 7042
13 2 0 1 0 7290
14 2 0 2 0 7327
25 2 0 1 0 7351
13 4 0 1 0 8855
14 4 0 2 0 8938
25 4 0 1 0 8966

```

```

14 6 0 2 0 9108
25 6 0 1 0 9137
13 3 0 1 0 15425
14 3 0 2 0 15466
25 3 0 1 0 15491
13 5 0 1 0 15973
14 5 0 2 0 16034
25 5 0 1 0 16060
13 7 0 1 0 16155
14 7 0 2 0 16207
25 7 0 1 0 16231
26 1 0 2 0 25906
26 3 0 2 0 41456
26 2 0 2 0 41716
26 7 0 2 0 41867
26 6 0 2 0 42135
26 5 0 2 0 42372
26 4 0 2 0 42565
26 0 0 2 0 42890
gustav@bh1 $

```

Even though the format of the file is human readable, the file is still rather cryptic and not very helpful.

A better way is to convert the `clog` file to an `slog2` file with the `clogT0slog2` utility (here you have to type the full name of the file including its extension):

```

gustav@bh1 $ clogT0slog2 cpi.clog
GUI_LIBDIR is set. GUI_LIBDIR = \
    /N/B/gustav/src/mpich2-0.94b1/src/mpe/slog2sdk/lib
SLOG-2 Header:
version = SLOG 2.0.5
NumOfChildrenPerNode = 2
TreeLeafByteSize = 65536
MaxTreeDepth = 0
MaxBufferByteSize = 574
Categories is FBinfo(94 @ 682)
MethodDefs is FBinfo(0 @ 0)
LineIDMaps is FBinfo(100 @ 776)
TreeRoot is FBinfo(574 @ 108)
TreeDir is FBinfo(38 @ 876)
Annotations is FBinfo(0 @ 0)
Postamble is FBinfo(0 @ 0)

Number of Drawables = 16
Number of Unmatched Events = 0
Total ByteSize of the logfile = 3688
timeElapsed between 1 & 2 = 873 msec
timeElapsed between 2 & 3 = 50 msec
gustav@bh1 $ ls *log*
cpi.alog cpi.clog cpi.slog2
gustav@bh1 $

```

and then run program `jumpshot` on the `slog2` file.

`Jumpshot` is a rather large (and so heavy that it is quite unusable over an ISDN line) Java program that reads the `slog2` file and displays events and states of the program in the form of multiple long horizontal “thermometers”,



each “thermometer” corresponding to a separate process. Message passing is symbolized by arrows that connect various “thermometers”. This way you can see exactly when a message has been sent and when it has been received. Various states of the program are symbolized by various colours of the “thermometer” lines. For example, computation may be symbolized by the “thermometer” going red, whereas waiting for data may be symbolized by, say, black or gray. The idea behind an efficient parallel program is to have as much red and as little black as possible. The window of `jumpshot` that displays all this is called the Timeline Window.

A long program, which involves a lot of communication may produce a very dense picture in the Timeline Window that is going to be difficult to read at first glance. You can use a magnifying glass gizmo, invoked from the bottom right corner of the window, to stretch fragments of the log, so that you can see what happens there in greater detail. There is also another magnifying glass in the tool bar above the window, which can be used to select specific portions of the log for a closer look.

In order to customize the logging, you have to instrument your program explicitly. Here is the example of `cpilog.c`, obtained from the MPE component of the MPICH2 source, instrumented this way:

```
gustav@bh1 $ pwd
/N/B/gustav/src/I590/mpe
gustav@bh1 $ cat cpilog.c
#include <mpi.h>
#include <mpe.h>
#include <math.h>
#include <stdio.h>

double f( double );
double f( double a)
{
    return (4.0 / (1.0 + a*a));
}

int main( int argc, char *argv[])
{
    int n, myid, numprocs, i, j;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;
    double startwtime = 0.0, endwtime;
    int namelen;
    int event1a, event1b, event2a, event2b,
        event3a, event3b, event4a, event4b;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    MPI_Get_processor_name(processor_name,&namelen);
    fprintf(stderr,"Process %d running on %s\n", myid, processor_name);

    MPE_Init_log();
```

```

/* Get event ID from MPE, user should NOT assign event ID */
event1a = MPE_Log_get_event_number();
event1b = MPE_Log_get_event_number();
event2a = MPE_Log_get_event_number();
event2b = MPE_Log_get_event_number();
event3a = MPE_Log_get_event_number();
event3b = MPE_Log_get_event_number();
event4a = MPE_Log_get_event_number();
event4b = MPE_Log_get_event_number();

if (myid == 0) {
MPE_Describe_state(event1a, event1b, "Broadcast", "red");
MPE_Describe_state(event2a, event2b, "Compute", "blue");
MPE_Describe_state(event3a, event3b, "Reduce", "green");
MPE_Describe_state(event4a, event4b, "Sync", "orange");
}

if (myid == 0)
{
n = 1000000;
startwtime = MPI_Wtime();
}
MPI_Barrier(MPI_COMM_WORLD);

MPE_Start_log();

for (j = 0; j < 5; j++)
{
MPE_Log_event(event1a, 0, "start broadcast");
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPE_Log_event(event1b, 0, "end broadcast");

MPE_Log_event(event4a,0,"Start Sync");
MPI_Barrier(MPI_COMM_WORLD);
MPE_Log_event(event4b,0,"End Sync");

MPE_Log_event(event2a, 0, "start compute");
h = 1.0 / (double) n;
sum = 0.0;
for (i = myid + 1; i <= n; i += numprocs)
{
x = h * ((double)i - 0.5);
sum += f(x);
}
mypi = h * sum;
MPE_Log_event(event2b, 0, "end compute");

MPE_Log_event(event3a, 0, "start reduce");
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
MPE_Log_event(event3b, 0, "end reduce");
}

MPE_Finish_log("cpilog");

if (myid == 0)
{
endwtime = MPI_Wtime();
}

```

```

printf("pi is approximately %.16f, Error is %.16f\n",
      pi, fabs(pi - PI25DT));
printf("wall clock time = %f\n", endwtime-startwtime);
}
MPI_Finalize();
return(0);
}
gustav@bh1 $

```

The program should be linked with the `-lmpe` library *only* as follows:

```

gustav@bh1 $ mpicc -o cpilog cpilog.c -lmpe
gustav@bh1 $

```

Then, once it's run, it'll generate the log file on `cpilog.clog`, which can be then processed with `clogT0slog2` and viewed with `jumpshot`.

Let us have a closer look at this instrumented program.

The call to `MPE_Init_log()` initiates the MPE logging subsystem. This call has to be issued explicitly only if you are going to link the program with the `-lmpe` library. If you link the program with the `-llmpe -lmpe` libraries, the function is called by the `-llmpe` library right at the beginning of the program implicitly, but then *everything* is going to be logged, including the new stuff you have requested in your code and all the defaults as well. The advantage of starting and stopping the logging within your program explicitly is that you can decide to log only a specific part of the program or even to log various parts of it on various files.

Once the logging has been initialized we can define various events:

```

event1a = MPE_Log_get_event_number();
event1b = MPE_Log_get_event_number();
event2a = MPE_Log_get_event_number();
event2b = MPE_Log_get_event_number();
event3a = MPE_Log_get_event_number();
event3b = MPE_Log_get_event_number();
event4a = MPE_Log_get_event_number();
event4b = MPE_Log_get_event_number();

```

These calls to `MPE_Log_get_event_number` merely assign dynamically generated integer numbers to variables `event1a` through `event4b`. You could assign the numbers to these *events* manually and this is how it's been done in the past (see, e.g., "Using MPI"), but this is not a recommended practice today. At this stage we have only specified that we are going to have eight different events, but we haven't told MPE what these events are going to be yet.

Next we see the definition of *states* issued by the means of calls to function `MPE_Describe_state`:

```

if (myid == 0) {
    MPE_Describe_state(event1a, event1b, "Broadcast", "red");
    MPE_Describe_state(event2a, event2b, "Compute", "blue");
    MPE_Describe_state(event3a, event3b, "Reduce", "green");
    MPE_Describe_state(event4a, event4b, "Sync", "orange");
}

```

These definitions still don't define what the events themselves are, but by now the programmer has already made her choices. Events `event1a` and `event1b` correspond to the beginning and the end of broadcast. When the log is written and then read and processed by `jumpshot`, the duration of a broadcast will be symbolized by a horizontal red line in the Timeline Window, and the legend attached to the display will say that the red colour corresponds to a broadcast. Similarly, computation will be symbolized by blue, the `MPI_Reduce` will be symbolized by green and the duration of `MPI_Barrier` will be symbolized by orange. So, in this case a lot of blue means "good", and a lot of other colours mean "bad".

The state definitions need not be specified by more than just one process. All processes will log their events independently, but it is enough for MPE to read state definitions from one process only. These will then be applied to all other processes.

The log itself is started by the call to `MPE_Start_log()` and it is only now that we specify what the events `event1a` through `event4b` really mean, for example:

```
MPE_Log_event(event1a, 0, "start broadcast");
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPE_Log_event(event1b, 0, "end broadcast");
```

Function `MPE_Log_event` logs the event `event1a` labeled "start broadcast" just before calling `MPI_Bcast`, and the logs the event `event1b` with the label "end broadcast" just after function `MPI_Bcast` has returned. Everything between these two events will be marked with the red colour on the `jumpshot` display, as has been specified with the calls to `MPE_Describe_state`.

Of course, the programmer may well get it all wrong, which is exactly where automatic logging is advantageous.

The call to `MPI_Barrier` is similarly flanked with logged events:

```
MPE_Log_event(event4a,0,"Start Sync");
MPI_Barrier(MPI_COMM_WORLD);
MPE_Log_event(event4b,0,"End Sync");
```

and then we flank the whole computation section with events `event2a` and `event2b`:

```
MPE_Log_event(event2a, 0, "start compute");
h = 1.0 / (double) n;
sum = 0.0;
for (i = myid + 1; i <= n; i += numprocs)
{
    x = h * ((double)i - 0.5);
    sum += f(x);
}
mypi = h * sum;
MPE_Log_event(event2b, 0, "end compute");
```

Finally we flank the call to `MPI_Reduce` with `event3a` and `event3b`:

```
MPE_Log_event(event3a, 0, "start reduce");
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
MPE_Log_event(event3b, 0, "end reduce");
```

The call to

```
MPE_Finish_log("cpilog");
```

closes the log and writes it on the file `cpilog.clog`. This last call has to be issued explicitly only if you link the program with `-lmpe` only. If you were to link it with `-llmpe -lmpe` instead, the file would be generated automatically and its name would default to the name of the binary.

To summarize let me list the synopsis for each MPE log function discussed in this section:

```
int MPE_Init_log(void);
int MPE_Start_log(void);
int MPE_Stop_log(void);
int MPE_Finish_log(char *logfilename);
int MPE_Describe_state(int start, int end, char *name, char *color);
int MPE_Describe_event(int event, char *name);
int MPE_Log_event(int event, int intdata, char *chardata);
int MPE_Log_get_event_number(void);
```

### 5.4.2 MPE Graphics

MPE graphics are designed to display data in parallel from multiple MPI processes on a single X window. The routines provided are very simple, but sufficient for basic graphics and, in particular, for parallel display of images. Apart from drawing routines MPE also provides two graphics input routines that can be used to read the position of the mouse and to read coordinates of a rectangular region selected within the window by dragging the mouse.

Unfortunately you will find no manual entries that describe these functions in any detail in “User’s Guide for MPE” – they are merely mentioned there – but there are some examples of their use in “Using MPI” (see, e.g., section 5.3, “Visualizing the Mandelbrot Set” therein).

The Mandelbrot set example code can be found in

```
/N/hpc/mpich2/src/mpich2-0.94b1/src/mpe/contrib/mandel
```

but this code is rather hard to read. It is over-featured, some of the features don’t work, and it is nearly 5,000 lines long, which for a computation this trivial is quite excessive.

Instead we are going to study in this section two much smaller examples, which you will find on:

```
/N/hpc/mpich2/src/mpich2-0.94b1/src/mpe/contrib/test/cxgraphics.c
```

and

```
/N/hpc/mpich2/src/mpich2-0.94b1/src/mpe/contrib/life/life_g.c
```

**Program cxgraphics.c**

This program is very, very short. Here is the code.

```

#include <stdio.h>
#include <stdlib.h>
#include "mpe.h"
#include "mpe_graphics.h"

int main( int argc, char** argv )
{
    MPE_XGraph graph;
    int ierr, mp_size, my_rank;
    MPE_Color my_color;
    char ckey;

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &mp_size );
    MPI_Comm_rank( MPI_COMM_WORLD, &my_rank );

    ierr = MPE_Open_graphics( &graph, MPI_COMM_WORLD, NULL,
                             -1, -1, 400, 400, 0 );
    if ( ierr != MPE_SUCCESS ) {
        fprintf( stderr, "%d : MPE_Open_graphics() fails\n", my_rank );
        ierr = MPI_Abort( MPI_COMM_WORLD, 1 );
    }
    my_color = (MPE_Color) (my_rank + 1);
    if ( my_rank == 0 )
        ierr = MPE_Draw_string( graph, 187, 205, MPE_BLUE, "Hello" );
    ierr = MPE_Draw_circle( graph, 200, 200, 20+my_rank*5, my_color );
    ierr = MPE_Update( graph );

    if ( my_rank == 0 ) {
        fprintf( stdout, "Hit any key then return to continue " );
        fscanf( stdin, "%s", &ckey );
        fprintf( stdout, "\n" );
    }
    MPI_Barrier( MPI_COMM_WORLD );

    ierr = MPE_Close_graphics( &graph );

    MPI_Finalize();

    return 0;
}

```

Here is how to link and compile this program:

```

gustav@bh1 $ mpicc -o cxgraphics cxgraphics.c -lmpe -L/usr/X11R6/lib -lX11 -lm
gustav@bh1 $

```

In order to run it you must undertake the following steps:

1. If your variable DISPLAY points to an ssh socket, which is going to be the case if you connected to the AVIDD cluster using `slogin -X`, redefine DISPLAY to point to your physical display. Use either the IP number or the fully domainized name of the machine from which you have made

the connection. Note that the laboratory PCs don't have fully domainized names, so in this case you need to use their IP numbers. To find what they are, use the `netstat -rn` command in your local Cygwin environment.

2. On your display machine, e.g., on your PC, issue the command `xhost +`. This command is a little dangerous, because it opens your X11 display to read/write operations from any other system. The alternative would be to list the names of all AVIDD nodes, and then for each of them issue the command more specific command, e.g., `xhost +bc64.uits.indiana.edu`.
3. Now only you should start the MPD, i.e., issue the command `mpdboot`. If you had MPD running before, you should shut it down with `mpdallexit` and then restart it.
4. Finally, with MPD running you can execute the program by simply saying something like

```
$ mpiexec -n 16 cxgraphics
```

Now let us have a closer look at the program and see how the MPE graphics routines are used in it.

After the usual initializations:

```
MPI_Init( &argc, &argv );
MPI_Comm_size( MPI_COMM_WORLD, &mp_size );
MPI_Comm_rank( MPI_COMM_WORLD, &my_rank );
```

the program opens the X11 window by calling

```
ierr = MPE_Open_graphics( &graph, MPI_COMM_WORLD, NULL,
                        -1, -1, 400, 400, 0 );
```

- The first argument, `&graph` is the pointer to an opaque MPE variable `MPE_XGraph`. Once the window has been successfully opened, other drawing routines will refer to this `graph`.
- The second argument is the communicator.
- The third argument is the name of the display, for example it could be `"beige.ucs.indiana.edu:0.0"`. If instead the name is set to `NULL`, every process is going to get the name from its MPD, which, in turn, gets it from the environment on the console node at the time of MPD booting. You should now understand, why you cannot use the `ssh` socket, such as, e.g., `localhost:27.0`. This socket exists on the console node only, but not on any other node. Consequently processes running on other nodes would not be able to access it and function `MPE_Open_graphics` would return an error.
- The next four arguments specify the positions of the upper left corner and the bottom right corner in pixels. Here `-1` says *go as far to the left, or upwards, as you can*. So the window is going to be created in the upper left corner and it is going to be 400 pixels wide and 400 pixels high.

- Function `MPE_Open_graphics` may be used either in the collective or in the non-collective mode. Whether it is used in the collective mode is specified by the last argument, which is a `BOOLEAN` variable. If it is `FALSE`, the call is non-collective, otherwise it is collective. Here it is non-collective, because the last argument is `0 = FALSE`.

Because the mode is non-collective various processes may get different returns from the function call. We test the returns in a way that is very reminiscent of the way other MPI errors have been handled:

```
if ( ierr != MPE_SUCCESS ) {
    fprintf( stderr, "%d : MPE_Open_graphics() fails\n", my_rank );
    ierr = MPI_Abort( MPI_COMM_WORLD, 1 );
}
```

A process that has failed to access the X11 display and open its portion of the X11 window writes an error message on standard error and aborts, which is going to take the whole MPI program down.

Assuming that this has not happened, the processes define their color:

```
my_color = (MPE_Color) (my_rank + 1);
```

and process of rank zero writes the word "Hello" on the window:

```
if ( my_rank == 0 )
    ierr = MPE_Draw_string( graph, 187, 205, MPE_BLUE, "Hello" );
```

- Function `MPE_Draw_string` takes `graph` (as returned by the call to `MPE_Open_graphics`) as its first argument.
- The second and third arguments are the  $x$  and  $y$  coordinates in pixels, measured from the upper left corner of the window (this is an X11 tradition), of the beginning of the string to be drawn.
- The fourth argument is the colour in which to draw the string, and
- the last argument is the string itself.

Now each process draws a circle in its own color with the centre at (200,200). This is accomplished by calling

```
ierr = MPE_Draw_circle( graph, 200, 200, 20+my_rank*5, my_color );
```

- Function takes `graph` as its first argument.
- Its second and third arguments are the coordinates of the centre of the circle measured in pixels from the upper left corner of the window.
- Its fourth argument is the radius of the circle measured in pixels, and
- its last argument is the colour in which the circle is to be drawn.

The next call to

```
ierr = MPE_Update( graph );
```



simply ensures that all that's been drawn on the `graph` gets sent to the X11 window that's associated with it.

At this stage the program waits for the user to type something on the console's keyboard. Otherwise the picture would vanish almost instantaneously and we might not have the time enough to admire it:

```

    if ( my_rank == 0 ) {
        fprintf( stdout, "Hit any key then return to continue " );
        fscanf( stdin, "%s", &key );
        fprintf( stdout, "\n" );
    }
    MPI_Barrier( MPI_COMM_WORLD );

```

Observe that it is process zero that is in charge of the console – this is always the case in `MPICH2` – the other processes wait for it on the barrier. The barrier must be there, because what follows it is

```

    ierr = MPE_Close_graphics( &graph );

```

If other processes did not have to wait on the barrier, they would take their portions of the graph down, but, as it happens, they all get to it only after the user has typed something on the keyboard.

The program ends with the usual:

```

    MPI_Finalize();
    return 0;
}

```

Now let me introduce officially the synopsis for each of the graphic functions discussed in this section, as well as some other ones to be used in the next section:

```

int MPE_Open_graphics(MPE_Xgraph *handle, MPI_Comm comm, char *display,
                     int x, int y, int is_collective);
int MPE_Draw_point(MPE_Xgraph handle, int x, int y, MPE_Color color);
int MPE_Draw_line(MPE_Xgraph handle, int x1, int y1, int x2, int y2,
                 MPE_Color color);
int MPE_Draw_circle(MPE_Xgraph handle, int centerx, int centery, int radius,
                  MPE_Color color);
int MPE_Update(MPE_XGraph handle);
int MPE_Num_colors(MPE_Xgraph handle, int *number_of_colors);
int MPE_Make_color_array(MPE_XGraph handle, int number_of_colors,
                       MPE_Color array[]);
int MPE_Close_graphics(MPE_XGraph handle);
int MPE_Draw_string(MPE_XGraph handle, int x, int y, MPE_Color, char* string);

```

The predefined colors can be found on

```

/N/hpc/mpich2/include/mpe_graphics.c

```

In our case they are

```

MPE_WHITE, MPE_BLACK, MPE_RED, MPE_YELLOW,
MPE_GREEN, MPE_CYAN, MPE_BLUE, MPE_MAGENTA,
MPE_AQUAMARINE, MPE_FORESTGREEN, MPE_ORANGE,
MPE_MAROON, MPE_BROWN, MPE_PINK, MPE_CORAL,
MPE_GRAY

```

Observe that all processes share the `graph`. Each process can access any pixel on the `graph`. It is up to the programmer to ensure that they don't step on each other's toes, unless, of course, this is what the programmer wants.

**Program life\_g.c**

This program is somewhat more complicated than the previous example. But it is still quite simple in its use of graphics. There is one new element here that is related to our use of `MPI_Sendrecv` in section 5.2.5.

The computation itself is quite similar to Jacobi iterations. The difference is that here the only values allowed in the matrix are zero and one. The iteration takes a sum over all neighbours of an  $(i, j)$  element of the matrix, *including* the diagonal neighbours too (here there is the first difference), so we have 8 neighbours this time instead of just 4 as was the case in the diffusion problem, and now

- if the sum is 3 then the value of the  $(i, j)$  element is set to 1,
- if the sum is 2 then the value of the  $(i, j)$  element is left as it was before,
- otherwise the value of the  $(i, j)$  element is set to 0.

The result is a peculiar pattern that keeps evolving within the matrix. The program calculates the evolution and displays the pattern as it evolves.

Here is the code.

```
#include <mpi.h>
#define MPE_GRAPHICS
#include "mpe.h"
#include <stdio.h>

static MPE_XGraph graph;

static char *displayname = 0;
extern void srand48();
extern double drand48();
extern char * malloc();
static int width = 400, height = 400;

#define BORN 1
#define DIES 0

/* The Life function */
double life(matrix_size, ntimes, comm)
int matrix_size;
int ntimes;
MPI_Comm comm;
{
    int rank, size ;
    int next, prev ;
    int i, j, k;
    int mysize, sum ;
    int **matrix, **temp, **addr ;
    double slavetime, totaltime, starttime ;
    int my_offset;

    /* Determine size and my rank in communicator */
    MPI_Comm_size(comm, &size) ;
    MPI_Comm_rank(comm, &rank) ;
```

```

/* Set neighbors */
if (rank == 0)
    prev = MPI_PROC_NULL;
else
    prev = rank-1;
if (rank == size - 1)
    next = MPI_PROC_NULL;
else
    next = rank+1;

/* Determine my part of the matrix */
mysize = matrix_size/size + ((rank < (matrix_size % size)) ? 1 : 0 ) ;
my_offset = rank * (matrix_size/size);
if (rank > (matrix_size % size)) my_offset += (matrix_size % size);
else
    my_offset += rank;

/* allocate the memory dynamically for the matrix */
matrix = (int **)malloc(sizeof(int *)*(mysize+2)) ;
temp = (int **)malloc(sizeof(int *)*(mysize+2)) ;
for (i = 0; i < mysize+2; i++) {
    matrix[i] = (int *)malloc(sizeof(int)*(matrix_size+2)) ;
    temp[i] = (int *)malloc(sizeof(int)*(matrix_size+2)) ;
}

/* Initialize the boundaries of the life matrix */
for (j = 0; j < matrix_size+2; j++)
    matrix[0][j] = matrix[mysize+1][j] = temp[0][j] = temp[mysize+1][j] = DIES ;
for (i = 0; i < mysize+2; i++)
    matrix[i][0] = matrix[i][matrix_size+1] = temp[i][0] = temp[i][matrix_size+1] = DIES ;

/* Initialize the life matrix */
for (i = 1; i <= mysize; i++) {
    srand48((long)(1000^(i-1+mysize))) ;
    for (j = 1; j <= matrix_size; j++)
        if (drand48() > 0.5)
            matrix[i][j] = BORN ;
        else
            matrix[i][j] = DIES ;
}

/* Open the graphics display */
MPE_Open_graphics( &graph, MPI_COMM_WORLD, displayname,
    -1, -1, width, height, 0 );

/* Play the game of life for given number of iterations */
starttime = MPI_Wtime() ;
for (k = 0; k < ntimes; k++) {
    MPI_Request      req[4];
    MPI_Status       status[4];

    /* Send and receive boundary information */
    MPI_Isend(&matrix[1][0],matrix_size+2,MPI_INT,prev,0,comm,req);
    MPI_Irecv(&matrix[0][0],matrix_size+2,MPI_INT,prev,0,comm,req+1);
    MPI_Isend(&matrix[mysize][0],matrix_size+2,MPI_INT,next,0,comm,req+2);
    MPI_Irecv(&matrix[mysize+1][0],matrix_size+2,MPI_INT,next,0,comm,req+3);

```

```

MPI_Waitall(4, req, status);

/* For each element of the matrix ... */
for (i = 1; i <= mysize; i++) {
    for (j = 1; j < matrix_size+1; j++) {

        /* find out the value of the current cell */
        sum = matrix[i-1][j-1] + matrix[i-1][j] + matrix[i-1][j+1]
            + matrix[i][j-1] + matrix[i][j+1]
            + matrix[i+1][j-1] + matrix[i+1][j] + matrix[i+1][j+1] ;

        /* check if the cell dies or life is born */
        if (sum < 2 || sum > 3)
            temp[i][j] = DIES ;
        else if (sum == 3)
            temp[i][j] = BORN ;
        else
            temp[i][j] = matrix[i][j] ;
    }
}
int xloc, yloc, xwid, ywid;
xloc = ((my_offset + i - 1) * width) / matrix_size;
yloc = ((j - 1) * height) / matrix_size;
xwid = ((my_offset + i) * width) / matrix_size - xloc;
ywid = (j * height) / matrix_size - yloc;
MPE_Fill_rectangle( graph, xloc, yloc, xwid, ywid, temp[i][j] );
}
}
MPE_Update( graph );
/* Swap the matrices */
addr = matrix ;
matrix = temp ;
temp = addr ;
}

/* Return the average time taken/processor */
slavetime = MPI_Wtime() - starttime;
MPI_Reduce (&slavetime, &totaltime, 1, MPI_DOUBLE, MPI_SUM, 0, comm);
return (totaltime/(double)size);}

int main(argc, argv)
int argc ;
char *argv[] ;
{
    int rank, N, iters ;
    double time ;

    MPI_Init (&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank) ;

    /* If I'm process 0, determine the matrix size and # of iterations */
    /* This relies on the MPI implementation properly flushing output
       that does not end in a newline. MPI does not require this, though
       high-quality implementations will do this.
    */
    #if !defined (SGI_MPI) && !defined (IBM_MPI)
    if ( rank == 0 ) {

```

```

    printf("Matrix Size : ") ;
    scanf("%d",&N) ;
    printf("Iterations : ") ;
    scanf("%d",&iters) ;
}
#else
N=20;
iters=50;
#endif

/* Broadcast the size and # of iterations to all processes */
MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD) ;
MPI_Bcast(&iters, 1, MPI_INT, 0, MPI_COMM_WORLD) ;

if (argc > 2 && strcmp( argv[1], "-display" ) == 0) {
    displayname = malloc( strlen( argv[2] ) + 1 );
    strcpy( displayname, argv[2] );
}

/* Call the life routine */
time = life ( N, iters, MPI_COMM_WORLD );

/* Print the total time taken */
if (rank == 0)
    printf("[%d] Life finished in %lf seconds\n",rank,time/100);

MPE_Close_graphics(&graph);
MPI_Finalize();
}

```

The code comprises two distinct parts. There is the short `main` program that is quite easy to decipher and a little more complicated function `life` that does all the work.

So let us begin with `main`.

After the initial MPI incantations, process number 0 obtains some information from standard input, namely, the size of the matrix and the desired number of iterations:

```

if ( rank == 0 ) {
    printf("Matrix Size : ") ;
    scanf("%d",&N) ;
    printf("Iterations : ") ;
    scanf("%d",&iters) ;
}

```

This information is then broadcast to other processes:

```

MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD) ;
MPI_Bcast(&iters, 1, MPI_INT, 0, MPI_COMM_WORLD) ;

```

Every process also analyses the command line (every process gets a copy of `argv` and `argc` from `MPI_Init`) to find if the user has specified the `-display` option, in which case the name of the X11 display is also read from the command line:

```

if (argc > 2 && strcmp( argv[1], "-display" ) == 0) {
    displayname = malloc( strlen( argv[2] ) + 1 );
    strcpy( displayname, argv[2] );
}

```

Otherwise the name of the display will be obtained from MPDs, i.e., from the environmental variable `DISPLAY` at the time MPD was booted on the console node. Here `displayname` is a global (static) variable, so it doesn't have to be passed to function `life`. The display itself is going to be opened and then written to by `life`. Function `life` is called as follows:

```
time = life ( N, iters, MPI_COMM_WORLD );
```

After the program returns, graphics are closed and MPI itself shuts down:

```
MPE_Close_graphics(&graph);
MPI_Finalize();
}
```

And this is the end of `main`.

Function `life` begins its life from finding about the size of the communicator that's been passed to it as a variable. Then every process finds about its rank within this communicator:

```
MPI_Comm_size(comm, &size) ;
MPI_Comm_rank(comm, &rank) ;
```

Now we can see some interesting manipulations, whose purpose is very similar to what we did in the diffusion program. But there we simply used powerful MPI functions, where here these manipulations are carried out explicitly.

First the processes are organized into one dimensional grid, so that each process knows about its `previous` process and its `next` process:

```
/* Set neighbors */
if (rank == 0)
    prev = MPI_PROC_NULL;
else
    prev = rank-1;
if (rank == size - 1)
    next = MPI_PROC_NULL;
else
    next = rank+1;
```

Now the matrix, which is going to be  $N \times N$  is going to be divided amongst the processes (within `life` the parameter `N` is called `matrix_size`):

```
mysize = matrix_size/size + ((rank < (matrix_size % size)) ? 1 : 0) ;
my_offset = rank * (matrix_size/size);
if (rank > (matrix_size % size)) my_offset += (matrix_size % size);
else
    my_offset += rank;
```

Having decided about the portion of the matrix each process is going to look after, we allocate memory for it:

```
/* allocate the memory dynamically for the matrix */
matrix = (int **)malloc(sizeof(int *)*(mysize+2)) ;
temp = (int **)malloc(sizeof(int *)*(mysize+2)) ;
for (i = 0; i < mysize+2; i++) {
    matrix[i] = (int *)malloc(sizeof(int)*(matrix_size+2)) ;
    temp[i] = (int *)malloc(sizeof(int)*(matrix_size+2)) ;
}
```

Now we have the same problem with the "ghost columns" (the division of the matrix is one dimensional in this program) that we had in the diffusion program and we initialize these regions to zero:

```

/* Initialize the boundaries of the life matrix */
for (j = 0; j < matrix_size+2; j++)
    matrix[0][j] = matrix[mysize+1][j] = temp[0][j] = temp[mysize+1][j] = DIES ;
for (i = 0; i < mysize+2; i++)
    matrix[i][0] = matrix[i][matrix_size+1] = temp[i][0] = temp[i][matrix_size+1]
] = DIES ;

```

The interiors of the matrices that belong to the processes, on the other hand, are initialized to ones and zeros at random:

```

/* Initialize the life matrix */
for (i = 1; i <= mysize; i++) {
    srand48((long)(1000^(i-1+mysize))) ;
    for (j = 1; j<= matrix_size; j++)
        if (drand48() > 0.5)
            matrix[i][j] = BORN ;
        else
            matrix[i][j] = DIES ;
}

```

Now function `life` opens the graph on the X11 display:

```

/* Open the graphics display */
MPE_Open_graphics( &graph, MPI_COMM_WORLD, displayname,
                  -1, -1, width, height, 0 );

```

where `width` and `height` are constants, equal 400 each. The window is going to be displayed in the top left corner of the X11 display. The open is not collective. Because the size of the X11 window is fixed in this program, the matrix, which may be either smaller or larger than  $400 \times 400$  will have to be either squeezed or stretched in order to be displayed.

Now we begin the iterations. Each iteration begins with the exchange of border columns:

```

/* Send and receive boundary information */
MPI_Isend(&matrix[1][0],matrix_size+2,MPI_INT,prev,0,comm,req);
MPI_Irecv(&matrix[0][0],matrix_size+2,MPI_INT,prev,0,comm,req+1);
MPI_Isend(&matrix[mysize][0],matrix_size+2,MPI_INT,next,0,comm,req+2);
MPI_Irecv(&matrix[mysize+1][0],matrix_size+2,MPI_INT,next,0,comm,req+3);
MPI_Waitall(4, req, status);

```

This is done differently from the example discussed in section about diffusion. I have mentioned there that `MPI_Sendrecv` saves us from serialization of the communication. Another way to avoid such serialization is to use non-blocking sends and receives. Observe that we use a new function here, `MPI_Waitall`, to do the waiting. This function waits not for just one request to be completed. Instead it waits for the whole array of requests to be completed. In this case the array, `req`, comprises 4 individual requests. The variable `status` points to the array of statuses rather than to an individual status.

Having exchanged the border information we can now commence the computation itself, which is as I have already explained at the beginning of this section:

```

/* For each element of the matrix ... */
for (i = 1; i <= mysize; i++) {
    for (j = 1; j < matrix_size+1; j++) {

```

```

/* find out the value of the current cell */
sum = matrix[i-1][j-1] + matrix[i-1][j] + matrix[i-1][j+1]
    + matrix[i][j-1] + matrix[i][j+1]
    + matrix[i+1][j-1] + matrix[i+1][j] + matrix[i+1][j+1] ;

/* check if the cell dies or life is born */
if (sum < 2 || sum > 3)
    temp[i][j] = DIES ;
else if (sum == 3)
    temp[i][j] = BORN ;
else
    temp[i][j] = matrix[i][j] ;
{
    [ ... draw the graph ... ]
}
}
}

```

For each element of the matrix we evaluate the sum over all its *eight* neighbours and then set the value of the element itself to 1, 0 or leave it unchanged, depending on what the sum is.

Now we have to display the matrix by either squeezing it or stretching, as required, and then throwing the result on the display:

```

{ int xloc, yloc, xwid, ywid;
  xloc = ((my_offset + i - 1) * width) / matrix_size;
  yloc = ((j - 1) * height) / matrix_size;
  xwid = ((my_offset + i) * width) / matrix_size - xloc;
  ywid = (j * height) / matrix_size - yloc;
  MPE_Fill_rectangle( graph, xloc, yloc, xwid, ywid, temp[i][j] );
}

```

The drawing itself is done by filling the rectangle at an appropriate position with the blot, whose color is given by the value of the corresponding matrix element, i.e., with either zero or one, which is going to translate into white and black.

Having finished with the update, the current picture is flushed onto the display by calling

```
MPE_Update( graph );
```

and the matrix itself is being updated too:

```

/* Swap the matrices */
addr = matrix ;
matrix = temp ;
temp = addr ;

```

An important point here is that the updates must not be carried out on the actual matrix (why?). This is why we have two matrices here. The original one and then the updated one. At the end of each iteration we swap them around, i.e., the updated one becomes the original one (for the next iteration) and the original one becomes the temporary space on which the update is going to be



calculated. This is done without any data copying. Instead we merely reassign the pointers.

Having finished with all the requested iterations, function `life` estimates time spent on the whole operation and returns it to the calling program:

```
/* Return the average time taken/processor */
slavetime = MPI_Wtime() - starttime;
MPI_Reduce (&slavetime, &totaltime, 1, MPI_DOUBLE, MPI_SUM, 0, comm);
return (totaltime/(double)size);}
```

### 5.4.3 Exercises

- 1 Rewrite program `life_g` to make use of MPI functions discussed in section 5.2.5 that talked about the diffusion program.
- 2 Rewrite program (1) so that the whole matrix is laid out onto a 2-dimensional grid of processes.
- 3 Rewrite program (2) to dump the whole matrix onto an MPI file at the end of its execution.
- 4 Rewrite program (3) to read the initial data from an MPI file if such a request is made on the command line. Otherwise the program should initialize the matrix as before.



## Chapter 6

# The Assignment

The starting point for the assignment is the program you have developed in exercise 4 in section 5.4.3. But this time the computation is going to be a little different.

1. Assume both `matrix` and `temp` to be `real`, not integer.
2. Let every term of `temp` be constructed simply as follows:

```
temp[i][j] = 0.25 * (matrix[i-1][j] + matrix[i+1][j] +  
                    matrix[i][j-1] + matrix[i][j+1]);
```

3. Assume that the boundary conditions for the whole region, are fixed, i.e., they do not change throughout the computation. You will have to make sure that you don't overwrite them accidentally. Assume 500.0 on the left boundary and 0.0 on the remaining three boundaries of the region.
4. The colours provided by MPE vary between `MPE_WHITE = 0` and `MPE_GRAY = 15`. Display the values of `matrix` in such a way that 500.0 corresponds to 15 and 0.0 corresponds to 0. Ensure that you don't stray beyond 15 or beyond 0 and that the colour values are indeed integers, otherwise MPE will throw an error.
5. Add timing to the program, as discussed in section 4.4. Measure the time used for computation and for generation of the data file. Develop the logic of the program so that it takes time allocated to the run from the environment or from the command line and that it dumps the checkpoint file and the log file before its time runs out. The program should log whether the computation has been completed or if it needs to be continued. The computation will be completed when the largest difference between `temp[i][j]` and `matrix[i][j]` for the *whole* computational region is less than  $\epsilon$ , the value of which should be obtained from the command line or from the environment.

6. Develop a PBS script that is going to submit and resubmit the run until the computation has been completed.
7. Test all functions developed above carefully. Provide a `man` page for the program describing all its features, possible bugs, command line arguments, etc. Provide a `README` file that describes in detail the compilation and installation of the program.
8. The program itself should be meticulously commented so that its reader can understand your every step.

The assignment should be completed no later than by the 19<sup>th</sup> of December.

**Method of Delivery** The program, its `man` page, the corresponding PBS scripts, and a `README` file that describes the whole lot, providing explanations about how to compile, install and run the program – interactively and under PBS – should be placed in a selected directory in your AVIDD `$HOME`. You should then e-mail the location of the directory to me, so that I can go there and check your work.

Once you have laboured enough on this assignment, you may wish to look at how this program can be written in High Performance Fortran. Connect to <http://beige.ucs.indiana.edu/P573/node113.html>. You will see there that the whole computation, which is carried out in parallel, is captured on the mere 6 lines of the following loop:

```
do i = 1, iterations
  where (mask)
    field = (eoshift(field, 1, dim=1) + eoshift(field, -1, dim=1) &
            + eoshift(field, 1, dim=2) + eoshift(field, -1, dim=2)) * 0.25
  end where
end do
```

**Additional Comments** Pointers and arrays are *not* the same in C, even though they are often treated as the same, yet, this usually leads to problems. Furthermore, multidimensional arrays in C are really arrays of arrays. Consider the following code taken from `life_g.c`:

```
int    **matrix, **temp, **addr ;
...
matrix = (int **)malloc(sizeof(int *)*(mysize+2)) ;
temp = (int **)malloc(sizeof(int *)*(mysize+2)) ;
for (i = 0; i < mysize+2; i++) {
  matrix[i] = (int *)malloc(sizeof(int)*(matrix_size+2)) ;
  temp[i] = (int *)malloc(sizeof(int)*(matrix_size+2)) ;
}
```

Observe that for every `matrix[i]` we call a separate `malloc` here. This means that you cannot assume that the data pointed to by `matrix` is going to be laid out in the computer's memory contiguously. Consequently, if you then attempt to describe it to MPI in terms of a stride and some matrix geometry, thinking of `matrix` as an array of rank 2, your description will not match the reality and when MPI attempts to retrieve data based

on this description, it'll get the wrong data from the wrong locations and it will write it on wrong locations too. You may even end up crashing the process, if you attempt the write.

This, however, is not the case for statically declared and defined arrays. If you define:

```
double matrix[500][500];
```

then the compiler will create a contiguous space for the  $500 \times 500$  doubles – then only columns will be columns, and the “distance” between the rows, i.e., the stride, will be the same for every pair of adjacent rows.

Although GNU and other antiquated C compilers may fail on large statically allocated arrays with `segmentation fault`, this should not happen if you use the Intel compiler. The MPI software installed in my `$HOME` is compiled using the Intel compiler.

How then can you create an array of rank 2 or higher dynamically, so that all the data in the array is laid out contiguously – the same way it is going to happen for a statically declared array? The answer is you cannot do this in C. C is not a good language for working with multidimensional arrays. Instead you have to `malloc` a single array of rank 1, i.e., a long vector and then design your own pointer arithmetic for it, so that a pair of indexes, `[i][j]`, will translate into, say, `i * NCOL + j`.



# Chapter 7

## HDF5

### 7.1 Introduction

HDF stands for “Hierarchical Data Format”. It is a library of functions that let you structure your data files. The library provides functions for writing and reading HDF files and there are also stand-alone utilities for viewing them, analyzing them, for converting GIF images to HDF5 images and for compilation and linking of programs that use HDF.

The version of HDF we are going to work with is HDF5. Although conceptually very similar to HDF4, HDF5 is quite different. HDF5 functions have different names, the file format is different, the data model is different, and, most importantly for us, HDF5 binds to MPI-IO so that you can write HDF5 files from MPI programs in parallel.

At the time of this writing HDF5 is not installed on the AVIDD cluster officially yet. I have it currently in my home directory and you will have to put `/N/B/gustav/bin` in front of your command search `PATH` and `/N/B/gustav/lib` in front of your library search path, in order to run HDF5 examples. The reason for this is that there are still some problems with parallel HDF5 files and I want to have these resolved with NCSA folks before I tell AVIDD administrators that HDF5 is ready to install. But we are going to work with sequential HDF5 initially, so these problems should not affect us. And by the time we get to the parallel IO part, they may well be fixed.

HDF5 is a relatively new product. I’ve been told that its parallel component has not been tested on IA32 yet (so we may as well be the first to use it). But conceptually HDF5 goes back to 1998, when HDF4 developers got a grant to develop HDF further in order to address shortcomings of HDF4 and to make HDF work with MPI-IO. Since its early release in 1998, HDF5 has been used in numerous software packages such as Cactus, Chombo, Flash, Swarm, Vis5D,

Intel Array Visualizer, LabVIEW, and some others. HDF5 is also used by some 90 organizations or so, which is a lot for a highly specialized software package of this type.

HDF development was paid for by the Department of Defense, the Department of Energy, NASA, Hughes Information Technology Corporation, Microsoft, NSF, Phillips Petroleum, the State of Illinois, Boeing, NCSA, and other partners. It was developed in Urbana-Champaign by researchers associated with NCSA. The current head of the project is Michael J. Folk, [mfolk@ncsa.uiuc.edu](mailto:mfolk@ncsa.uiuc.edu), but if you have any queries regarding HDF5 you should send mail to *HDF User Support* at NCSA, [hdfhelp@ncsa.uiuc.edu](mailto:hdfhelp@ncsa.uiuc.edu), since HDF problems are now handled by the whole army of programmers. Naturally, within this course, I590, you should probably contact me first, and if I can't answer your question, we will then go to NCSA.

NCSA provides quite nice on-line documentation that covers HDF5. The central point of it is <http://hdf.ncsa.uiuc.edu/HDF5/>, from where you can go to

- the Tutorial, <http://hdf.ncsa.uiuc.edu/HDF5/doc/Tutor/>,
- User's Guide <http://hdf.ncsa.uiuc.edu/HDF5/doc/UG/>,
- Reference Manual [http://hdf.ncsa.uiuc.edu/HDF5/doc/RM\\_H5Front.html](http://hdf.ncsa.uiuc.edu/HDF5/doc/RM_H5Front.html),
- Frequently Asked Questions, <http://hdf.ncsa.uiuc.edu/HDF5-FAQ.html>

and other documents.

## 7.2 Structured versus Flat Files

An HDF5 file can be thought of as a *directory tree within a file*, the leaves of which are datasets or images.

This is a little similar to AFS filesets, which are also implemented as UFS files. Each AFS fileset is a whole lightweight file system, with directories and individual AFS files, but physically it is implemented as a single UFS file. But whereas AFS couples to the kernel, so that you can go to an AFS fileset and use the standard UNIX `ls` command (or Windows `dir`) to view the directory of an AFS fileset, HDF5 is implemented on the application level. Consequently, you have to use HDF5 utilities or HDF5 library calls in order to view what's inside HDF5 files.

HDF5 directories are called *groups*. As is the case with directories, HDF5 groups can contain other groups as well as datasets (in the UNIX world we would think of a dataset as a non-directory file).

If you have ever worked closely with the old Macintosh operating system (pre MacOSX), you may remember that Macintosh files comprised two forks. To each file there was a data fork and an annotation fork. Similarly, each HDF5 dataset may contain data as well as attributes. Attributes are annotations.



They can be used to provide additional information about the data, e.g., units, or when and where the data was collected.

Here is a conceptual picture of a small HDF5 file:

```
HDF5 "dset.h5" {
GROUP "/" {
  DATASET "dset" {
    DATATYPE H5T_STD_I32BE
    DATASPACE SIMPLE { ( 4, 6 ) / ( 4, 6 ) }
    DATA {
      1, 2, 3, 4, 5, 6,
      7, 8, 9, 10, 11, 12,
      13, 14, 15, 16, 17, 18,
      19, 20, 21, 22, 23, 24
    }
    ATTRIBUTE "Units" {
      DATATYPE H5T_STD_I32BE
      DATASPACE SIMPLE { ( 2 ) / ( 2 ) }
      DATA {
        100, 200
      }
    }
  }
}
}
```

The file is called `dsetf.h5`. The top level group in each HDF5 file is called `"/`. This is much the same as the top level directory in UNIX. In this case this group has no sub-groups, i.e., there are no subdirectories here. But we have one dataset, whose full pathname is `"/dset"`, which comprises four items (or “forks”). The first item specifies the type of the data. HDF5 introduces its own typing, much like MPI, so that you can write data in a portable format. If you were to write an HDF5 file on a PC, and then take it to, e.g., a Cray-X1, you should be able to read it with HDF5 utilities or programs without any loss of data. The type of the data written on this dataset is `H5T_STD_I32BE`, which stands for a 32-bit big-endian integer. The second item specifies the size of the data space. It says that the data is written as a simple  $4 \times 6$  matrix, with one data item per one slot. Then we have the data itself, which is the third item, and finally the attribute. The attribute itself is basically a small dataset, structured the same way as the dataset it describes. It has a datatype field, a dataspace field and then the data itself. In this case the data comprises two integers, but more often you would store a string there that would describe the data the attribute is attached to.

Now, you should not think that the data is written on the HDF5 file exactly as shown in the conceptual listing above. If you tried to just view the content of the HDF5 file with `cat` or `type`, all you’d see would be binary junk. You would not get much further with `od` either.

Here is another *conceptual* example of an HDF file:

```
HDF5 "groups.h5" {
GROUP "/" {
  GROUP "MyGroup" {
    GROUP "Group_A" {
```

```

    DATASET "dset2" {
        DATATYPE  H5T_STD_I32BE
        DATASPACE SIMPLE { ( 2, 10 ) / ( 2, 10 ) }
        DATA {
            1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
            1, 2, 3, 4, 5, 6, 7, 8, 9, 10
        }
    }
}
GROUP "Group_B" {
}
DATASET "dset1" {
    DATATYPE  H5T_STD_I32BE
    DATASPACE SIMPLE { ( 3, 3 ) / ( 3, 3 ) }
    DATA {
        1, 2, 3,
        1, 2, 3,
        1, 2, 3
    }
}
}
}
}
}

```

This file comprises the following groups (directories):

- /
- /MyGroup, which contains the dataset /MyGroup/dset1
- /MyGroup/GroupA, which contains the dataset /MyGroup/GroupA/dset2
- /MyGroup/GroupB, which is empty

Because /MyGroup contains /MyGroup/GroupA, which, in turn, contains /MyGroup/GroupA/dset2, you could also say that /MyGroup contains dset2 too.

Now let me show you how you what you are going to see if you view this file with some of the HDF5 tools. The name of the file is `groups.h5` and the name of the tool I am going to use is `h5ls`. It can be used, like UNIX `ls` to view the directory of the HDF5 file, but also, unlike `ls` to view the contents of the datasets.

There is no man entry provided with `h5ls`, but if you invoke it with any of the `-h`, `-?` or `--help` options, you'll get a brief synopsis:

```

gustav@bh1 $ h5ls --help
usage: h5ls [OPTIONS] [OBJECTS...]
OPTIONS
  -h, -?, --help      Print a usage message and exit
  -a, --address       Print addresses for raw data
  -d, --data          Print the values of datasets
  -e, --errors        Show all HDF5 error reporting
  -f, --full          Print full path names instead of base names
  -g, --group         Show information about a group, not its contents
  -l, --label         Label members of compound datasets
  -r, --recursive     List all groups recursively, avoiding cycles
  -s, --string        Print 1-byte integer datasets as ASCII

```

```

-S, --simple      Use a machine-readable output format
-wN, --width=N  Set the number of columns of output
-v, --verbose    Generate more verbose output
-V, --version    Print version number and exit
-x, --hexdump    Show raw data in hexadecimal format

```

#### OBJECTS

Each object consists of an HDF5 file name optionally followed by a slash and an object name within the file (if no object is specified within the file then the contents of the root group are displayed). The file name may include a `printf(3C)` integer format such as `"%05d"` to open a file family.

```
gustav@bh1 $
```

So let us just try `h5ls groups.h5` first:

```

gustav@bh1 $ h5ls groups.h5
MyGroup          Group
gustav@bh1 $

```

Well, the program says that there is one group there, at the top, called `MyGroup`. We can see *all* groups though with the *recursive* listing, much the same as you would invoke `ls -R` in order to a content of the whole directory tree:

```

gustav@bh1 $ h5ls -r groups.h5
/MyGroup          Group
/MyGroup/Group_A  Group
/MyGroup/Group_A/dset2  Dataset {2, 10}
/MyGroup/Group_B  Group
/MyGroup/dset1    Dataset {3, 3}
gustav@bh1 $

```

You can view the content of any selected dataset within the HDF5 file by using the `-d` switch and passing the full name of the dataset to `h5ls` as follows:

```

gustav@bh1 $ h5ls -d groups.h5/MyGroup/Group_A/dset2
MyGroup/Group_A/dset2  Dataset {2, 10}
  Data:
    (0,0) 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
gustav@bh1 $

```

The data is not listed as a matrix, but you are informed about it being a matrix by the formatting statement `Dataset {2, 10}`.

The `-v` switch, which stands for `--verbose`, gives us much more information:

```

gustav@bh1 $ h5ls -v -r groups.h5
Opened "groups.h5" with sec2 driver.
/MyGroup          Group
  Location: 0:1:0:1576
  Links:    1
/MyGroup/Group_A  Group
  Location: 0:1:0:2552
  Links:    1
/MyGroup/Group_A/dset2  Dataset {2/2, 10/10}
  Location: 0:1:0:5896
  Links:    1
  Modified: 2003-11-10 14:21:40 EST
  Storage:  80 logical bytes, 80 allocated bytes, 100.00% utilization
  Type:     32-bit big-endian integer

```

```

/MyGroup/Group_B          Group
  Location: 0:1:0:3528
  Links: 1
/MyGroup/dset1           Dataset {3/3, 3/3}
  Location: 0:1:0:5624
  Links: 1
  Modified: 2003-11-10 14:21:40 EST
  Storage: 36 logical bytes, 36 allocated bytes, 100.00% utilization
  Type: 32-bit big-endian integer
gustav@bh1 $

```

Here we find not only what is the location of each item within the HDF5 file, but even when each of the items was modified and how each dataset utilizes the space that has been allocated to it. In HDF5 you allocate a space for a dataset separately, and the dataset does not have to use all of it. But in this case the datasets fill the space that's been allocated to them entirely.

Program `h5dump` can be used to view the whole file displayed in the same way as I have done above at the beginning of this section:

```

gustav@bh1 $ h5dump dset.h5
HDF5 "dset.h5" {
GROUP "/" {
  DATASET "dset" {
    DATATYPE  H5T_STD_I32BE
    DATASPACE SIMPLE { ( 4, 6 ) / ( 4, 6 ) }
    DATA {
      1, 2, 3, 4, 5, 6,
      7, 8, 9, 10, 11, 12,
      13, 14, 15, 16, 17, 18,
      19, 20, 21, 22, 23, 24
    }
  }
  ATTRIBUTE "Units" {
    DATATYPE  H5T_STD_I32BE
    DATASPACE SIMPLE { ( 2 ) / ( 2 ) }
    DATA {
      100, 200
    }
  }
}
}
gustav@bh1 $

```

This kind of notation, which is quite similar to the way you structure C and C++ programs, is called the Data Description Language, or DDL for short. The language can be formalized, using, e.g., Backus-Naur Form, but I'll stay away from it, because DDL is intuitive enough to be easily understandable without formalizing. But `h5dump` can dump your HDF5 data in other formats too. For example, if you use the `-x` switch, the content of the file will be dumped in XML. XML description though is horribly verbose and far from being as intuitive and clear as DDL.

Instead of dumping the whole HDF5, you can dump only a select object. For example:

```

gustav@bh1 $ h5dump -a /dset/Units dset.h5

```

```

HDF5 "dset.h5" {
ATTRIBUTE "/dset/Units" {
    DATATYPE  H5T_STD_I32BE
    DATASPACE SIMPLE { ( 2 ) / ( 2 ) }
    DATA {
        100, 200
    }
}
}
}
gustav@bh1 $

```

This command lists the selected attribute, in this case `/dset/Units`, associated with the dataset `/dset`. A single data set can have several attributes with different names associated with it.

Calling `h5dump` with the `--help` option, invokes the brief description of the utility with all options listed and explained.

Now, once you know what structured files are, let us go back to the basic question of this section: “Structured versus Flat Files”. Why should we bother about structured files if we can always structure our data using the directory tree itself? After all, instead of writing various datasets on various parts of a structured HDF5 file, we could accomplish much the same by writing them on various files, possibly located in various directories. We could write separate files containing attributes and so on. In other words, we could simply take the whole structure of an HDF5 file out and lay it on top of a file system. Doesn’t an HDF5 file immitate a file system internally? This, after all, is what most scientists do anyway.

The answer is *portability*, portability seen in two ways. First, portability from a researcher to a researcher. If you have your data organized in a directory tree, in order to exchange it with other researchers, you have to send them the whole tree – presumably collated into a single file, e.g., a `tar` archive. But `tar` is an operating system dependent utility. `tar` implies UNIX and when the files get unpacked on, e.g., a Windows system or a VMS system or an MVS system or some other equally exotic system, they may not come out right. The data, if written in a *small-endian* fashion, may get all scrambled if a file is unpacked on a *big-endian* system. The annotations may get all lost. File names may get corrupted. Sometimes even the directory structure may get altered and lost too. Some systems may not allow as deep a directory nesting as other systems. So this brings us right to the issue of portability between operating systems and machine architectures.

HDF5 structured files, by bringing the structure *into* the file itself, release us from the dependence on the file system and on the operating system. By providing us with machine independent formats for data they release us from the dependence on the machine’s architecture and hardware.

By letting us put all the data, annotations, and structuring into a single file, HDF5 helps us manage the data too. If you have a lot of data scattered over a huge directory, it’s quite easy to get things wrong and either corrupt the data by renaming files incorrectly or placing them in the wrong directories, or even lose it by overwriting a file accidentally.

Last, but not least, traversing a directory tree and dealing with a large number of files from within your application, can be expensive.

If you can replace it all with a single file that has all the required structuring, annotations and data inside, your life as a programmer and maintainer may get quite a lot easier. At the end of the day, you may think of HDF5 files as small data bases. They are, sort of, right in between, where a single flat file is too primitive for handling your problem, and where a fully blown data base would be an overkill.

## 7.3 Creating and Structuring HDF5 Files

HDF5 files are handled in a way that is quite similar to directories. If you want to create a file in a directory, you must create the directory first. Similarly, if you want to create a dataset in an HDF5 group, you must create the group first. But before you can create the group, you must create the HDF5 file itself. The latter corresponds to the creation of a file system. When you create an HDF5 file, the top level group, "/", is created automatically too. So this means that an HDF5 file is never empty, there is at least one group in it. And this, again, is much like the story with the file system. A file system is never empty either. There is always, at least, a top level directory in it.

Once you have created an HDF5 file, you can close it and go away. This is the same as with creating a file system. Once you have created it, you don't have to write on it right away. You can go home and write on it tomorrow, in the full knowledge that it won't go away in the meantime.

All example programs in this section are taken from the NCSA Tutorial.

### 7.3.1 HDF5 File Creation

The first program we are going to look at does just this: it creates an HDF5 file, then closes it and goes away. The file is *largely* empty, meaning that there is going to be the top level group in it, but nothing else. We are going to add more structure to it later.

Here is the program:

```

/*
 *   Creating and closing a file.
 */

#include "hdf5.h"
#define FILE "file.h5"

main() {

    hid_t      file_id; /* file identifier */
    herr_t     status;

    /* Create a new file using default properties. */
    file_id = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

```

```

    /* Terminate access to the file. */
    status = H5Fclose(file_id);
}

```

Now I am going to compile and run this program:

```

gustav@bh1 $ pwd
/N/B/gustav/src/I590/HDF5
gustav@bh1 $ which h5cc
/N/B/gustav/bin/h5cc
gustav@bh1 $ h5cc -o h5_crtfile h5_crtfile.c
gustav@bh1 $ ./h5_crtfile
gustav@bh1 $

```

The program executes quietly and creates file `file.h5` in the working directory. Let us have a look at it with `h5dump`:

```

gustav@bh1 $ h5dump file.h5
HDF5 "file.h5" {
GROUP "/" {
}
}
gustav@bh1 $

```

Well, there is the top level group in it and nothing more.

Function `H5Fcreate` creates an HDF5 file. It belongs to the class of *File API Functions*. All functions in this class have names that begin with `H5F`. You will learn about other classes soon enough. The third letter in an HDF5 function name always indicates the class the function belongs to.

Function `H5Fcreate` takes four arguments. The first one is the string that corresponds to the file's name. The second argument is a flag that specifies how the file should be created. There are two flags you can use in this place:

**H5F\_ACC\_TRUNC** which specifies that if you are trying to *create* a file that exists already, the existing file will be truncated, i.e., all data stored on the original file will be erased;

**H5F\_ACC\_EXCL** which specifies that if you are trying to create a file that exists already, the function should return error. The error is returned as a negative integer, otherwise the function returns a file *identifier*, which, technically, is a positive integer.

The third argument is the *file creation property list* and the fourth argument is the *file access property list*. We'll talk about the property lists later and in the meantime we are going to use the default indicator `H5P_DEFAULT`.

The file identifier is an opaque variable of type `hid_t`, which is defined on `H5Ipublic.h` to be a signed integer, so this is why you can use it at the same time for an error indicator. If `H5Fcreate` fails for whatever reason, the value of the identifier returned is going to be negative.

The file creation operation opens the newly created file at the same time. The file is now available for further operations on it.

Having created the file we close it with the call to `H5Fclose`. This call just takes the file identifier, previously returned by the call to `H5Fcreate`, as its sole

argument and returns the status of the operation on `status`, which is a variable of type `herr_t`, defined on `H5public.h` as a signed integer. It is negative in case an error has been detected and positive otherwise.

### 7.3.2 HDF5 Datasets

In this example we are going to do more than just create an empty HDF5 file. We are going to create a file and then we are going to create a dataset within its top level group. But we are still going to postpone filling the dataset with any data. The file system equivalent of this operation is to create an empty file in, say, a top level directory of some file system, and then fill the file with nulls, just so as to reserve sufficient space for the file itself.

Here is the program:

```

/*
 * Creating and closing a dataset.
 */

#include "hdf5.h"
#define FILE "dset.h5"

main() {

    hid_t      file_id, dataset_id, dataspace_id; /* identifiers */
    hsize_t    dims[2];
    herr_t     status;

    /* Create a new file using default properties. */
    file_id = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

    /* Create the data space for the dataset. */
    dims[0] = 4;
    dims[1] = 6;
    dataspace_id = H5Screate_simple(2, dims, NULL);

    /* Create the dataset. */
    dataset_id = H5Dcreate(file_id, "/dset", H5T_STD_I32BE, dataspace_id, H5P_DEFAULT);

    /* End access to the dataset and release resources used by it. */
    status = H5Dclose(dataset_id);

    /* Terminate access to the data space. */
    status = H5Sclose(dataspace_id);

    /* Close the file. */
    status = H5Fclose(file_id);
}

```

And here is how the program is compiled, linked, and run:

```

gustav@bh1 $ h5cc -o h5_crtdat h5_crtdat.c
gustav@bh1 $ ./h5_crtdat
gustav@bh1 $ ls
dset.h5  h5_crtdat    h5_crtdat.o  h5_crtdatfile.c
file.h5  h5_crtdat.c    h5_crtdatfile  h5_crtdatfile.o
gustav@bh1 $

```



The program executes silently, but here you can again see that a new HDF5 file has been created, `dset.h5`. To see what's inside it use `h5dump`:

```
gustav@bh1 $ h5dump dset.h5
HDF5 "dset.h5" {
GROUP "/" {
  DATASET "dset" {
    DATATYPE  H5T_STD_I32BE
    DATASPACE  SIMPLE { ( 4, 6 ) / ( 4, 6 ) }
    DATA {
      0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0
    }
  }
}
}
gustav@bh1 $
```

So this program indeed created a file and then it created an empty fileset within it.

Let us have a look at the program.

After we have created the file itself with the call to `H5Fcreate`:

```
file_id = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
```

we are going to create first the *space* for the dataset. Functions that manipulate *space* that is then going to be allocated to datasets or attributes, all have their names beginning with `H5S`. The third letter, `S`, stands for *space*. The specific function we are going to call here is `H5Screate_simple`

```
dims[0] = 4;
dims[1] = 6;
dataspace_id = H5Screate_simple(2, dims, NULL);
```

It creates the so called *simple* space. When is the data space *simple*? If the data corresponds to a regular  $N$ -dimensional array of data points, e.g., data on a regular rectangular grid, then the corresponding data space is *simple*, otherwise it is called *complex*. So data on irregular grids, resulting from dynamic refinement of meshes, must be written on *complex* data spaces.

Function `H5Screate_simple` takes three arguments. The first argument is the rank of the array. In our case we are going to create space for an array of rank 2, i.e., a 2-dimensional matrix. The second argument is an array of length given by the first argument, whose entries specify the length of the matrix in each dimension. In our case we are going to have a  $4 \times 6$  matrix filled with integers, so the second argument is `[4,6]`.

Now the third argument would be used if we wanted to create a data space that could be stretched if, e.g., we wanted to fill the dataset with more data than originally intended. We can specify here up to how much the space can be stretched, e.g., up to the size of `[400,600]`, or we could place no limits at all on its stretchability, in which case the third argument would be `[H5S_UNLIMITED,H5S_UNLIMITED]`. If the third array is `NULL`, as is the case in this example, the generated data space is not-stretchable.

Arrays used in place of the second and third argument of `H5Screate_simple` are of type `hsize_t *`, where `hsize_t` is defined on `H5public.h` to be `size_t`, which on the AVIDD system defaults to a 32-bit integer.

Function `H5Screate_simple` returns a dataset identifier, which is of the same type as a file identifier and other identifiers returned by other HDF5 function, i.e., `hid_t`.

Having created the space for the dataset, we are now ready to create the dataset itself. The dataset is created within the pre-allocated space by calling function `H5Dcreate`:

```
dataset_id = H5Dcreate(file_id, "/dset", H5T_STD_I32BE, dataspace_id, H5P_DEFAULT);
```

All functions that operate on datasets have their names beginning with `H5D`, where `D` stands for *dataset*. Function `H5Dcreate` takes five arguments:

- The first argument is the file identifier – but here we can also use the group identifier, if we have it.
- The second argument is the name of the dataset to be created - here we use the full pathname of the dataset. If we used the group identifier in the first argument, we could use the relative name of the dataset in the second argument. It would be relative to the group specified by the first argument.
- The third argument is the datatype to use when creating the dataset. HDF5 supports various portable and non-portable, i.e., *native* datatypes here. In this case we use the portable `H5T_STD_I32BE`, 32-bit big-endian integer.
- The fourth argument is the dataspace identifier. In order to create a dataset, we must have the dataspace in which to create it. Now, observe that the dataset itself is created outside of the file. We did not have to use either a file identifier or a group identifier in our call to create the data space. The data space is a device through which we format the data in the dataset itself.
- Finally, the last argument is the *dataset creation property list*. Here we don't specify any such list relying on a default, `H5P_DEFAULT`, instead.

Having created the dataset, we are now going to close it. This is like closing a file in a directory tree:

```
status = H5Dclose(dataset_id);
```

Function `H5Dclose` takes just one argument, the dataset identifier. Then we have to close the data space:

```
status = H5Sclose(dataspace_id);
```

Here the function `H5Sclose` also takes just one argument, namely the data space identifier. And finally we close the file itself:

```
status = H5Fclose(file_id);
```

So this is how it all works. But... we still haven't filled the data set with any data. It's been merely created and sized, but it's filled with zeros. We are going to write real data on it in the next section.

### 7.3.3 Writing On and Reading From HDF5 Datasets

In this example we are going to write data on the dataset created in the previous section. This time we are not going to create the file and the dataset again, because they are already there. Instead we are going to open them. Then we'll write on the dataset and then we'll read the data back and then we'll close the dataset and the file.

Here's the program:

```

/*
 * Writing and reading an existing dataset.
 */

#include "hdf5.h"
#define FILE "dset.h5"

main() {

    hid_t      file_id, dataset_id; /* identifiers */
    herr_t     status;
    int        i, j, dset_data[4][6];

    /* Initialize the dataset. */
    for (i = 0; i < 4; i++)
        for (j = 0; j < 6; j++)
            dset_data[i][j] = i * 6 + j + 1;

    /* Open an existing file. */
    file_id = H5Fopen(FILE, H5F_ACC_RDWR, H5P_DEFAULT);

    /* Open an existing dataset. */
    dataset_id = H5Dopen(file_id, "/dset");

    /* Write the dataset. */
    status = H5Dwrite(dataset_id, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT,
                     dset_data);

    status = H5Dread(dataset_id, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT,
                    dset_data);

    /* Close the dataset. */
    status = H5Dclose(dataset_id);

    /* Close the file. */
    status = H5Fclose(file_id);
}

```

Let me compile the program first:

```

gustav@bh1 $ h5cc -o h5_rdwt h5_rdwt.c
gustav@bh1 $

```

Now I am going to run it, but before I do I'm going to show you what's on the file `dset.h5` and then I'm going to show you again what's in it after the program has completed its execution:

```

gustav@bh1 $ h5dump dset.h5
HDF5 "dset.h5" {

```

```

GROUP "/" {
  DATASET "dset" {
    DATATYPE H5T_STD_I32BE
    DATASPACE SIMPLE { ( 4, 6 ) / ( 4, 6 ) }
    DATA {
      0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0
    }
  }
}
}
}
gustav@bh1 $ ./h5_rdwrt
gustav@bh1 $ h5dump dset.h5
HDF5 "dset.h5" {
GROUP "/" {
  DATASET "dset" {
    DATATYPE H5T_STD_I32BE
    DATASPACE SIMPLE { ( 4, 6 ) / ( 4, 6 ) }
    DATA {
      1, 2, 3, 4, 5, 6,
      7, 8, 9, 10, 11, 12,
      13, 14, 15, 16, 17, 18,
      19, 20, 21, 22, 23, 24
    }
  }
}
}
}
gustav@bh1 $

```

Well, we have indeed written the data on the dataset – without corrupting the rest of the file in the process.

So, how it's done.

The program begins with filling the two dimensional array with data:

```

for (i = 0; i < 4; i++)
  for (j = 0; j < 6; j++)
    dset_data[i][j] = i * 6 + j + 1;

```

Then we open the file by calling function `H5Fopen`:

```
file_id = H5Fopen(FILE, H5F_ACC_RDWR, H5P_DEFAULT);
```

By now you can already tell that this is a file manipulation function, because its name begins with `HDF5`. The function takes three arguments:

- The first argument is the name of the HDF5 file to be opened.
- The second argument is a *file access flag*. There are two flags you can use in this case:

**H5F\_ACC\_RDWR** which says that the file is to be opened for reading and writing;

**H5F\_ACC\_RDONLY** which says that the file is to be opened for reading only.

- The third argument is the *file access property list*, and, as before, we are going to use the default list given by `H5P_DEFAULT`.

So we have opened the file. Now we have to open the dataset:

```
dataset_id = H5Dopen(file_id, "/dset");
```

Function `H5Dopen` opens an *existing* dataset. The first argument is the file identifier and the second argument is the full pathname of the dataset. But you can also use a group identifier in the place of the first argument, if you have it, in which case the second argument would be the name of the fileset in relation to the group. In this case, it would not have the preceding slash.

Once we have opened the dataset, we can write on it. We do this by calling function `H5Dwrite`:

```
status = H5Dwrite(dataset_id, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT,
                 dset_data);
```

which takes the following arguments:

- The first argument is the dataset identifier.
- The second argument informs HDF about the format of data in the memory of the computer. The way that the data is going to be written on the fileset has been specified when the fileset was created. HDF5 libraries will perform required conversions. Normally the format of the data in memory is native, unless you have specifically changed it, which can, of course, be done. So here we say `H5T_NATIVE_INT`.
- The third argument is the identifier of the memory dataspace. Specifying `H5S_ALL` in this slot means that the whole memory dataspace will be used for the IO operation.
- The fourth argument is the file space identifier. Specifying `H5S_ALL` in this slot means that the whole file dataspace will be used for the IO operation.
- The fifth argument is the data transfer property list and we use just a default list given by `H5P_DEFAULT`.
- Finally the last argument is the pointer to the data array itself.

Function `H5Dwrite` returns an integer, which is negative if the operation has failed for some reason.

The next step is to read the data back:

```
status = H5Dread(dataset_id, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT,
                dset_data);
```

The arguments of function `H5Dread` here are exactly the same as for `H5Dwrite`: the operation is entirely symmetric with respect to the latter, which is rather nice. The reason why this can be done this way is because the datasets here are pre-sized. The reading is the same as writing in reverse.

Having finished with the dataset we have to close it first:

```
status = H5Dclose(dataset_id);
```

and then we close the file:

```
status = H5Fclose(file_id);
```

**Exercise**

- 1 Modify the program discussed in the previous section to verify that the data read from the file is indeed the same as the data written on it.

**7.3.4 HDF5 Attributes**

In this section we are going to annotate the dataset with an attribute. Attributes, as remarked earlier, are really small datasets. But they have their own functions for attribute creation, writing, reading and closing. Here is the example program, which illustrates how to use these functions.

```

/*
 *   Creating a dataset attribute.
 */

#include "hdf5.h"
#define FILE "dset.h5"

main() {

    hid_t      file_id, dataset_id, attribute_id, dataspace_id; /* identifiers */
    hsize_t    dims;
    int        attr_data[2];
    herr_t     status;

    /* Initialize the attribute data. */
    attr_data[0] = 100;
    attr_data[1] = 200;

    /* Open an existing file. */
    file_id = H5Fopen(FILE, H5F_ACC_RDWR, H5P_DEFAULT);

    /* Open an existing dataset. */
    dataset_id = H5Dopen(file_id, "/dset");

    /* Create the data space for the attribute. */
    dims = 2;
    dataspace_id = H5Screate_simple(1, &dims, NULL);

    /* Create a dataset attribute. */
    attribute_id = H5Acreate(dataset_id, "Units", H5T_STD_I32BE, dataspace_id, H5P_DEFAULT);

    /* Write the attribute data. */
    status = H5Awrite(attribute_id, H5T_NATIVE_INT, attr_data);

    /* Close the attribute. */
    status = H5Aclose(attribute_id);

    /* Close the dataspace. */
    status = H5Sclose(dataspace_id);

    /* Close to the dataset. */
    status = H5Dclose(dataset_id);

    /* Close the file. */
    status = H5Fclose(file_id);
}

```

```
}

```

The program is compiled with `h5cc`:

```
gustav@bh1 $ h5cc -o h5_crtatt h5_crtatt.c
gustav@bh1 $

```

And now let me show you that it indeed adds an attribute to the dataset:

```
gustav@bh1 $ h5dump dset.h5
HDF5 "dset.h5" {
GROUP "/" {
  DATASET "dset" {
    DATATYPE  H5T_STD_I32BE
    DATASPACE SIMPLE { ( 4, 6 ) / ( 4, 6 ) }
    DATA {
      1, 2, 3, 4, 5, 6,
      7, 8, 9, 10, 11, 12,
      13, 14, 15, 16, 17, 18,
      19, 20, 21, 22, 23, 24
    }
  }
}
}
gustav@bh1 $ ./h5_crtatt
gustav@bh1 $ h5dump dset.h5
HDF5 "dset.h5" {
GROUP "/" {
  DATASET "dset" {
    DATATYPE  H5T_STD_I32BE
    DATASPACE SIMPLE { ( 4, 6 ) / ( 4, 6 ) }
    DATA {
      1, 2, 3, 4, 5, 6,
      7, 8, 9, 10, 11, 12,
      13, 14, 15, 16, 17, 18,
      19, 20, 21, 22, 23, 24
    }
  }
  ATTRIBUTE "Units" {
    DATATYPE  H5T_STD_I32BE
    DATASPACE SIMPLE { ( 2 ) / ( 2 ) }
    DATA {
      100, 200
    }
  }
}
}
}
gustav@bh1 $

```

So, it works then.

The program begins with the initialization of the attribute data array:

```
attr_data[0] = 100;
attr_data[1] = 200;

```

Then we open the file and the dataset as before. Next we create the dataspace for the attribute, and this works exactly the same way as for the datasets:

```
dims = 2;
dataspace_id = H5Screate_simple(1, &dims, NULL);

```

and finally we create the attribute itself:

```
attribute_id = H5Acreate(dataset_id, "Units", H5T_STD_I32BE, dataspace_id, H5P_DEFAULT);
```

Function `H5Acreate` belongs to the family of functions for manipulations of attributes. Their names begin with `H5A`, where `A` stands for *attribute*. The function takes the following arguments:

- The first argument is the identifier of the dataset the attribute is going to be associated with. This argument can be also a group identifier, in which case the attribute will be associated with the group.
- The second argument is the name of the attribute.
- The third argument is the type of data in the attribute. Here we are going to write two integers in the 32-bit big-endian format.
- The fourth argument is the dataspace identifier.
- And the last argument is the *attribute creation property list*, which we are going to replace with the default `H5P_DEFAULT`.

Observe that these parameters are exactly the same as the parameters used by `H5Dcreate`, but you must not use the latter function here. Attributes are attributes and datasets are datasets. HDF5 is fussy about this.

Once we have created the attribute we are going to fill it with data. This we do by calling function `H5Awrite`, which is markedly simpler than `H5Dwrite`, this is where the difference between datasets and attributes becomes more apparent:

```
status = H5Awrite(attribute_id, H5T_NATIVE_INT, attr_data);
```

This function takes just three arguments: the attribute identifier, the memory data type and the data buffer itself. The memory data type is, of course, native, and so HDF5 functions will have to perform the translation to the 32-bit big-endian requested when the attribute was created.

Now we have to close the attribute by calling `H5Aclose` – it takes just one argument, namely the attribute identifier, and then we close all other things we've opened before, i.e., the dataspace, the dataset and the datafile itself:

```
/* Close the attribute. */
status = H5Aclose(attribute_id);

/* Close the dataspace. */
status = H5Sclose(dataspace_id);

/* Close to the dataset. */
status = H5Dclose(dataset_id);

/* Close the file. */
status = H5Fclose(file_id);
```



### 7.3.5 HDF5 Groups

In this section I am going to show you how to create additional HDF5 groups. We are going to create groups:

```
/MyGroup
/MyGroup/Group_A
/MyGroup/Group_B
```

We will use the absolute pathname of the group in one case and a relative pathname of the group in another.

Here is the code that does this. We are going to create a new file, `groups.h5`, this time:

```
/*
 *   Creating groups using absolute and relative names.
 */

#include "hdf5.h"
#define FILE "groups.h5"

main() {

    hid_t      file_id, group1_id, group2_id, group3_id; /* identifiers */
    herr_t      status;

    /* Create a new file using default properties. */
    file_id = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

    /* Create group "MyGroup" in the root group using absolute name. */
    group1_id = H5Gcreate(file_id, "/MyGroup", 0);

    /* Create group "Group_A" in group "MyGroup" using absolute name. */
    group2_id = H5Gcreate(file_id, "/MyGroup/Group_A", 0);

    /* Create group "Group_B" in group "MyGroup" using relative name. */
    group3_id = H5Gcreate(group1_id, "Group_B", 0);

    /* Close groups. */
    status = H5Gclose(group1_id);
    status = H5Gclose(group2_id);
    status = H5Gclose(group3_id);

    /* Close the file. */
    status = H5Fclose(file_id);
}
}
```

Here is how the binary is made and then how the program is run and then what comes out of it:

```
gustav@bh1 $ h5cc -o h5_crtgrpar h5_crtgrpar.c
gustav@bh1 $ ./h5_crtgrpar
gustav@bh1 $ h5dump groups.h5
HDF5 "groups.h5" {
GROUP "/" {
  GROUP "MyGroup" {
    GROUP "Group_A" {
    }
  }
}
```

```

        GROUP "Group_B" {
        }
    }
}
}
gustav@bh1 $

```

Creation of groups is easier than creation of datasets, because you don't have to bother with creating the dataspace. Function `H5Gcreate` belongs to the family of functions for manipulation of HDF5 groups. Functions in this family have names beginning with `H5G`.

Function `H5Gcreate` takes the following arguments:

- The first argument is the file or the group identifier.
- The second argument is the name of the new group. This name may be absolute or relative. If you have used the group identifier for the first argument, then you may define the name of the new group with respect to the group within which it is going to be created. The code does it in two ways. Compare:

```

group1_id = H5Gcreate(file_id, "/MyGroup", 0);
group2_id = H5Gcreate(file_id, "/MyGroup/Group_A", 0);

```

and

```

group3_id = H5Gcreate(group1_id, "Group_B", 0);

```

After the groups have been created, they have to be closed by calling `H5Gclose`. This function takes the group identifier as the sole argument.

### 7.3.6 HDF5 Groups and Datasets

The example program in this section is a little longer, but it doesn't really introduce anything new, which is good. It is a summary example. We are going to add some datasets to the file `groups.h5` created in the previous section. The dataset are going to be created within various groups too.

Here is the program:

```

/*
 * Create two datasets within groups.
 */

#include "hdf5.h"
#define FILE "groups.h5"

main() {

    hid_t      file_id, group_id, dataset_id, dataspace_id; /* identifiers */
    hsize_t    dims[2];
    herr_t     status;
    int        i, j, dset1_data[3][3], dset2_data[2][10];

    /* Initialize the first dataset. */

```

```
for (i = 0; i < 3; i++)
    for (j = 0; j < 3; j++)
        dset1_data[i][j] = j + 1;

/* Initialize the second dataset. */
for (i = 0; i < 2; i++)
    for (j = 0; j < 10; j++)
        dset2_data[i][j] = j + 1;

/* Open an existing file. */
file_id = H5Fopen(FILE, H5F_ACC_RDWR, H5P_DEFAULT);

/* Create the data space for the first dataset. */
dims[0] = 3;
dims[1] = 3;
dataspace_id = H5Screate_simple(2, dims, NULL);

/* Create a dataset in group "MyGroup". */
dataset_id = H5Dcreate(file_id, "/MyGroup/dset1", H5T_STD_I32BE, dataspace_id,
                      H5P_DEFAULT);

/* Write the first dataset. */
status = H5Dwrite(dataset_id, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT,
                  dset1_data);

/* Close the data space for the first dataset. */
status = H5Sclose(dataspace_id);

/* Close the first dataset. */
status = H5Dclose(dataset_id);

/* Open an existing group of the specified file. */
group_id = H5Gopen(file_id, "/MyGroup/Group_A");

/* Create the data space for the second dataset. */
dims[0] = 2;
dims[1] = 10;
dataspace_id = H5Screate_simple(2, dims, NULL);

/* Create the second dataset in group "Group_A". */
dataset_id = H5Dcreate(group_id, "dset2", H5T_STD_I32BE, dataspace_id, H5P_DEFAULT);

/* Write the second dataset. */
status = H5Dwrite(dataset_id, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT,
                  dset2_data);

/* Close the data space for the second dataset. */
status = H5Sclose(dataspace_id);

/* Close the second dataset */
status = H5Dclose(dataset_id);

/* Close the group. */
status = H5Gclose(group_id);

/* Close the file. */
status = H5Fclose(file_id);
```

```
}

```

Now let me compile and then run the program. I will show what the file looks like before and then after the program has completed its execution.

```
gustav@bh1 $ h5cc -o h5_crtgrpd h5_crtgrpd.c
gustav@bh1 $ h5dump groups.h5
HDF5 "groups.h5" {
  GROUP "/" {
    GROUP "MyGroup" {
      GROUP "Group_A" {
      }
      GROUP "Group_B" {
      }
    }
  }
}
gustav@bh1 $ ./h5_crtgrpd
gustav@bh1 $ h5dump groups.h5
HDF5 "groups.h5" {
  GROUP "/" {
    GROUP "MyGroup" {
      GROUP "Group_A" {
        DATASET "dset2" {
          DATATYPE H5T_STD_I32BE
          DATASPACE SIMPLE { ( 2, 10 ) / ( 2, 10 ) }
          DATA {
            1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
            1, 2, 3, 4, 5, 6, 7, 8, 9, 10
          }
        }
      }
      GROUP "Group_B" {
      }
      DATASET "dset1" {
        DATATYPE H5T_STD_I32BE
        DATASPACE SIMPLE { ( 3, 3 ) / ( 3, 3 ) }
        DATA {
          1, 2, 3,
          1, 2, 3,
          1, 2, 3
        }
      }
    }
  }
}
gustav@bh1 $

```

The program begins by generating data for the two datasets that are going to be created on `groups.h5`:

```
/* Initialize the first dataset. */
for (i = 0; i < 3; i++)
  for (j = 0; j < 3; j++)
    dset1_data[i][j] = j + 1;

/* Initialize the second dataset. */
for (i = 0; i < 2; i++)
  for (j = 0; j < 10; j++)

```

```
dset2_data[i][j] = j + 1;
```

The HDF5 file is then opened and the dataspace is prepared for the first dataset:

```
/* Open an existing file. */
file_id = H5Fopen(FILE, H5F_ACC_RDWR, H5P_DEFAULT);

/* Create the data space for the first dataset. */
dims[0] = 3;
dims[1] = 3;
dataspace_id = H5Screate_simple(2, dims, NULL);
```

The dataset is then created in the /MyGroup HDF5 group:

```
/* Create a dataset in group "MyGroup". */
dataset_id = H5Dcreate(file_id, "/MyGroup/dset1", H5T_STD_I32BE, dataspace_id,
                      H5P_DEFAULT);

/* Write the first dataset. */
status = H5Dwrite(dataset_id, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT,
                  dset1_data);

/* Close the data space for the first dataset. */
status = H5Sclose(dataspace_id);

/* Close the first dataset. */
status = H5Dclose(dataset_id);
```

The data itself is written, then the dataspace is closed first and then the dataset itself is closed.

In order to create the second dataset we open the group /MyGroup/Group\_A first, because this is where we are going to create the dataset:

```
group_id = H5Gopen(file_id, "/MyGroup/Group_A");
```

Now we create the dataspace for the second dataset:

```
/* Create the data space for the second dataset. */
dims[0] = 2;
dims[1] = 10;
dataspace_id = H5Screate_simple(2, dims, NULL);
```

and create and write the dataset itself. Observe that this time we use the group identifier for the first argument and the relative name of the dataset for the second in the call to H5Dcreate:

```
/* Create the second dataset in group "Group_A". */
dataset_id = H5Dcreate(group_id, "dset2", H5T_STD_I32BE, dataspace_id, H5P_DEFAULT);

/* Write the second dataset. */
status = H5Dwrite(dataset_id, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT,
                  dset2_data);
```

Now we have to close the dataspace, the dataset, the group, and finally the file:

```
/* Close the data space for the second dataset. */
status = H5Sclose(dataspace_id);

/* Close the second dataset */
status = H5Dclose(dataset_id);
```

```
/* Close the group. */
status = H5Gclose(group_id);

/* Close the file. */
status = H5Fclose(file_id);
```

## 7.4 Property Lists

So far we have used a *default* property list, `H5P_DEFAULT`, wherever such was required. Property lists are a lot more in HDF5 than just a decoration or the means of tweaking I/O here and there. Numerous functions, including MPI-IO based *parallel IO* are activated by the means of property lists. Semantically, the device of a property list saves HDF5 developers from having to overload HDF5 functions with an excessive number of parameters, many of which may not be normally used at all.

HDF5 property lists are opaque objects, which are manipulated by invoking special functions only, and this is just as well, because it reduces the chance of a programmer making a mistake.

Property lists are used to customize operations such as

- file creation
- accessing a file
- dataset creation
- dataset read/write

For every one of these operations HDF5 provides an operation specific default property list, which can be then customized with various functions from the H5P family (where “P” stands for *Property*). And so we have for:

**file creation** `H5P_FILE_CREATE` default property list;

**accessing a file** `H5P_FILE_ACCESS` default property list;

**dataset creation** `H5P_DATASET_CREATE` default property list;

**dataset read/write** `H5P_DATASET_XFER` default property list.

What sort of customizations can we ask for by modifying the default property lists? You can see them all quickly, if you connect to <http://hdf.ncsa.uiuc.edu/HDF5/doc/RM.H5P.html>. You will find there:

- 9 functions for customizing file creation properties;
- 35 functions for customizing file access properties including 4 for interaction with MPI;
- 24 functions for customizing dataset creation;

- 23 functions for customizing dataset read/write including 2 for interaction with MPI.

Now, roughly half of these functions are **get** functions, which just *get* you the existing property of some type, and half are **set** functions, which *set* a specified property in the list, so this whole property list business is not as intimidating as it seems to be at first glance. But there is still a lot of hidden functionality here.

Some specific properties that can be activated or de-activated are as follows:

**file creation** properties:

**user block** HDF5 may leave a block of certain size at the beginning of the file, for the user to fill with whatever non-HDF5 data the user wants.

**byte size of offsets and lengths** the byte sizes of offsets and lengths on HDF5 files can be set to be 2, 4, 8 or 16. Normally you will probably want them 8-bytes long, but they may default to 4-bytes long on IA32 systems (e.g., AVIDD) – you may need to check this and then correct.

**sizing the symbol table** HDF5 files contain directories (or *groups*) within, organized hierarchically. This is done in a way that is similar to a directory structure, i.e., there is a symbol table there, which is used to look up a specific group or a dataset. You can set the size of parameters used to control the symbol table nodes.

**sizing B-trees for chunked datasets** So far we have seen contiguous and rigorously pre-sized datasets. But HDF5 datasets can be extended dynamically. This is done, again, in a way that is similar to how files are written on disks. They are not normally written contiguously. The writing process jumps all over the disk writing the file wherever it finds space. The locations of the *chunks* of data are then stored on a B-tree, which has to be traversed in order to read the whole file. You can also store data similarly *within* an HDF5 file and a *chunked* dataset will then be described by a B-tree, whose parameters can be controlled by the means of property lists.

**file access** properties:

**memory caching** HDF5 can cache whole files in memory on a specific request. All IO operations are then done against memory, and the file itself may never even be written to disk – unless, again, specifically requested. Alternatively, HDF5 can be asked to use a specifically sized caches for the metadata and for the raw data for files that are not going to be memory cached in entirety.

**file families** If you work with a file system that imposes a limit on the size of the file, e.g., 2GBs (common to 32-bit UFS and NFS), and

your dataset exceeds this, you have the option of writing your single logical HDF5 file *physically* in the form of a family of files, all below the file system size limit.

**logging** You can activate logging on all IO requests against an HDF5 file.

**splitting** You can split a single logical HDF5 file so that its metadata and data live on separate physical files. This is similar to old MacOS file forks.

**MPI access** MPI files are associated with MPI communicators, and they may have MPI-IO info structures, that contain hints, associated with them too. If an MPI file is to be written in HDF5 format, then the communicator and the info structure must be passed to HDF5 file access functions in the form of a property list.

**Globus hooks** In order to operate on files in the Globus environment, the user must provide Globus hints on a special Globus info structure (much like the MPI info structure). These are then passed to HDF5 processes by the means of a property list.

**SRB hooks** HDF5 can also co-exist with the SDSC's Storage Request Broker. SRB also requires an info structure to operate on SRB files and this can be passed through a property list.

**Streams** HDF5 files can be made to *stream* directly into IO-sockets

**dataset creation** properties:

**Dataset layout** HDF5 data sets can be laid out in three ways. First, if the dataset itself is very small, it can be stored in entirety, in the object header. This is similar to storing very small files within an i-node in some file systems. Second, the dataset may be stored contiguously, and then we may as well request that it be chunked instead, in which case, the data may be physically scattered throughout the whole file – in chunks. The size of the chunks can be customized too.

**Data compression** We may request that data stored on an HDF5 file be compressed. Both the compression method and the degree of compression may be selected.

**Data filtering** Compression is a form of data filtering, but we may also request that a user-defined filter be applied to data streams on writing and reading datasets. The filter may be used, e.g., to encrypt the data, or to carry out on the fly selection of data.

**dataset read/write** properties:

**Error detection** We may enable error detection on reads and writes. Normally devices such as individual disk drives and disk arrays do hardware level error detection anyway. Here you can add your own additional error detection method, e.g., checksum.



**Callbacks** If you have defined your own data filtering, here you may additionally define a callback, which is going to be activated when the filter fails.

**MPI-IO** You can specify here whether a write or a read operation against an HDF5/MPI-IO file is to be *collective* or *independent*.

**Blocking** If you do a lot of small reads and writes, you can request that these operations be blocked, i.e., HDF5 will collect all I/O until the block is full, and then only will the block be transferred to the media. You can request a specific size of the block.

These are not *all* properties and features available. They are just the ones that caught my eye.

Even though HDF5 provides a lot of functionality here, you should not run amok with it. Remember that IO is *always* going to be *orders of magnitude* slower than computation and memory data access. Consequently, the best way to do IO is to do as little of it as possible. Get all the data you need into memory, if you can, and then operate on it there. After you finish, write it out and update the file. Read and write in large blocks rather than in tiny amounts. Do not use files for temporary scratch space, and certainly not to communicate between processes. The so-called “out-of-core” jobs are unbelievably wasteful. If you lack sufficient memory on, say, 8 nodes, go for 32 nodes, but always try to fit *all* data you need to compute on in memory itself.

### 7.4.1 Modifying Property Lists

In this section I’m going to show you how to create desired property lists and how to pass them to functions. The procedure is fairly simple and consists of two steps basically:

1. Create a default property list for a given operation by calling function `H5Pcreate`.
2. Modify the property list generated above by calling a specific property list modification function.

For example, in order to create a file with 64-bit objects offsets and lengths you would do as follows (this example is taken from the HDF5 Tutorial at NCSA):

```
hid_t create_plist;
hid_t file_id;

[...]

create_plist = H5Pcreate(H5P_FILE_CREATE);
H5Pset_sizes(create_plist, 8, 8);

file_id = H5Fcreate("test.h5", H5F_ACC_TRUNC,
                  create_plist, H5P_DEFAULT);

[...]
```

```
H5Fclose(file_id);
```

Here we create the default property list with `H5Pcreate`, which is invoked with the `H5P_FILE_CREATE` specifier, first, and then modify the list by calling function `H5Pset_sizes`, which takes the property list *identifier* as its first argument, byte-size of object offsets as its second argument and byte-size of object lengths as its third argument. Once we have the list to our liking, we can use it in the call to `H5Fcreate`.

It is OK to call more than one list modification function, if you need more features, of course.

In the next couple of sub-sections, I am going to discuss various examples of using property lists, as applied to accessing files, creating datasets and then writing and reading them. MPI-IO/HDF5 examples will be shown here too.

## 7.4.2 Create a File Family

This simple example code creates a file family and then fills the family with data. Even though physically we are going to write on multiple files, logically, we treat the IO operation like a single HDF5 dataset write.

Here is the code (this example is also taken from the NCSA HDF5 Tutorial):

```
/*
 * This example writes data to the HDF5 file.
 * Data conversion is performed during write operation.
 */

#include "hdf5.h"

#define DIM0 40
#define DIM1 60

#define H5FILE_NAME "SDS%d.h5"
int
main (void)
{
    hid_t      file, dataset;          /* handles */
    hid_t      dataspace;
    hid_t      plist_id;
    int        i, j, dset_data[DIM0][DIM1];
    hsize_t    family_size = 1000;
    hsize_t    dims[2] = {DIM0,DIM1};
    herr_t     status;

    plist_id = H5Pcreate(H5P_FILE_ACCESS);
    status = H5Pset_fapl_family(plist_id, family_size, H5P_DEFAULT);

    file = H5Fcreate(H5FILE_NAME, H5F_ACC_TRUNC, H5P_DEFAULT, plist_id);
    dataspace = H5Screate_simple(2, dims, NULL);

    dataset = H5Dcreate (file, "family-dataset", H5T_NATIVE_INT, dataspace,
H5P_DEFAULT);

    for (i = 0; i < DIM0; i++)
```

```

    for (j = 0; j < DIM1; j++)
        dset_data[i][j] = i * 6 + j + 1;

    status = H5Dwrite(dataset, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT,
                    dset_data);

    status = H5Dclose(dataset);
    status = H5Sclose(dataspace);
    status = H5Fclose(file);
}

```

The program is compiled with:

```

gustav@bh1 $ h5cc -o h5_family h5_family.c
gustav@bh1 $

```

and then it is run by simply typing its name:

```

gustav@bh1 $ ./h5_family
gustav@bh1 $

```

After the program returns, you can see that it has generated 12 data files:

```

gustav@bh1 $ ls
SDS0.h5  SDS11.h5  SDS4.h5  SDS7.h5  h5_family
SDS1.h5  SDS2.h5  SDS5.h5  SDS8.h5  h5_family.c
SDS10.h5 SDS3.h5  SDS6.h5  SDS9.h5  h5_family.o
gustav@bh1 $

```

You cannot view any of the HDF5 data files in this family with a simple call to `h5dump` followed by an explicit name of any of the files. To view this data do as follows:

```

gustav@bh1 $ h5dump -f family SDS%d.h5
HDF5 "SDS%d.h5" {
GROUP "/" {
  DATASET "family-dataset" {
    DATATYPE H5T_STD_I32LE
    DATASPACE SIMPLE { ( 40, 60 ) / ( 40, 60 ) }
    DATA {
      1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
      [...]
      277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290,
      291, 292, 293, 294
    }
  }
}
}
gustav@bh1 $

```

You can also call `h5dump` without the `-f family` switch, in which case `h5dump` will try five different drivers against the file name you have given it, eventually finding that it can read it with the `family` driver.

Let us have a look at the program itself now. We begin by creating the default property list for file access with `H5Pcreate` and then modify it by calling `H5Pset_fapl_family`:

```

plist_id = H5Pcreate(H5P_FILE_ACCESS);
status = H5Pset_fapl_family(plist_id, family_size, H5P_DEFAULT);

```

Function `H5Pset_fapl_family` takes the property list identifier as its first argument. The second argument is the size of each file family member *in bytes*. The third argument is the property list to be used for each of the file family member, in this case just `H5P_DEFAULT`.

Now we create the HDF5 *logical* file itself:

```

file = H5Fcreate(H5FILE_NAME, H5F_ACC_TRUNC, H5P_DEFAULT, plist_id);

```

Observe that `H5FILE_NAME` is defined as `"SDS%d.h5"`. Once the file family is created `%d` will be replaced with an appropriate number. The third argument in the call to `H5Fcreate` is the file create property list, whereas the fourth argument is the file access property list. The specific property we are talking about here, i.e., the property that the file is to be split into the family, is an *access* property, and so it is the fourth argument that has been customized to convey our request.

The next couple of steps in the program are standard. We create a simple data space, then we create the dataset, then we initialize the data in memory and write the data on the dataset.

Finally we close the dataset, close the dataspace and close the file.

At no stage do we have to worry about dealing with multiple files. HDF5 takes care of this. Logically, we have opened a single HDF5 file for writing in this program.

### 7.4.3 Create an MPI-IO File

In this section we are going to create an MPI file. It is going to be an HDF5 file too, so it won't be empty, even though we are not going to write anything on it explicitly yet. The file is going to be opened by all processes in the `MPI_COMM_WORLD` communicator.

Here is the program taken from the NCSA HDF5 Tutorial, Creating/Accessing a File with PHDF5

```

/*
 * This example creates an HDF5 file.
 */

#include "hdf5.h"

#define H5FILE_NAME    "SDS_row.h5"

int
main (int argc, char **argv)
{
    /*
     * HDF5 APIs definitions
     */
    hid_t      file_id;          /* file and dataset identifiers */
    hid_t      plist_id;        /* property list identifier( access template) */
    herr_t      status;

```

```

/*
 * MPI variables
 */
int mpi_size, mpi_rank;
MPI_Comm comm = MPI_COMM_WORLD;
MPI_Info info = MPI_INFO_NULL;

/*
 * Initialize MPI
 */
MPI_Init(&argc, &argv);
MPI_Comm_size(comm, &mpi_size);
MPI_Comm_rank(comm, &mpi_rank);

/*
 * Set up file access property list with parallel I/O access
 */
plist_id = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_fapl_mpio(plist_id, comm, info);

/*
 * Create a new file collectively.
 */
file_id = H5Fcreate(H5FILE_NAME, H5F_ACC_TRUNC, H5P_DEFAULT, plist_id);

/*
 * Close property list.
 */
H5Pclose(plist_id);

/*
 * Close the file.
 */
H5Fclose(file_id);

MPI_Finalize();

return 0;
}

```

Observe that the program does not include `<mpi.h>` explicitly. This is because it is included by `<hdf5.h>` through `<H5public.h>`. We compile this program with `h5cc`, which invokes `mpicc` internally:

```

gustav@bh1 $ h5cc -o MPI_file_create MPI_file_create.c
gustav@bh1 $ mv MPI_file_create ~/bin/MPI_file_create

```

After compilation and linking I have moved the binary to my `~/bin` so that it will be available on other nodes when I run it under `mpiexec`. Then I start the MPD machine to run the program:

```

gustav@bh1 $ mpdboot
gustav@bh1 $ mpdtrace | wc
    26    26   129
gustav@bh1 $

```

and go to GPFS to run it:

```

gustav@bh1 $ cd /N/gpfs/gustav
gustav@bh1 $ mpiexec -n 8 MPI_file_create
gustav@bh1 $ ls
SDS_row.h5  t_mpio_1wMr  test  tests
gustav@bh1 $

```

There are various other files and directories in this area, but `SDS_row.h5` has been created and it is a genuine HDF5 file as you can see if you run `h5dump` on it:

```

gustav@bh1 $ h5dump SDS_row.h5
HDF5 "SDS_row.h5" {
GROUP "/" {
}
}
gustav@bh1 $

```

Now let us have a look at the program itself. This is a typical simple MPI program, which begins with the usual:

```

MPI_Init(&argc, &argv);
MPI_Comm_size(comm, &mpi_size);
MPI_Comm_rank(comm, &mpi_rank);

```

But right after this we invoke HDF5 functions. First we create a default file access list:

```

plist_id = H5Pcreate(H5P_FILE_ACCESS);

```

and then we invoke function `H5Pset_fapl_mpio`:

```

H5Pset_fapl_mpio(plist_id, comm, info);

```

which is going to associate the file with the `MPI_COMM_WORLD` communicator and with the `MPI_INFO_NULL` info object, because we have declared that:

```

MPI_Comm comm = MPI_COMM_WORLD;
MPI_Info info = MPI_INFO_NULL;

```

Now we create the file itself:

```

file_id = H5Fcreate(H5FILE_NAME, H5F_ACC_TRUNC, H5P_DEFAULT, plist_id);

```

and this is all we are going to do in this program. Having created the file, we close the property list, then close the file itself and finally we close MPI with the call to `MPI_Finalize` – remember that this is an MPI program after all:

```

H5Pclose(plist_id);
H5Fclose(file_id);
MPI_Finalize();
return 0;
}

```

#### 7.4.4 Create an Extendible Dataset

In our previous examples, the datasets were of fixed pre-defined size. HDF5 lets you define datasets that can grow. But datasets like these have to be *chunked*, in other words, if they are to grow, you must allow for the data itself to be physically scattered around the file in chunks.

The following program, which is a little on the large side (by the standards of this tutorial, which tries to keep programming examples very short), illustrates how to create and then manipulate such an extendible dataset. So you will see a few more new elements in it – but, read on, I'll explain it all. The program is taken from the NCSA HDF5 Tutorial.

Here is the program:

```
#include "hdf5.h"

#define FILE      "ext.h5"
#define DATASETNAME "ExtendibleArray"
#define RANK      2

int
main (void)
{
    hid_t      file;                /* handles */
    hid_t      dataspace, dataset;
    hid_t      filespace;
    hid_t      cparms;
    hid_t      memspace;

    hsize_t    dims[2] = { 3, 3};   /* dataset dimensions
                                     at creation time */
    hsize_t    dims1[2] = { 3, 3};  /* data1 dimensions */
    hsize_t    dims2[2] = { 7, 1};  /* data2 dimensions */

    hsize_t    maxdims[2] = {H5S_UNLIMITED, H5S_UNLIMITED};
    hsize_t    size[2];
    hssize_t   offset[2];
    hsize_t    i,j;
    herr_t     status, status_n;
    int        data1[3][3] = { {1, 1, 1}, /* data to write */
                               {1, 1, 1},
                               {1, 1, 1} };
    int        data2[7] = { 2, 2, 2, 2, 2, 2, 2};

    /* Variables used in reading data back */
    hsize_t    chunk_dims[2] = {2, 5};
    hsize_t    chunk_dimsr[2];
    hsize_t    dimsr[2];
    int        data_out[10][3];
    int        rank, rank_chunk;

    /* Create the data space with unlimited dimensions. */
    dataspace = H5Screate_simple (RANK, dims, maxdims);

    /* Create a new file. If file exists its contents will be overwritten. */
    file = H5Fcreate (FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

    /* Modify dataset creation properties, i.e. enable chunking */
    cparms = H5Pcreate (H5P_DATASET_CREATE);
    status = H5Pset_chunk ( cparms, RANK, chunk_dims);

    /* Create a new dataset within the file using cparms
       creation properties. */
}
```

```

dataset = H5Dcreate (file, DATASETNAME, H5T_NATIVE_INT, dataspace,
                    cparms);

/* Extend the dataset. This call assures that dataset is 3 x 3.*/
size[0] = 3;
size[1] = 3;
status = H5Dextend (dataset, size);

/* Select a hyperslab */
fileSpace = H5Dget_space (dataset);
offset[0] = 0;
offset[1] = 0;
status = H5Sselect_hyperslab (fileSpace, H5S_SELECT_SET, offset, NULL,
                              dims1, NULL);

/* Write the data to the hyperslab */
status = H5Dwrite (dataset, H5T_NATIVE_INT, dataspace, fileSpace,
                  H5P_DEFAULT, data1);

/* Extend the dataset. Dataset becomes 10 x 3 */
dims[0] = dims1[0] + dims2[0];
size[0] = dims[0];
size[1] = dims[1];
status = H5Dextend (dataset, size);

/* Select a hyperslab */
fileSpace = H5Dget_space (dataset);
offset[0] = 3;
offset[1] = 0;
status = H5Sselect_hyperslab (fileSpace, H5S_SELECT_SET, offset, NULL,
                              dims2, NULL);

/* Define memory space */
dataspace = H5Screate_simple (RANK, dims2, NULL);

/* Write the data to the hyperslab */
status = H5Dwrite (dataset, H5T_NATIVE_INT, dataspace, fileSpace,
                  H5P_DEFAULT, data2);

/* Close resources */
status = H5Dclose (dataset);
status = H5Sclose (dataspace);
status = H5Sclose (fileSpace);
status = H5Fclose (file);

/*****
Read the data back
*****/

file = H5Fopen (FILE, H5F_ACC_RDONLY, H5P_DEFAULT);
dataset = H5Dopen (file, DATASETNAME);
fileSpace = H5Dget_space (dataset);
rank = H5Sget_simple_extent_ndims (fileSpace);
status_n = H5Sget_simple_extent_dims (fileSpace, dimsr, NULL);

cparms = H5Dget_create_plist (dataset);
if (H5D_CHUNKED == H5Pget_layout (cparms))

```



```

{
    rank_chunk = H5Pget_chunk (cparms, 2, chunk_dimsr);
}

memspace = H5Screate_simple (rank,dimsr,NULL);
status = H5Dread (dataset, H5T_NATIVE_INT, memspace, filespace,
                 H5P_DEFAULT, data_out);

printf("\n");
printf("Dataset: \n");
for (j = 0; j < dimsr[0]; j++)
{
    for (i = 0; i < dimsr[1]; i++)
        printf("%d ", data_out[j][i]);
    printf("\n");
}

status = H5Pclose (cparms);
status = H5Dclose (dataset);
status = H5Sclose (filespace);
status = H5Sclose (memspace);
status = H5Fclose (file);
}

```

This is a sequential program, so it is compiled, linked and run in the usual HDF5 way:

```

gustav@bh1 $ h5cc -o h5_extend h5_extend.c
gustav@bh1 $ ./h5_extend

```

```

Dataset:
1 1 1
1 1 1
1 1 1
2 0 0
2 0 0
2 0 0
2 0 0
2 0 0
2 0 0
2 0 0
2 0 0
gustav@bh1 $

```

But when you run `h5dump` on the HDF5 data file generated by this program you will notice something new:

```

gustav@bh1 $ h5dump ext.h5
HDF5 "ext.h5" {
GROUP "/" {
  DATASET "ExtendibleArray" {
    DATATYPE  H5T_STD_I32LE
    DATASPACE SIMPLE { ( 10, 3 ) / ( H5S_UNLIMITED, H5S_UNLIMITED ) }
    DATA {
      1, 1, 1,
      1, 1, 1,
      1, 1, 1,
      2, 0, 0,
      2, 0, 0,
      2, 0, 0,
      2, 0, 0,
    }
  }
}

```

```

        2, 0, 0,
        2, 0, 0,
        2, 0, 0
    }
}
}
gustav@bh1 $

```

The new element to notice is the annotation:

```
DATASPACE SIMPLE { ( 10, 3 ) / ( H5S_UNLIMITED, H5S_UNLIMITED ) }
```

which says that the dataset currently of size  $10 \times 3$  can grow to unlimited size. Previously we would have an annotation such as:

```
DATASPACE SIMPLE { ( 40, 60 ) / ( 40, 60 ) }
```

which says that this particular dataset is of size  $40 \times 60$  and cannot be resized.

The above program resizes the dataset twice, before the dataset itself and then the file are closed.

So, let us discuss the program now.

The program begins with the creation of a simple  $3 \times 3$  dataspace, which is allowed to grow *without limits*:

```

#define RANK          2
...
hsize_t      dims[2]  = { 3, 3};
...
hsize_t      maxdims[2] = {H5S_UNLIMITED, H5S_UNLIMITED};
...
dataspace = H5Screate_simple (RANK, dims, maxdims);

```

Then we create the new HDF5 file in the usual way, assuming default property lists:

```
file = H5Fcreate (FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
```

and then we get down to generating the property list for the dataset, which specifies the chunking:

```

#define RANK          2
...
hsize_t      chunk_dims[2] = {2, 5};
...
cparms = H5Pcreate (H5P_DATASET_CREATE);
status = H5Pset_chunk ( cparms, RANK, chunk_dims);

```

Function `H5Pset_chunk` takes three arguments: the first one is the property list identifier, then the rank of the array that specifies the chunking. Here we have a two-dimensional data space, and so our chunk array *has to be* two-dimensional too. The chunking in the first dimension is going to be 2 (of elementary data items) and the chunking in the second dimension is going to be 5 (of elementary data items).

Now we create the dataset itself:

```
#define DATASETNAME "ExtendibleArray"
...
    dataset = H5Dcreate (file, DATASETNAME, H5T_NATIVE_INT, dataspace,
                        cparms);
```

and here we also say that the data in the dataset should be stored in the native integer format. When we looked at the file itself though, we saw `DATATYPE H5T_STD_I32LE` there, even though the program does not specify this anywhere explicitly. From this we infer that IA32's native format is little-endian.

This call:

```
size[0] = 3;
size[1] = 3;
status = H5Dextend (dataset, size);
```

is not really necessary, because we have already defined our dataspace to be  $3 \times 3$ , but it doesn't hurt either. We are merely confirming the dataset to be  $3 \times 3$  here.

Now the program embarks on manipulations that are a little too soon in our course, because they refer to *hyperslab selection*. Here is what happens. The first call to `H5Dget_space`

```
file_space = H5Dget_space (dataset);
```

returns an identifier for a copy of the dataspace for the dataset. In this space we are now going to select a hyperslab by calling function `H5Sselect_hyperslab`

```
hsize_t    dims1[2] = { 3, 3};
...
offset[0] = 0;
offset[1] = 0;
status = H5Sselect_hyperslab (file_space, H5S_SELECT_SET, offset, NULL,
                              dims1, NULL);
```

This function call goes to the dataspace pointed to by the `file_space` identifier and selects a portion of data, which is  $3 \times 3$  and begins at the beginning of the dataset, because the offsets are both zero. The fourth parameter, which is `NULL`, is used to define stride and the last parameter, which is `NULL` too, is used to define the size of the block. In both cases `NULL` invokes defaults, i.e., the stride is 1 and the block size is 1 too. With this function, as you see, you can pick up portions of data from the dataset, but in this case we are just selecting all there is at the moment, because our dataspace at present is  $3 \times 3$ .

Now we write the data to the dataset:

```
int        data1[3][3] = { {1, 1, 1},      /* data to write */
                          {1, 1, 1},
                          {1, 1, 1} };
...
status = H5Dwrite (dataset, H5T_NATIVE_INT, dataspace, file_space,
                  H5P_DEFAULT, data1);
```

Well, if this was not an extendible dataset, we could not write to it any more, because it would be already full. So now we are going to extend it, for real this time:

```

hsize_t      dims[2]  = { 3, 3};
hsize_t      dims1[2] = { 3, 3};
hsize_t      dims2[2] = { 7, 1};
...
dims[0]      = dims1[0] + dims2[0];
size[0]      = dims[0];
size[1]      = dims[1];
status = H5Dextend (dataset, size);

```

This time the dataset is going to be  $10 \times 3$ . We have already filled its top  $3 \times 3$  portion with data. Now we want to select a hyperslab that covers the as yet unused space and write something there:

```

hsize_t      dims2[2] = { 7, 1};
...
file_space = H5Dget_space (dataset);
offset[0] = 3;
offset[1] = 0;
status = H5Sselect_hyperslab (file_space, H5S_SELECT_SET, offset, NULL,
                             dims2, NULL);

```

We create new dataspace for this new hyperslab and then write new data into it:

```

#define RANK      2
hsize_t      dims2[2] = { 7, 1};
int          data2[7]  = { 2, 2, 2, 2, 2, 2, 2};
...
dataspace = H5Screate_simple (RANK, dims2, NULL);
status = H5Dwrite (dataset, H5T_NATIVE_INT, dataspace, file_space,
                  H5P_DEFAULT, data2);

```

Observe that this dataspace is  $7 \times 1$ , i.e., we are going to write on the first column only, and only on rows 4 through 10. Before I go any further, let me again bring back the `h5dump` of the file:

```

HDF5 "ext.h5" {
GROUP "/" {
  DATASET "ExtendibleArray" {
    DATATYPE H5T_STD_I32LE
    DATASPACE SIMPLE { ( 10, 3 ) / ( H5S_UNLIMITED, H5S_UNLIMITED ) }
    DATA {
      1, 1, 1,
      1, 1, 1,
      1, 1, 1,
      2, 0, 0,
      2, 0, 0,
      2, 0, 0,
      2, 0, 0,
      2, 0, 0,
      2, 0, 0,
      2, 0, 0,
      2, 0, 0
    }
  }
}
}

```

As you see we have indeed written one-s on the top  $3 \times 3$  part of the space and then added a column of two-s on the left hand side of the data space.

Having done all this, we close the dataset, then the dataspace, then the filespace and finally the file itself.

The second part of the program opens the file for reading, then opens the dataset and then extracts information about it as follows:

```
file = H5Fopen (FILE, H5F_ACC_RDONLY, H5P_DEFAULT);
dataset = H5Dopen (file, DATASETNAME);
filespace = H5Dget_space (dataset);
rank = H5Sget_simple_extent_ndims (filespace);
status_n = H5Sget_simple_extent_dims (filespace, dimsr, NULL);
```

The two new functions here are `H5Sget_simple_extent_ndims` and `H5Sget_simple_extent_dims`. The first one returns the dimensionality of the dataspace and the second one returns the size in each dimension, here on `dimsr`, and the maximum size, here on `NULL`, which means that we don't care about this.

Now we are going to enquire about the data layout on the file by extracting the property list from the dataset and then checking the list itself with `H5Pget_layout`

```
cparms = H5Dget_create_plist (dataset);
if (H5D_CHUNKED == H5Pget_layout (cparms))
{
    rank_chunk = H5Pget_chunk (cparms, 2, chunk_dimsr);
}
```

Function `H5Dget_create_plist` returns an identifier for a copy of the dataset creation property list for the dataset. Once we have the list we check if the data was chunked and if it was we retrieve the size of chunks by calling function `H5Pget_chunk`. The function returns chunking on the last parameter, here it is `chunk_dimsr`.

Having performed these interrogations, only to show how it's done, because we don't really do anything with the returned data about chunking in this program, we get down to reading the data itself. First we create the memory space, then read the data itself and print it on standard output:

```
memspace = H5Screate_simple (rank,dimsr,NULL);
status = H5Dread (dataset, H5T_NATIVE_INT, memspace, filespace,
                 H5P_DEFAULT, data_out);

printf("\n");
printf("Dataset: \n");
for (j = 0; j < dimsr[0]; j++)
{
    for (i = 0; i < dimsr[1]; i++)
        printf("%d ", data_out[j][i]);
    printf("\n");
}
```

Finally we have to close all that's opened before exiting the program:

```
status = H5Pclose (cparms);
status = H5Dclose (dataset);
status = H5Sclose (filespace);
status = H5Sclose (memspace);
status = H5Fclose (file);
}
```

### 7.4.5 Create a Compressed Dataset

Default writes on HDF5 datasets are neither compressed nor is error checking activated for them either. Both can be turned on by the means of dataset creation property lists. In this section I am going to show you how to activate compression.

The following program, taken from the NCSA HDF5 Tutorial, does the following. First it creates a standard HDF5 data file called `zip.h5`. A group `/Data` is then created in the file. Then we get down to generate a property list for the dataset creation. The list is going to activate *two* features: chunking and compression. Then we create the dataset `/Data/Compressed_Data` using the list. The data itself is generated, then written on the dataset. At this stage we close the dataspace, the dataset, the group and the file.

Then we re-open the file, the group and the dataset. The data is read in full. There is no need for any hocus pocus with property lists here, because the required property is already attached to the dataset on the file and HDF5 learns about it when it opens the dataset. The decompression is activated automatically when the data is read. Having read the data we print a small portion of it on standard output, then close the dataset, the group and the file.

Here's the program:

```

/* Create compressed dataset */

#include "hdf5.h"

#define FILE    "zip.h5"

/* Uncomment to remove compression and
   comment out line above
#define FILE    "unzip.h5"
*/

#define RANK    2

int
main(void)
{
    hid_t    file, grp;
    hid_t    dataset, dataspace;
    hid_t    plist;

    herr_t    status;
    hsize_t    dims[2];
    hsize_t    cdims[2];

    int    idx;
    int    i,j;
    int    buf[1000][20];
    int    rbuf [1000][20];

    /*
     * Create a file.
     */

```

```

file = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
printf ("H5Fcreate returns: %d\n", file);

/*
 * Create a group in the file.
 */
grp = H5Gcreate(file, "/Data", 0);
printf ("H5Gcreate returns: %d\n", grp);

/*
 * Create dataset "Compressed Data" in the group using absolute
 * name. Dataset creation property list is modified to use
 * GZIP compression with the compression effort set to 6.
 * Note that compression can be used only when dataset is chunked.
 */
dims[0] = 1000;
dims[1] = 20;
cdims[0] = 20;
cdims[1] = 20;
dataspace = H5Screate_simple(RANK, dims, NULL);
printf ("H5Screate_simple: %d\n", dataspace);

/* Uncomment this section if you want to use GZIP compression
   Be sure to comment out the line following, as well.
*/
plist = H5Pcreate(H5P_DATASET_CREATE);
printf ("H5Pcreate returns: %d\n", plist);
status = H5Pset_chunk(plist, 2, cdims);
printf ("H5Pset_chunk returns: %d\n", status);
status = H5Pset_deflate( plist, 6);
printf ("H5Pset_deflate returns: %d\n", status);

dataset = H5Dcreate(file, "/Data/Compressed_Data", H5T_STD_I32BE,
                   dataspace, plist);

/*
dataset = H5Dcreate(file, "/Data/Uncompressed_Data", H5T_STD_I32BE,
                   dataspace, H5P_DEFAULT);
*/

printf ("H5Dcreate returns: %d\n", dataset);

for (i = 0; i < dims[0]; i++) {
    for (j=0; j<dims[1]; j++) {
        buf[i][j] = i+j;
    }
}
status = H5Dwrite(dataset, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT, buf);
printf ("H5Dwrite: %d\n", status);

status = H5Sclose(dataspace);
printf ("H5Sclose: %d\n", status);

status = H5Dclose(dataset);

```

```

printf ("H5Dclose: %d\n", status);

status = H5Gclose (grp);
printf ("H5Gclose: %d\n", status);

status = H5Fclose(file);
printf ("H5Fclose: %d\n", status);

/*
 * Now reopen the file and group in the file.
 */
file = H5Fopen(FILE, H5F_ACC_RDWR, H5P_DEFAULT);
printf ("H5Fopen: %d\n", file);
grp = H5Gopen(file, "Data");
printf ("H5Gopen: %d\n", grp);

dataset = H5Dopen(grp, "Compressed_Data");

/* Uncomment, if removing compression
and comment out line above
dataset = H5Dopen(grp, "Uncompressed_Data");
*/
printf ("H5Dopen: %d\n", dataset);

status = H5Dread (dataset, H5T_NATIVE_INT, H5S_ALL, H5S_ALL,
                 H5P_DEFAULT, rbuf);

printf ("\nData (10 lines):\n");

for (i=0; i<10; i++)
{
    for (j=0; j<20; j++)
        printf(" %d", rbuf[i][j]);
    printf ("\n");
}

status = H5Dclose(dataset);
printf ("\nH5Dclose: %d\n", status);

status = H5Gclose (grp);
printf ("H5Gclose: %d\n", status);

status = H5Fclose(file);
printf ("H5Fclose: %d\n", status);

}

```

The program is compiled and linked with `h5cc` and run normally by invoking its name:

```

gustav@bh1 $ h5cc -o h5_zip h5_zip.c
gustav@bh1 $ ./h5_zip
H5Fcreate returns: 67108864
H5Gcreate returns: 201326592
H5Screate_simple: 335544322
H5Pcreate returns: 805306377
H5Pset_chunk returns: 0
H5Pset_deflate returns: 0

```



```

H5Dcreate returns: 402653184
H5Dwrite: 0
H5Sclose: 0
H5Dclose: 0
H5Gclose: 0
H5Fclose: 0
H5Fopen: 67108865
H5Gopen: 201326593
H5Dopen: 402653185

```

```
Data (10 lines):
```

```

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28

```

```

H5Dclose: 0
H5Gclose: 0
H5Fclose: 0
gustav@bh1 $

```

As the program runs it prints returns of its various internal function calls on standard output. The small portion of data printed at the end shows that the data read back from the compressed dataset is indeed correct. The data array itself is  $1000 \times 20$  and its entries are  $a_{ij} = i + j$ , i.e., 0, 1, 2, ... in the first row, then 1, 2, 3, ... in the second row, 2, 3, 4, ... in the third row and so on.

But has the data been compressed? There are  $1000 \times 20 = 20,000$  4-byte long integers in the dataset, which translates into 80,000 bytes. But the file is only 11,312 bytes long:

```

gustav@bh1 $ ls -l zip.h5
-rw-r--r-- 1 gustav ucs 11312 Nov 24 12:56 zip.h5
gustav@bh1 $

```

so the data in it indeed must have been compressed. You can run `h5dump` on this file and you'll get all the data back uncompressed. But you won't find any hint that the data in the file is compressed either. To see this look at the file with `h5ls`:

```

gustav@bh1 $ h5ls -r -v zip.h5
Opened "zip.h5" with sec2 driver.
/Data                               Group
  Location: 0:1:0:1576
  Links: 1
/Data/Compressed_Data               Dataset {1000/1000, 20/20}
  Location: 0:1:0:1952
  Links: 1
  Modified: 2003-11-24 12:56:24 EST
  Chunks: {20, 20} 1600 bytes
  Storage: 80000 logical bytes, 5316 allocated bytes, 1504.89% utilization
  Filter-0: deflate-1 OPT {6}

```

```

Type:      32-bit big-endian integer
gustav@bh1 $

```

Here you can see that the 80,000 logical bytes have been squeezed into 5,316 physical bytes and that a `deflate-1 OPT {6}` filter, as we have requested with the call to `H5Pset_deflate`:

```
status = H5Pset_deflate( plist, 6);
```

has been used.

There are no new elements in this program other than the call to `H5Pset_deflate`, so I won't discuss the program in detail. It should be easy for you to see, by now, how the program goes about its business. Function `H5Pset_deflate` takes a property list as its first argument. The function activates the GNU `gzip` algorithm on the data. If you look at the `gzip` man page, you'll see that you can regulate the compression speed by calling `gzip` with a flag such as `-1` or `-9`. `-1`, which is equivalent to `--fast`, results in very fast but not very effective compression. On the other hand `-9`, which is equivalent to `--best`, results in slow but very effective compression. You can do the same when you call `H5Pset_deflate`. The second argument is the compression speed argument from `gzip`. It can be any integer between 1 and 9.

#### 7.4.6 Activate Checksum for Error Detection

The program in this section activates checksum for error detection on writes to and reads from a dataset. The program begins by creating a simple dataspace and then filling a data array of rank 2 with integers following the formula  $a_{ij} = 6 \times i + j + 1$ . The HDF5 file `cksum.h5` is created and then we get down to constructing a property list for the dataset. The list contains two requests. The first request is chunking, and the second request is that a filter should be applied to IO operations. The requested filter is `H5Z_FILTER_FLETCHER32`, which implements checksum error detection. Finally we create a dataset `/dset` with requested properties. The write takes place, upon which we close the property list, the dataset and the file.

In the second part of the program we open the file and the the dataset. We create a property list again, but this time we use it in order to inspect whether checksum has been enabled on the data set – the result of the inspection is printed on standard output. The data is then read, but not printed on standard output. We only print the exit status of `H5Dread`. Then we close the property list, the dataset and the file.

Here is the program itself (this program originates from the NCSA HDF5 Tutorial):

```

/*
   h5cksum: This example uses the Fletcher32 checksum algorithm
           for error detection
*/

#include "hdf5.h"
#define FILE "cksum.h5"

```

```

main() {

    hid_t      file_id, dataset_id, dataspace_id, dxpl, property_id;
    hsize_t    dims[2];
    herr_t     status;
    hsize_t    chkdim[2] = {2,3};
    int        i, j, dset_data[4][6];
    H5Z_EDC_t  edc;

    /* Create the data space for the dataset. */
    dims[0] = 4;
    dims[1] = 6;
    dataspace_id = H5Screate_simple(2, dims, NULL);

    /* Initialize the dataset. */
    for (i = 0; i < 4; i++)
        for (j = 0; j < 6; j++)
            dset_data[i][j] = i * 6 + j + 1;

    /* Create a new file using default properties. */
    file_id = H5Fcreate (FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

    /* Create the dataset with checksum filter */
    property_id = H5Pcreate(H5P_DATASET_CREATE);
    status = H5Pset_chunk (property_id, 2, chkdim);
    status = H5Pset_filter (property_id, H5Z_FILTER_FLETCHER32, 0, 0, NULL);

    dataset_id = H5Dcreate (file_id, "/dset", H5T_STD_I32BE, dataspace_id,
                           property_id);

    status = H5Dwrite(dataset_id, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT,
                     dset_data);

    status = H5Pclose (property_id);
    status = H5Dclose(dataset_id);
    status = H5Fclose(file_id);

    file_id = H5Fopen(FILE, H5F_ACC_RDWR, H5P_DEFAULT);
    dataset_id = H5Dopen (file_id, "/dset");

    dxpl = H5Pcreate (H5P_DATASET_XFER);
    edc = H5Pget_edc_check (dxpl);
    if (edc == 1) printf ("Checksum is enabled\n");
    if (edc == 0) printf ("Checksum is disabled\n");

    status = H5Dread(dataset_id, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, dxpl,
                    dset_data);
    printf ("H5Dread returns: %i\n", status);

    status = H5Pclose (dxpl);
    status = H5Dclose(dataset_id);
    status = H5Fclose(file_id);

}

```

Here is how the program is compiled, linked and run:

```
gustav@bh1 $ h5cc -o h5_cksum h5_cksum.c
gustav@bh1 $ ./h5_cksum
Checksum is enabled
H5Dread returns: 0
gustav@bh1 $
```

The generated HDF5 file has a simple and small dump:

```
gustav@bh1 $ h5dump cksum.h5
HDF5 "cksum.h5" {
GROUP "/" {
  DATASET "dset" {
    DATATYPE H5T_STD_I32BE
    DATASPACE SIMPLE { ( 4, 6 ) / ( 4, 6 ) }
    DATA {
      1, 2, 3, 4, 5, 6,
      7, 8, 9, 10, 11, 12,
      13, 14, 15, 16, 17, 18,
      19, 20, 21, 22, 23, 24
    }
  }
}
}
gustav@bh1 $
```

You can see that the dataset has checksum filter attached to it when you run `h5ls` on the file:

```
gustav@bh1 $ h5ls -v -r cksum.h5
Opened "cksum.h5" with sec2 driver.
/dset          Dataset {4/4, 6/6}
  Location:    0:1:0:976
  Links:       1
  Modified:    2003-11-24 18:20:54 EST
  Chunks:      {2, 3} 24 bytes
  Storage:     96 logical bytes, 112 allocated bytes, 85.71% utilization
  Filter-0:    fletcher32-3 {}
  Type:        32-bit big-endian integer
gustav@bh1 $
```

The program itself should be easy to understand. It calls two new functions we haven't encountered yet. The first one is `H5Pset_filter` and the second one is `H5Pget_edc_check`.

Function `H5Pset_filter` adds a filter to the filter pipeline, through which data will flow to and from the dataset. The first argument is a property list we're working on. The second argument is the name of the filter to be added. You can choose one of the following:

**H5Z\_FILTER\_DEFLATE** This is the `gzip` filter, which we have installed implicitly in the other program by calling `H5Pset_deflate`.

**H5Z\_FILTER\_SHUFFLE** This is a data shuffling filter. It may help compress data more effectively.

**H5Z\_FILTER\_FLETCHER32** This is the checksum filter we install in this program.

**H5Z\_FILTER\_SZIP** This is a yet another data compression filter that uses the SZIP algorithm.

The third argument is a bit-vector, i.e., an integer, that can be used to specify additional properties of the filter. In the case of this program it is simply 0.

The last argument is an array of integers that can be used to pass additional values to the filter and the next to last argument is the length of this array. We don't have such an array in this program, so we simply put NULL in its place.

In summary our call to `H5Pset_filter` looks as follows:

```
status = H5Pset_filter (property_id, H5Z_FILTER_FLETCHER32, 0, 0, NULL);
```

The second new function call, `H5Pget_edc_check`, is used in the reading part of the program in order to find if checksum has been enabled on the dataset. This function is very easy to use. It takes only one argument, the property list itself, and returns a logical value, TRUE for checksum enable and FALSE for checksum disabled.

But also observe another difference. When the property list itself is created, it is created against the `H5P_DATASET_XFER` constant – which represents the default for the raw data transfer property list.

This is what this part of the code looks like:

```
dxpl = H5Pcreate (H5P_DATASET_XFER);
edc = H5Pget_edc_check (dxpl);
if (edc == 1) printf ("Checksum is enabled\n");
if (edc == 0) printf ("Checksum is disabled\n");
```

## 7.5 Hyperslab Selection

We have already encountered the hyperslab selection procedure, i.e., a procedure that lets you read or write a portion of a dataset, in section 7.4.4 that talked about *extendible* datasets. In this section we are going to study hyperslab selection in more detail, the more so as hyperslab selection is crucial to MPI-IO writes and reads, because it is through this mechanism that processes get to stay away from each other's territory. In other words, hyperslab selection plays a role that is similar to MPI-IO file views.

Our first example is going to be sequential though, and here we will exercise hyperslab selection both on the file and on the memory side, because hyperslab selection can be used also to select a portion of a memory buffer from which to read or to which to write.

But then we'll switch to MPI-IO/HDF5 dataset manipulations and there we'll illustrate various dataset partitioning mechanisms whereupon each MPI process is going to select its own hyperslab within the dataset that resides on an HDF5 file.

### 7.5.1 Sequential Example

The program we are going to discuss in this section writes the following array on an HDF5 dataset:

```

0 1 2 3 4 5
1 2 3 4 5 6
2 3 4 5 6 7
3 4 5 6 7 8
4 5 6 7 8 9

```

Then the dataset and the file get closed and then we open them again for reading. This time, though, we create a  $7 \times 7 \times 3$  *memory* dataspace, which is filled with zeros initially. The following shows one of the 2-dimensional slices of this data space, e.g., the front slice:

```

0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0

```

Then we are going to read only a portion of the *file* dataspace onto this 3-dimensional cube of zeros – such a portion is called the *hyperslab* – and, to make things more fancy, we read the data onto a specific location in the memory data space, so that the front slice of the memory data space eventually looks as follows:

```

0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
3 4 5 6 0 0 0
4 5 6 7 0 0 0
5 6 7 8 0 0 0
0 0 0 0 0 0 0

```

You can now appreciate how this is going to be useful in dividing the file dataspace amongst MPI processes, both for writing and for reading.

Here is the example program taken from the NCSA HDF5 Tutorial:

```

/*****

This example shows how to write and read a hyperslab. It
is derived from the h5_read.c and h5_write.c examples in
the "Introduction to HDF5".

*****/

#include "hdf5.h"

#define FILE      "sds.h5"

```

```

#define DATASETNAME "IntArray"
#define NX_SUB 3 /* hyperslab dimensions */
#define NY_SUB 4
#define NX 7 /* output buffer dimensions */
#define NY 7
#define NZ 3
#define RANK 2
#define RANK_OUT 3

#define X 5 /* dataset dimensions */
#define Y 6

int
main (void)
{
    hsize_t dimsf[2]; /* dataset dimensions */
    int data[X][Y]; /* data to write */

    /*
     * Data and output buffer initialization.
     */
    hid_t file, dataset; /* handles */
    hid_t dataspace;
    hid_t memspace;
    hsize_t dimsm[3]; /* memory space dimensions */
    hsize_t dims_out[2]; /* dataset dimensions */
    herr_t status;

    int data_out[NX][NY][NZ ]; /* output buffer */

    hsize_t count[2]; /* size of the hyperslab in the file */
    hssize_t offset[2]; /* hyperslab offset in the file */
    hsize_t count_out[3]; /* size of the hyperslab in memory */
    hssize_t offset_out[3]; /* hyperslab offset in memory */
    int i, j, k, status_n, rank;

    /******
     This writes data to the HDF5 file.
     *****/

    /*
     * Data and output buffer initialization.
     */
    for (j = 0; j < X; j++) {
for (i = 0; i < Y; i++)
    data[j][i] = i + j;
    }
    /*
     * 0 1 2 3 4 5
     * 1 2 3 4 5 6
     * 2 3 4 5 6 7
     * 3 4 5 6 7 8
     * 4 5 6 7 8 9
     */
}

```

```

/*
 * Create a new file using H5F_ACC_TRUNC access,
 * the default file creation properties, and the default file
 * access properties.
 */
file = H5Fcreate (FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

/*
 * Describe the size of the array and create the data space for fixed
 * size dataset.
 */
dimsf[0] = X;
dimsf[1] = Y;
dataspace = H5Screate_simple (RANK, dimsf, NULL);

/*
 * Create a new dataset within the file using defined dataspace and
 * default dataset creation properties.
 */
dataset = H5Dcreate (file, DATASETNAME, H5T_STD_I32BE, dataspace,
                    H5P_DEFAULT);

/*
 * Write the data to the dataset using default transfer properties.
 */
status = H5Dwrite (dataset, H5T_NATIVE_INT, H5S_ALL, H5S_ALL,
                  H5P_DEFAULT, data);

/*
 * Close/release resources.
 */
H5Sclose (dataspace);
H5Dclose (dataset);
H5Fclose (file);

/*****

This reads the hyperslab from the sds.h5 file just
created, into a 2-dimensional plane of the 3-dimensional
array.

*****/

    for (j = 0; j < NX; j++) {
for (i = 0; i < NY; i++) {
    for (k = 0; k < NZ ; k++)
data_out[j][i][k] = 0;
}
}

/*
 * Open the file and the dataset.
 */
file = H5Fopen (FILE, H5F_ACC_RDONLY, H5P_DEFAULT);
dataset = H5Dopen (file, DATASETNAME);

```



```

dataspace = H5Dget_space (dataset); /* dataspace handle */
rank      = H5Sget_simple_extent_ndims (dataspace);
status_n  = H5Sget_simple_extent_dims (dataspace, dims_out, NULL);
printf("\nRank: %d\nDimensions: %lu x %lu \n", rank,
(unsigned long)(dims_out[0]), (unsigned long)(dims_out[1]));

/*
 * Define hyperslab in the dataset.
 */
offset[0] = 1;
offset[1] = 2;
count[0]  = NX_SUB;
count[1]  = NY_SUB;
status = H5Sselect_hyperslab (dataspace, H5S_SELECT_SET, offset, NULL,
                             count, NULL);

/*
 * Define the memory dataspace.
 */
dimsm[0] = NX;
dimsm[1] = NY;
dimsm[2] = NZ;
memspace = H5Screate_simple (RANK_OUT, dimsm, NULL);

/*
 * Define memory hyperslab.
 */
offset_out[0] = 3;
offset_out[1] = 0;
offset_out[2] = 0;
count_out[0]  = NX_SUB;
count_out[1]  = NY_SUB;
count_out[2]  = 1;
status = H5Sselect_hyperslab (memspace, H5S_SELECT_SET, offset_out, NULL,
                             count_out, NULL);

/*
 * Read data from hyperslab in the file into the hyperslab in
 * memory and display.
 */
status = H5Dread (dataset, H5T_NATIVE_INT, memspace, dataspace,
                 H5P_DEFAULT, data_out);
printf ("Data:\n ");
for (j = 0; j < NX; j++) {
for (i = 0; i < NY; i++) printf("%d ", data_out[j][i][0]);
printf("\n ");
}
printf("\n");
/*
 * 0 0 0 0 0 0 0
 * 0 0 0 0 0 0 0
 * 0 0 0 0 0 0 0
 * 3 4 5 6 0 0 0
 * 4 5 6 7 0 0 0
 * 5 6 7 8 0 0 0
 * 0 0 0 0 0 0 0
 */

```

```

/*
 * Close and release resources.
 */
H5Dclose (dataset);
H5Sclose (dataspace);
H5Sclose (memspace);
H5Fclose (file);
}

```

Here is how to compile, link and run the program:

```

gustav@bh1 $ h5cc -o h5_hyperslab h5_hyperslab.c
gustav@bh1 $ ./h5_hyperslab

```

```

Rank: 2
Dimensions: 5 x 6
Data:
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
3 4 5 6 0 0
4 5 6 7 0 0
5 6 7 8 0 0
0 0 0 0 0 0

```

```

gustav@bh1 $

```

Now let us discuss the program in detail.

The program begins with initialization of the data that is going to be written on the file:

```

#define X      5                /* dataset dimensions */
#define Y      6
...
int          data[X][Y];
...
for (j = 0; j < X; j++) {
    for (i = 0; i < Y; i++)
        data[j][i] = i + j;
}

```

which should generate a  $5 \times 6$  array of integers:

```

0 1 2 ...
1 2 3 ...
2 3 4 ...
...

```

The HDF5 data file is then created, we create a simple dataspace, then create a dataset associated with the dataspace, and then write *all* the data on the dataset filling it entirely. Then we close the dataspace, the dataset and the file:

```

file = H5Fcreate (FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
dimsf[0] = X; dimsf[1] = Y;
dataspace = H5Screate_simple (RANK, dimsf, NULL);
dataset = H5Dcreate (file, DATASETNAME, H5T_STD_I32BE, dataspace,
                    H5P_DEFAULT);

```

```

    status = H5Dwrite (dataset, H5T_NATIVE_INT, H5S_ALL, H5S_ALL,
                      H5P_DEFAULT, data);
    H5Sclose (dataspace);
    H5Dclose (dataset);
    H5Fclose (file);

```

So far there has been nothing new here. The `H5Dwrite` writes the whole lot and *onto* the whole lot: `H5S_ALL`, `H5S_ALL`.

The fancy stuff begins in the data reading part that follows.

We begin this part of the program by creating and initializing with zeros a new 3-dimensional array:

```

#define NX 7                                /* output buffer dimensions */
#define NY 7
#define NZ 3
...
    int      data_out[NX][NY][NZ ];
...

    for (j = 0; j < NX; j++) {
        for (i = 0; i < NY; i++) {
            for (k = 0; k < NZ ; k++)
                data_out[j][i][k] = 0;
        }
    }

```

Then we open the file again, but this time for reading only and open the dataset:

```

#define FILE      "sds.h5"
#define DATASETNAME "IntArray"
...
    file = H5Fopen (FILE, H5F_ACC_RDONLY, H5P_DEFAULT);
    dataset = H5Dopen (file, DATASETNAME);

```

The following three calls extract information about the dataspace associated with the dataset:

```

    dataspace = H5Dget_space (dataset); /* dataspace handle */
    rank      = H5Sget_simple_extent_ndims (dataspace);
    status_n  = H5Sget_simple_extent_dims (dataspace, dims_out, NULL);
    printf("\nRank: %d\nDimensions: %lu x %lu \n", rank,
          (unsigned long)(dims_out[0]), (unsigned long)(dims_out[1]));

```

and once we got its dimensions we print this information on standard output. We've seen this operation several times before already. The resulting output is:

```

Rank: 2
Dimensions: 5 x 6

```

Now we are going to *narrow* this dataspace by selecting a hyperslab, i.e., a  $3 \times 4$  submatrix.

```

#define NX_SUB 3                            /* hyperslab dimensions */
#define NY_SUB 4
...
    offset[0] = 1;
    offset[1] = 2;
    count[0]  = NX_SUB;
    count[1]  = NY_SUB;
    status = H5Sselect_hyperslab (dataspace, H5S_SELECT_SET, offset, NULL,
                                  count, NULL);

```

Function `H5Sselect_hyperslab` is going to *narrow* the dataspace pointed to by its first argument (the identifier) focusing on `count[0]` columns and `count[1]` rows starting from a corner whose coordinates are `offset[0]` and `offset[1]`.

The second argument is the *selection operator*. The following selection operators can be used here:

**H5S\_SELECT\_SET** Replaces the existing selection with the parameters from this call. So this is what I meant when I said that we are going to *narrow* our original dataspace.

**H5S\_SELECT\_OR** Adds the new selection to the existing selection. This is the *OR* operator of the set theory.

**H5S\_SELECT\_AND** Retains only the overlapping portions of the new selection and the existing selection. This is the *AND* operator of the set theory.

**H5S\_SELECT\_XOR** Retains only the elements that are members of the new selection or the existing selection, excluding elements that are members of both selections. This is the exclusive *OR* of the set theory.

**H5S\_SELECT\_NOTB** Retains only elements of the existing selection that are not in the new selection. This is the *NOT* operator of the set theory.

**H5S\_SELECT\_NOTA** Retains only elements of the new selection that are not in the existing selection.

The two `NULL` parameters in this call correspond to the stride and the size of the block to be picked up. If the stride is `NULL` a contiguous hyperslab is selected. If the block size is `NULL` it defaults to one elementary data item.

So this is how we have constructed the hyperslab of the dataset on the file. This is what we are going to read. Now, where are we going to put it? To answer this question we define first the memory dataspace and then we take a hyperslab out of it:

```
#define NX 7
#define NY 7
#define NZ 3
#define NX_SUB 3
#define NY_SUB 4
#define RANK_OUT 3
...
    dimsm[0] = NX;
    dimsm[1] = NY;
    dimsm[2] = NZ;
    memspace = H5Screate_simple (RANK_OUT, dimsm, NULL);

    offset_out[0] = 3;
    offset_out[1] = 0;
    offset_out[2] = 0;
    count_out[0] = NX_SUB;
    count_out[1] = NY_SUB;
    count_out[2] = 1;
    status = H5Sselect_hyperslab (memspace, H5S_SELECT_SET, offset_out, NULL,
                                count_out, NULL);
```

The space onto which we are going to write the data will begin from row number 3 (remember that in C rows are counted from 0), column zero and plane zero too – so we are going to write on the front plane of the brick. The extents are going to be 3 rows and 4 columns, and we are not going to go beyond the front plane.

Now we are ready to read the data:

```
status = H5Dread (dataset, H5T_NATIVE_INT, memspace, dataspace,
                 H5P_DEFAULT, data_out);
```

The data is read from the already *narrowed* dataspace on the file into the just *narrowed* dataspace, here called `memspace`, in `data_out`.

The following print statements show the final effect of this operation:

```
printf ("Data:\n ");
for (j = 0; j < NX; j++) {
    for (i = 0; i < NY; i++) printf("%d ", data_out[j][i][0]);
    printf("\n ");
}
printf("\n");
```

The job done, we close the dataset, the dataspace, the memory dataspace, and finally the file:

```
H5Dclose (dataset);
H5Sclose (dataspace);
H5Sclose (memspace);
H5Fclose (file);
```

Now, at long last we can attempt an MPI-IO data transfer.

## 7.5.2 Partitioning MPI-IO/HDF5 Datasets

### Contiguous Hyperslabs

The following example code, also taken from the NCSA HDF5 Tutorial introduces almost nothing new, but when all that you've learnt so far gets combined together, you end up, miraculously, with a parallel write onto an HDF5 dataset that sits in an HDF5 file.

This time *before* I show you the program, I want to show you what it does first.

The program lives in this directory

```
gustav@bh1 $ pwd
/N/B/gustav/src/I590/HDF5/mpi-io
gustav@bh1 $ ls
hyperslab_by_row.c
gustav@bh1
```

and is compiled and linked with the `h5cc` wrapper, which knows all about MPI and MPI-IO too.

```
gustav@bh1 $ h5cc -o hyperslab_by_row hyperslab_by_row.c
gustav@bh1 $
```

I install the program in my `~/bin` so that `mpiexec` will find it. Then I go my GPFS directory.

```
gustav@bh1 $ mv hyperslab_by_row ~/bin
gustav@bh1 $ cd /N/gpfs/gustav
gustav@bh1 $ ls
t_mpio_1wMr test tests
gustav@bh1 $
```

MPD is running already on 26 nodes, but I'm going to use 4 only for this run:

```
gustav@bh1 $ mpdtrace | wc
      26      26     129
gustav@bh1 $ mpiexec -n 4 hyperslab_by_row
gustav@bh1 $
```

The job created an HDF5 file `SDS_row.h5` in this directory, and I am going to inspect it with `h5dump`.

```
gustav@bh1 $ ls
SDS_row.h5 t_mpio_1wMr test tests
gustav@bh1 $ h5dump SDS_row.h5
HDF5 "SDS_row.h5" {
  GROUP "/" {
    DATASET "IntArray" {
      DATATYPE  H5T_STD_I32LE
      DATASPACE  SIMPLE { ( 8, 5 ) / ( 8, 5 ) }
      DATA {
        10, 10, 10, 10, 10,
        10, 10, 10, 10, 10,
        11, 11, 11, 11, 11,
        11, 11, 11, 11, 11,
        12, 12, 12, 12, 12,
        12, 12, 12, 12, 12,
        13, 13, 13, 13, 13,
        13, 13, 13, 13, 13
      }
    }
  }
}
gustav@bh1 $
```

You will see when we get to analyze the program that process 0 wrote:

```
10, 10, 10, 10, 10,
10, 10, 10, 10, 10,
```

process 1 wrote

```
11, 11, 11, 11, 11,
11, 11, 11, 11, 11,
```

process 2 wrote

```
12, 12, 12, 12, 12,
12, 12, 12, 12, 12,
```

and process 3 wrote:

```
13, 13, 13, 13, 13,
13, 13, 13, 13, 13
```

The writes were collective, i.e., HDF5 used `MPI_File_write_all` to write the data on this file, and presumably, since the file lives on GPFS, the writes occurred in parallel.

So now let's see how this is done. Here is the program:

```

/*
 * This example writes data to the HDF5 file by rows.
 * Number of processes is assumed to be 1 or multiples of 2 (up to 8)
 */

#include "hdf5.h"

#define H5FILE_NAME      "SDS_row.h5"
#define DATASETNAME     "IntArray"
#define NX              8          /* dataset dimensions */
#define NY              5
#define RANK            2

int
main (int argc, char **argv)
{
    /*
     * HDF5 APIs definitions
     */
    hid_t      file_id, dset_id;          /* file and dataset identifiers */
    hid_t      filespace, memspace;      /* file and memory dataspace identifiers */
    hsize_t    dims[2];                  /* dataset dimensions */
    int        *data;                   /* pointer to data buffer to write */
    hsize_t    count[2];                 /* hyperslab selection parameters */
    hssize_t   offset[2];
    hid_t      plist_id;                 /* property list identifier */
    int        i;
    herr_t     status;

    /*
     * MPI variables
     */
    int mpi_size, mpi_rank;
    MPI_Comm comm = MPI_COMM_WORLD;
    MPI_Info info = MPI_INFO_NULL;

    /*
     * Initialize MPI
     */
    MPI_Init(&argc, &argv);
    MPI_Comm_size(comm, &mpi_size);
    MPI_Comm_rank(comm, &mpi_rank);

    /*
     * Set up file access property list with parallel I/O access
     */
    plist_id = H5Pcreate(H5P_FILE_ACCESS);
    H5Pset_fapl_mpio(plist_id, comm, info);

    /*
     * Create a new file collectively and release property list identifier.
     */

```

```

file_id = H5Fcreate(H5FILE_NAME, H5F_ACC_TRUNC, H5P_DEFAULT, plist_id);
H5Pclose(plist_id);

/*
 * Create the dataspace for the dataset.
 */
dimsf[0] = NX;
dimsf[1] = NY;
filespace = H5Screate_simple(RANK, dimsf, NULL);

/*
 * Create the dataset with default properties and close filespace.
 */
dset_id = H5Dcreate(file_id, DATASETNAME, H5T_NATIVE_INT, filespace,
H5P_DEFAULT);
H5Sclose(filespace);

/*
 * Each process defines dataset in memory and writes it to the hyperslab
 * in the file.
 */
count[0] = dimsf[0]/mpi_size;
count[1] = dimsf[1];
offset[0] = mpi_rank * count[0];
offset[1] = 0;
memspace = H5Screate_simple(RANK, count, NULL);

/*
 * Select hyperslab in the file.
 */
filespace = H5Dget_space(dset_id);
H5Sselect_hyperslab(filespace, H5S_SELECT_SET, offset, NULL, count, NULL);

/*
 * Initialize data buffer
 */
data = (int *) malloc(sizeof(int)*count[0]*count[1]);
for (i=0; i < count[0]*count[1]; i++) {
    data[i] = mpi_rank + 10;
}

/*
 * Create property list for collective dataset write.
 */
plist_id = H5Pcreate(H5P_DATASET_XFER);
H5Pset_dxpl_mpio(plist_id, H5FD_MPIO_COLLECTIVE);

status = H5Dwrite(dset_id, H5T_NATIVE_INT, memspace, filespace,
    plist_id, data);
free(data);

/*
 * Close/release resources.
 */
H5Dclose(dset_id);
H5Sclose(filespace);

```



```

    H5Sclose(memspace);
    H5Pclose(plist_id);
    H5Fclose(file_id);

    MPI_Finalize();

    return 0;
}

```

The program begins with already traditional MPI incantations:

```

MPI_Init(&argc, &argv);
MPI_Comm_size(comm, &mpi_size);
MPI_Comm_rank(comm, &mpi_rank);

```

Then we immediately switch to HDF5. We create a file access property list that contains MPI information and we create the file itself:

```

plist_id = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_fapl_mpio(plist_id, comm, info);
file_id = H5Fcreate(H5FILE_NAME, H5F_ACC_TRUNC, H5P_DEFAULT, plist_id);
H5Pclose(plist_id);

```

Now we create the dataspace and the dataset:

```

#define H5FILE_NAME      "SDS_row.h5"
#define DATASETNAME     "IntArray"
#define NX              8
#define NY              5
#define RANK            2
...
    dimsf[0] = NX;
    dimsf[1] = NY;
    filespace = H5Screate_simple(RANK, dimsf, NULL);
    dset_id = H5Dcreate(file_id, DATASETNAME, H5T_NATIVE_INT, filespace,
H5P_DEFAULT);
    H5Sclose(filespace);

```

The dataset is going to have 8 rows and 5 columns.

Now each process defines its own memory dataset and memory dataspace for its own data and also selects a hyperslab on the HDF5 dataset in the file, on which it is going to write:

```

count[0] = dimsf[0]/mpi_size;
count[1] = dimsf[1];
offset[0] = mpi_rank * count[0];
offset[1] = 0;
memspace = H5Screate_simple(RANK, count, NULL);
filespace = H5Dget_space(dset_id);
H5Sselect_hyperslab(filespace, H5S_SELECT_SET, offset, NULL, count, NULL);

```

Now each process initializes its own data:

```

data = (int *) malloc(sizeof(int)*count[0]*count[1]);
for (i=0; i < count[0]*count[1]; i++) {
    data[i] = mpi_rank + 10;
}

```

and then we get back to HDF5. Each process creates a data transfer property list, in which we request that the following write should be collective:

```
plist_id = H5Pcreate(H5P_DATASET_XFER);
H5Pset_dxpl_mpio(plist_id, H5FD_MPIO_COLLECTIVE);
```

Function `H5Pset_dxpl_mpio` is the only new function in this program. Apart from `H5FD_MPIO_COLLECTIVE` you can also request `H5FD_MPIO_INDEPENDENT`, which is a default actually.

Now we are ready to write the data:

```
status = H5Dwrite(dset_id, H5T_NATIVE_INT, memspace, filespace,
plist_id, data);
```

Observe that `filespace` corresponds to a different hyperslab for each process, so that they don't step on each other's toes.

After the write returns, we `free` the array `data`, which was malloced before, then close the dataset, the filespace, the memory space, the property list and finally the file itself.

And, at the end, we call `MPI_Finalize`, because it is an MPI program after all.

So this is how it works. HDF5 implements parallel IO on the dataset level and file partitioning is accomplished by the means of hyperslab selection.

### Striding Data

In this section we are going to use a stride and a block while selecting the hyperslab for partitioning of a fileset. The effect will be that the two processes that constitute the MPI pool in the example that follows will divide the HDF5 dataset into interleaving columns, i.e., process 1 is going to write column 1 while process 2 will write column 2, then process 1 will write column 3 while process 2 will write column 4 and so on. The result will be a dataset that looks as follows:

```
1 2 10 20 100 200
1 2 10 20 100 200
1 2 10 20 100 200
1 2 10 20 100 200
1 2 10 20 100 200
1 2 10 20 100 200
1 2 10 20 100 200
1 2 10 20 100 200
1 2 10 20 100 200
```

where the first digit of every number marks the process that has written the number.

The program must be run on two processes.

Here is how to compile and run it:

```
gustav@bh1 $ pwd
/N/B/gustav/src/I590/HDF5/mpi-io
gustav@bh1 $ h5cc -o hyperslab_by_col hyperslab_by_col.c
gustav@bh1 $ mv hyperslab_by_col ~/bin
gustav@bh1 $ mpdboot
gustav@bh1 $ mpdtrace | wc
      20      20      99
```

```

gustav@bh1 $ cd /N/gpfs/gustav
gustav@bh1 $ ls
SDS_row.h5  t_mpio_1wMr  test  tests
gustav@bh1 $ mpiexec -n 2 hyperslab_by_col
gustav@bh1 $ ls
SDS_col.h5  SDS_row.h5  t_mpio_1wMr  test  tests
gustav@bh1 $ h5dump SDS_col.h5
HDF5 "SDS_col.h5" {
GROUP "/" {
  DATASET "IntArray" {
    DATATYPE  H5T_STD_I32LE
    DATASPACE  SIMPLE { ( 8, 6 ) / ( 8, 6 ) }
    DATA {
      1, 2, 10, 20, 100, 200,
      1, 2, 10, 20, 100, 200,
      1, 2, 10, 20, 100, 200,
      1, 2, 10, 20, 100, 200,
      1, 2, 10, 20, 100, 200,
      1, 2, 10, 20, 100, 200,
      1, 2, 10, 20, 100, 200,
      1, 2, 10, 20, 100, 200,
      1, 2, 10, 20, 100, 200
    }
  }
}
}
}
gustav@bh1 $ mpdallexit
gustav@bh1 $

```

Now, here is the listing of the program, taken from the NCSA HDF5 Tutorial, and you'll find the discussion of the code further down.

```

/*
 * This example writes data to the HDF5 file by columns.
 * Number of processes is assumed to be 2.
 */

#include "hdf5.h"

#define H5FILE_NAME      "SDS_col.h5"
#define DATASETNAME     "IntArray"
#define NX              8          /* dataset dimensions */
#define NY              6
#define RANK            2

int
main (int argc, char **argv)
{
  /*
   * HDF5 APIs definitions
   */
  hid_t      file_id, dset_id;          /* file and dataset identifiers */
  hid_t      filespace, memspace;      /* file and memory dataspace identifiers */
  hsize_t    dims[2];                  /* dataset dimensions */
  hsize_t    dimsm[2];                 /* dataset dimensions */
  int        *data;                   /* pointer to data buffer to write */
  hsize_t    count[2];                 /* hyperslab selection parameters */
  hsize_t    stride[2];

```

```

hsize_t block[2];
hssize_t offset[2];
hid_t plist_id;          /* property list identifier */
int      i, j, k;
herr_t status;

/*
 * MPI variables
 */
int mpi_size, mpi_rank;
MPI_Comm comm = MPI_COMM_WORLD;
MPI_Info info = MPI_INFO_NULL;

/*
 * Initialize MPI
 */
MPI_Init(&argc, &argv);
MPI_Comm_size(comm, &mpi_size);
MPI_Comm_rank(comm, &mpi_rank);
/*
 * Exit if number of processes is not 2
 */
if (mpi_size != 2) {
    printf("This example to set up to use only 2 processes \n");
    printf("Quitting...\n");
    return 0;
}

/*
 * Set up file access property list with parallel I/O access
 */
plist_id = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_fapl_mpio(plist_id, comm, info);

/*
 * Create a new file collectively and release property list identifier.
 */
file_id = H5Fcreate(H5FILE_NAME, H5F_ACC_TRUNC, H5P_DEFAULT, plist_id);
H5Pclose(plist_id);

/*
 * Create the dataspace for the dataset.
 */
dimsf[0] = NX;
dimsf[1] = NY;
dimsm[0] = NX;
dimsm[1] = NY/2;
filespace = H5Screate_simple(RANK, dimsf, NULL);
memspace = H5Screate_simple(RANK, dimsm, NULL);

/*
 * Create the dataset with default properties and close filespace.
 */
dset_id = H5Dcreate(file_id, DATASETNAME, H5T_NATIVE_INT, filespace,
H5P_DEFAULT);
H5Sclose(filespace);

```

```

/*
 * Each process defines dataset in memory and writes it to the hyperslab
 * in the file.
 */
count[0] = 1;
count[1] = dimsm[1];
offset[0] = 0;
offset[1] = mpi_rank;
stride[0] = 1;
stride[1] = 2;
block[0] = dimsf[0];
block[1] = 1;

/*
 * Select hyperslab in the file.
 */
fileSpace = H5Dget_space(dset_id);
H5Sselect_hyperslab(fileSpace, H5S_SELECT_SET, offset, stride, count, block);

/*
 * Initialize data buffer
 */
data = (int *) malloc(sizeof(int)*dimsm[0]*dimsm[1]);
for (i=0; i < dimsm[0]*dimsm[1]; i=i+dimsm[1]) {
    k = 1;
    for (j=0; j < dimsm[1]; j++) {
        data[i + j] = (mpi_rank + 1) * k ;
        k = k * 10;
    }
}

/*
 * Create property list for collective dataset write.
 */
plist_id = H5Pcreate(H5P_DATASET_XFER);
H5Pset_dxpl_mpio(plist_id, H5FD_MPIO_COLLECTIVE);

status = H5Dwrite(dset_id, H5T_NATIVE_INT, memspace, fileSpace,
    plist_id, data);
free(data);

/*
 * Close/release resources.
 */
H5Dclose(dset_id);
H5Sclose(fileSpace);
H5Sclose(memspace);
H5Pclose(plist_id);
H5Fclose(file_id);

MPI_Finalize();

return 0;
}

```

This MPI/HDF5 code begins the same way the previous code has: we initialize MPI, find about the size of the communicator and every process then finds

its rank number. But then we check if the program is running on a number of processes that is different than 2 and if it is, we quit:

```

if (mpi_size != 2) {
    printf("This example to set up to use only 2 processes \n");
    printf("Quitting...\n");
    return 0;
}

```

Observe the dirty, UNIX-like, exit through "return" instead of `MPI_Abort`, or a jump to `MPI_Finalize` at the end of the program. This is not nice, but, clearly, the authors didn't care. Such an exit is bound to trigger MPD error messages. But ... let's plough on.

After this we create a property list for the MPI file access:

```

plist_id = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_fapl_mpio(plist_id, comm, info);

```

and then we create the file itself, whereupon the property list gets closed, because it is no longer needed:

```

#define H5FILE_NAME    "SDS_col.h5"
...
file_id = H5Fcreate(H5FILE_NAME, H5F_ACC_TRUNC, H5P_DEFAULT, plist_id);
H5Pclose(plist_id);

```

Now we are going to create two data spaces: one for the dataset on the file and one for the memory. Because we are going to split the dataset amongst the two processes of the pool, the memory data space for each process is going to be half of the dataset data space:

```

#define NX      8
#define NY      6
#define RANK    2
...
hsize_t      dimsf[2];
hsize_t      dimsm[2];
...
dimsf[0] = NX;
dimsf[1] = NY;
dimsm[0] = NX;
dimsm[1] = NY/2;
filespace = H5Screate_simple(RANK, dimsf, NULL);
memspace   = H5Screate_simple(RANK, dimsm, NULL);

```

Here `NX` is the number of rows and `NY` is the number of columns.

Now we can create the dataset and having done so, we're going to close the `filespace`, because it's no longer needed:

```

#define DATASETNAME "IntArray"
...
dset_id = H5Dcreate(file_id, DATASETNAME, H5T_NATIVE_INT, filespace,
H5P_DEFAULT);
H5Sclose(filespace);

```

At this stage, in preparation for writing, we are going to define our hyperslabs – these are going to be different for each process. The procedure is as follows: first we obtain the whole dataspace by interrogating the dataset and then we *modify* the dataspace by calling `H5Sselect_hyperslab` with the `H5S_SELECT_SET` argument. Here is the whole call:

```

count[0] = 1;
count[1] = dimsm[1]; /* i.e., count[1] = NY/2 = 3 */
offset[0] = 0;
offset[1] = mpi_rank; /* i.e., offset[1] = 0 or 1 */
stride[0] = 1;
stride[1] = 2;
block[0] = dimsf[0]; /* i.e., block[0] = NX = 8 */
block[1] = 1;

fileSpace = H5Dget_space(dset_id);
H5Sselect_hyperslab(fileSpace, H5S_SELECT_SET, offset, stride, count, block);

```

Let us read this definition aloud.

Consider process of rank 0. For this process `offset[0] = 0` and `offset[1] = 0` too. So for this process the hyperslab begins at the (0,0) location in the dataset, which corresponds to the beginning of the first column. But for process 1 `offset[0] = 0` and `offset[1] = 1`, so for this process the hyperslab begins at the (0,1) location in the dataset, which corresponds to the beginning of the second column.

For both processes `count[0] = 1`, `stride[0] = 1` – this corresponds to rows – and `count[1] = 3` and `stride[1] = 2` – this corresponds to columns – which means that each process is going to select every second block of data to the right three times, and each block is going to be 8 rows times 1 column, because `block[0] = 8` and `block[1] = 1`.

In other words, process 0 will select columns 1, 3, and 5 and process 1 will select columns 2, 4, and 6.

Now each process writes the following data to its memory space:

```

data = (int *) malloc(sizeof(int)*dimsm[0]*dimsm[1]);
for (i=0; i < dimsm[0]*dimsm[1]; i=i+dimsm[1]) {
    k = 1;
    for (j=0; j < dimsm[1]; j++) {
        data[i + j] = (mpi_rank + 1) * k ;
        k = k * 10;
    }
}

```

Observe that only the `k` index is used in constructing the data, whereas `i` and `j` indexes are used in placing it. Also observe that `k` is reset to 1 every time `j` reaches `dimsm[1] = 3`. And so, for the process of rank 0 we are going to have:

```
data[] = 1, 10, 100, 1, 10, 100, ... 1, 10, 100
```

and for the process of rank 1 we are going to have:

```
data[] = 2, 20, 200, 2, 20, 200, ... 2, 20, 200
```

Now remember that in C matrices are stored in the row-contiguous fashion, which means that when this data is going to be mapped onto our memory space, which is a matrix  $8 \times 3$ , we'll get:

```

1, 10, 100,
1, 10, 100,
...
1, 10, 100

```

and

```
2, 20, 200,
2, 20, 200,
...
2, 20, 200
```

Finally, we are ready to write the data. We create the property list for collective MPI-IO write (using `MPI_File_write_all`) and write the data by interpreting its memory layout according to `memspace` and its HDF5 file dataset layout according to `filespace`:

```
plist_id = H5Pcreate(H5P_DATASET_XFER);
H5Pset_dxpl_mpio(plist_id, H5FD_MPIO_COLLECTIVE);
status = H5Dwrite(dset_id, H5T_NATIVE_INT, memspace, filespace,
                plist_id, data);
```

And this is it: we free `data`, then we close the dataset, `filespace`, `memspace`, the property list and finally the HDF5 file itself. Then the processes meet at `MPI_Finalize` and we exit the program with `return 0`.

### Scattering Data

Whereas in the previous example we have divided the dataset into columns assigning every second column to a process participating in the MPI pool (which was restricted to two processes only), in this example we are going to divide the dataset differently. The dataset is going to be of size  $8 \times 4$ , i.e., we'll have 8 rows and 4 columns and 4 processes in the pool. Each process is going to write its own rank number (plus 1) on the dataset in a different place, so that the final picture is going to look as follows:

```
1 3 1 3
2 4 2 4
1 3 1 3
2 4 2 4
1 3 1 3
2 4 2 4
1 3 1 3
2 4 2 4
```

This is called *scattering* the data, although the data isn't really scattered at random at all – it is written according to a pattern, but it is written in a highly non-contiguous (and therefore *inefficient*) manner. I can't think of a context where you would need to do something like this, but it's nice to know that you can.

Here is how to compile, link and run the program, and what comes out of it eventually:

```
gustav@bh1 $ pwd
/N/B/gustav/src/I590/HDF5/mpi-io
gustav@bh1 $ h5cc -o hyperslab_by_pattern hyperslab_by_pattern.c
gustav@bh1 $ mv hyperslab_by_pattern ~/bin
```



```

gustav@bh1 $ cd
gustav@bh1 $ mpdboot
gustav@bh1 $ mpdtrace | wc
      20      20      99
gustav@bh1 $ cd /N/gpfs/gustav
gustav@bh1 $ ls
SDS_col.h5  SDS_row.h5  t_mpio_1wMr  test  tests
gustav@bh1 $ mpiexec -n 4 hyperslab_by_pattern
gustav@bh1 $ ls
SDS_col.h5  SDS_pat.h5  SDS_row.h5  t_mpio_1wMr  test  tests
gustav@bh1 $ h5dump SDS_pat.h5
HDF5 "SDS_pat.h5" {
GROUP "/" {
  DATASET "IntArray" {
    DATATYPE  H5T_STD_I32LE
    DATASPACE  SIMPLE { ( 8, 4 ) / ( 8, 4 ) }
    DATA {
      1, 3, 1, 3,
      2, 4, 2, 4,
      1, 3, 1, 3,
      2, 4, 2, 4,
      1, 3, 1, 3,
      2, 4, 2, 4,
      1, 3, 1, 3,
      2, 4, 2, 4
    }
  }
}
}
gustav@bh1 $ mpdallexit
gustav@bh1 $

```

The listing of the program that comes from the NCSA HDF5 Tutorial is shown below.

```

/*
 * This example writes data to the HDF5 file following some pattern
 *
 *      - | - | .....
 *      * V * V .....
 *      - | - | .....
 *      * V * V .....
 *      .....
 * Number of processes is assumed to be 4.
 */

#include "hdf5.h"

#define H5FILE_NAME      "SDS_pat.h5"
#define DATASETNAME     "IntArray"
#define NX              8                /* dataset dimensions */
#define NY              4
#define RANK            2
#define RANK1          1

int
main (int argc, char **argv)
{
    /*

```

```

* HDF5 APIs definitions
*/
hid_t      file_id, dset_id;          /* file and dataset identifiers */
hid_t      filespace, memspace;      /* file and memory dataspace identifiers */
hsize_t    dimsf[2];                 /* dataset dimensions */
hsize_t    dimsm[1];                 /* dataset dimensions */
int        *data;                    /* pointer to data buffer to write */
hsize_t    count[2];                 /* hyperslab selection parameters */
hsize_t    stride[2];
hssize_t   offset[2];
hid_t      plist_id;                 /* property list identifier */
int        i;
herr_t     status;

/*
* MPI variables
*/
int mpi_size, mpi_rank;
MPI_Comm comm = MPI_COMM_WORLD;
MPI_Info info = MPI_INFO_NULL;

/*
* Initialize MPI
*/
MPI_Init(&argc, &argv);
MPI_Comm_size(comm, &mpi_size);
MPI_Comm_rank(comm, &mpi_rank);
/*
* Exit if number of processes is not 4.
*/
if (mpi_size != 4) {
    printf("This example is set up to use 4 processes exactly\n");
    printf("Quitting...\n");
    return 0;
}

/*
* Set up file access property list with parallel I/O access
*/
plist_id = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_fapl_mpio(plist_id, comm, info);

/*
* Create a new file collectively and release property list identifier.
*/
file_id = H5Fcreate(H5FILE_NAME, H5F_ACC_TRUNC, H5P_DEFAULT, plist_id);
H5Pclose(plist_id);

/*
* Create the dataspace for the dataset.
*/
dimsf[0] = NX;
dimsf[1] = NY;
dimsm[0] = NX;
filespace = H5Screate_simple(RANK, dimsf, NULL);
memspace = H5Screate_simple(RANK1, dimsm, NULL);

```

```
/*
 * Create the dataset with default properties and close filespace.
 */
dset_id = H5Dcreate(file_id, DATASETNAME, H5T_NATIVE_INT, filespace,
H5P_DEFAULT);
H5Sclose(filespace);

/*
 * Each process defines dataset in memory and writes it to the hyperslab
 * in the file.
 */
count[0] = 4;
count[1] = 2;
stride[0] = 2;
stride[1] = 2;
if(mpi_rank == 0) {
    offset[0] = 0;
    offset[1] = 0;
}
if(mpi_rank == 1) {
    offset[0] = 1;
    offset[1] = 0;
}
if(mpi_rank == 2) {
    offset[0] = 0;
    offset[1] = 1;
}
if(mpi_rank == 3) {
    offset[0] = 1;
    offset[1] = 1;
}

/*
 * Select hyperslab in the file.
 */
filespace = H5Dget_space(dset_id);
status = H5Sselect_hyperslab(filespace, H5S_SELECT_SET, offset, stride, count, NULL);

/*
 * Initialize data buffer
 */
data = (int *) malloc(sizeof(int)*dimsm[0]);
for (i=0; i < (int)dimsm[0]; i++) {
    data[i] = mpi_rank + 1;
}

/*
 * Create property list for collective dataset write.
 */
plist_id = H5Pcreate(H5P_DATASET_XFER);
H5Pset_dxpl_mpio(plist_id, H5FD_MPIO_COLLECTIVE);

status = H5Dwrite(dset_id, H5T_NATIVE_INT, memspace, filespace,
    plist_id, data);
free(data);
```

```

    /*
     * Close/release resources.
     */
    H5Dclose(dset_id);
    H5Sclose(filespace);
    H5Sclose(memspace);
    H5Pclose(plist_id);
    H5Fclose(file_id);

    MPI_Finalize();

    return 0;
}

```

Let us analyze this program now. I'll skip the MPI opening, the abominable error handling, and then creation of the file itself (with an appropriate MPI-related property list), because it's all the same stuff as before, and we're going to halt at the point where the data spaces are created:

```

#define NX      8
#define NY      4
#define RANK    2
#define RANK1   1
...
    dimsf[0] = NX;
    dimsf[1] = NY;
    dimsm[0] = NX;
    filespace = H5Screate_simple(RANK, dimsf, NULL);
    memspace  = H5Screate_simple(RANK1, dimsm, NULL);

```

So, the `filespace` is going to be  $8 \times 4$  and the `memspace` is a one-dimensional array of length 8. Then we create the dataset on the HDF5 file and discard the `filespace`:

```

#define DATASETNAME "IntArray"
...
    dset_id = H5Dcreate(file_id, DATASETNAME, H5T_NATIVE_INT, filespace,
H5P_DEFAULT);
    H5Sclose(filespace);

```

Now we get down to selecting the hyperslab of the dataset. We initialize strides, counts and offsets:

```

    count[0] = 4;
    count[1] = 2;
    stride[0] = 2;
    stride[1] = 2;
    if(mpi_rank == 0) {
        offset[0] = 0;
        offset[1] = 0;
    }
    if(mpi_rank == 1) {
        offset[0] = 1;
        offset[1] = 0;
    }
    if(mpi_rank == 2) {
        offset[0] = 0;
        offset[1] = 1;
    }
}

```

```

if(mpi_rank == 3) {
    offset[0] = 1;
    offset[1] = 1;
}

```

Each process, as you see, is going to write  $4 \times 2$  blocks of data separated by a stride of 2 in each dimension. The block sizes will default to 1, because we are going to NULL in the block size slot. Process of rank 0 starts at the (0,0) location, process of rank 1 starts at the (1,0) location, process of rank 2 starts at the (0,1) location and process of rank 3 starts at the (1,1) location. With counts, strides and offsets defined, we call `H5Sselect_hyperslab`:

```

file_space = H5Dget_space(dset_id);
status = H5Sselect_hyperslab(file_space, H5S_SELECT_SET, offset, stride, count, NULL);

```

Each process is going to fill its memory dataspace simply with its own rank number (plus 1):

```

data = (int *) malloc(sizeof(int)*dimsm[0]);
for (i=0; i < (int)dimsm[0]; i++) {
    data[i] = mpi_rank + 1;
}

```

Now we are ready to write the data. We prep the property list for a collective write using `MPI_File_write_all` and write the data itself so that it gets mapped from memspace to file\_space:

```

plist_id = H5Pcreate(H5P_DATASET_XFER);
H5Pset_dxpl_mpio(plist_id, H5FD_MPIO_COLLECTIVE);
status = H5Dwrite(dset_id, H5T_NATIVE_INT, memspace, file_space,
    plist_id, data);

```

And this is it. We free `data`, then we close the dataset, the file\_space, the memory space, the property list and the file itself, and finally the processes meet on `MPI_Finalize` and exit.

### Chunking

In this last MPI-IO/HDF5 example we are going to write data from four processes, the data being much the same as in the previous example, i.e., the process rank number plus 1, on the  $8 \times 4$  dataset, but this time we'll write the data in blocks:

```

1 1 2 2
1 1 2 2
1 1 2 2
1 1 2 2
3 3 4 4
3 3 4 4
3 3 4 4
3 3 4 4

```

Observe that although at first glance this partitioning does not seem *very* different from what we had in our first MPI-IO/HDF5 example:

```

1 1 1 1
1 1 1 1
2 2 2 2
2 2 2 2
3 3 3 3
3 3 3 3
4 4 4 4
4 4 4 4

```

it is, in fact, very different. Because matrices are stored row-contiguous in C, in order to achieve the above pattern we can simply divide the dataset into four contiguous portions and assign each to a separate process. This is, in fact, the most efficient way of partitioning a dataset and you should stick to this, unless you really *have to* subject yourself to the kind of equilibristics we have been talking about in the last 3 sections (including this one).

So, the partition we are going to implement in the following example will not be contiguous.

Before I get to the program itself, let me show you how to compile, link and run the program first:

```

gustav@bh1 $ pwd
/N/B/gustav/src/I590/HDF5/mpi-io
gustav@bh1 $ h5cc -o hyperslab_by_chunk hyperslab_by_chunk.c
gustav@bh1 $ mv hyperslab_by_chunk ~/bin
gustav@bh1 $ cd
gustav@bh1 $ mpdboot

gustav@bh1 $
gustav@bh1 $ mpdtrace | wc
      20      20      99
gustav@bh1 $ cd /N/gpfs/gustav
gustav@bh1 $ ls
SDS_col.h5  SDS_pat.h5  SDS_row.h5  t_mpiio_1wMr  test  tests
gustav@bh1 $ mpiexec -n 4 hyperslab_by_chunk
gustav@bh1 $ ls
SDS_chnk.h5  SDS_col.h5  SDS_pat.h5  SDS_row.h5  t_mpiio_1wMr  test  tests
gustav@bh1 $ h5dump SDS_chnk.h5
HDF5 "SDS_chnk.h5" {
  GROUP "/" {
    DATASET "IntArray" {
      DATATYPE  H5T_STD_I32LE
      DATASPACE  SIMPLE { ( 8, 4 ) / ( 8, 4 ) }
      DATA {
        1, 1, 2, 2,
        1, 1, 2, 2,
        1, 1, 2, 2,
        1, 1, 2, 2,
        3, 3, 4, 4,
        3, 3, 4, 4,
        3, 3, 4, 4,
        3, 3, 4, 4
      }
    }
  }
}

```

```

    }
  }
}
}
gustav@bh1 $ mpdallexit
gustav@bh1 $

```

Here is the program, which I have taken from the NCSA HDF5 Tutorial.

```

/*
 * This example writes dataset sing chunking. Each process writes
 * exactly one chunk.
 *      - |
 *      * V
 * Number of processes is assumed to be 4.
 */

#include "hdf5.h"

#define H5FILE_NAME      "SDS_chnk.h5"
#define DATASETNAME     "IntArray"
#define NX              8          /* dataset dimensions */
#define NY              4
#define CH_NX           4          /* chunk dimensions */
#define CH_NY           2
#define RANK            2

int
main (int argc, char **argv)
{
    /*
     * HDF5 APIs definitions
     */
    hid_t      file_id, dset_id;          /* file and dataset identifiers */
    hid_t      filespace, memspace;       /* file and memory dataspace identifiers */
    hsize_t    dims[2];                   /* dataset dimensions */
    hsize_t    chunk_dims[2];            /* chunk dimensions */
    int        *data;                     /* pointer to data buffer to write */
    hsize_t    count[2];                  /* hyperslab selection parameters */
    hsize_t    stride[2];
    hsize_t    block[2];
    hssize_t   offset[2];
    hid_t      plist_id;                  /* property list identifier */
    int        i;
    herr_t     status;

    /*
     * MPI variables
     */
    int mpi_size, mpi_rank;
    MPI_Comm comm = MPI_COMM_WORLD;
    MPI_Info info = MPI_INFO_NULL;

    /*
     * Initialize MPI
     */
    MPI_Init(&argc, &argv);
    MPI_Comm_size(comm, &mpi_size);

```

```

MPI_Comm_rank(comm, &mpi_rank);
/*
 * Exit if number of processes is not 4.
 */
if (mpi_size != 4) {
    printf("This example to set up to use only 4 processes \n");
    printf("Quitting...\n");
    return 0;
}

/*
 * Set up file access property list with parallel I/O access
 */
plist_id = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_fapl_mpio(plist_id, comm, info);

/*
 * Create a new file collectively and release property list identifier.
 */
file_id = H5Fcreate(H5FILE_NAME, H5F_ACC_TRUNC, H5P_DEFAULT, plist_id);
H5Pclose(plist_id);

/*
 * Create the dataspace for the dataset.
 */
dimsf[0] = NX;
dimsf[1] = NY;
chunk_dims[0] = CH_NX;
chunk_dims[1] = CH_NY;
filespace = H5Screate_simple(RANK, dimsf, NULL);
memspace = H5Screate_simple(RANK, chunk_dims, NULL);

/*
 * Create chunked dataset.
 */
plist_id = H5Pcreate(H5P_DATASET_CREATE);
H5Pset_chunk(plist_id, RANK, chunk_dims);
dset_id = H5Dcreate(file_id, DATASETNAME, H5T_NATIVE_INT, filespace,
plist_id);
H5Pclose(plist_id);
H5Sclose(filespace);

/*
 * Each process defines dataset in memory and writes it to the hyperslab
 * in the file.
 */
count[0] = 1;
count[1] = 1;
stride[0] = 1;
stride[1] = 1;
block[0] = chunk_dims[0];
block[1] = chunk_dims[1];
if(mpi_rank == 0) {
    offset[0] = 0;
    offset[1] = 0;
}
}

```



```

if(mpi_rank == 1) {
    offset[0] = 0;
    offset[1] = chunk_dims[1];
}
if(mpi_rank == 2) {
    offset[0] = chunk_dims[0];
    offset[1] = 0;
}
if(mpi_rank == 3) {
    offset[0] = chunk_dims[0];
    offset[1] = chunk_dims[1];
}

/*
 * Select hyperslab in the file.
 */
file_space = H5Dget_space(dset_id);
status = H5Sselect_hyperslab(file_space, H5S_SELECT_SET, offset, stride, count, block);

/*
 * Initialize data buffer
 */
data = (int *) malloc(sizeof(int)*chunk_dims[0]*chunk_dims[1]);
for (i=0; i < (int)chunk_dims[0]*chunk_dims[1]; i++) {
    data[i] = mpi_rank + 1;
}

/*
 * Create property list for collective dataset write.
 */
plist_id = H5Pcreate(H5P_DATASET_XFER);
H5Pset_dxpl_mpio(plist_id, H5FD_MPIO_COLLECTIVE);

status = H5Dwrite(dset_id, H5T_NATIVE_INT, memspace, file_space,
    plist_id, data);
free(data);

/*
 * Close/release resources.
 */
H5Dclose(dset_id);
H5Sclose(file_space);
H5Sclose(memspace);
H5Pclose(plist_id);
H5Fclose(file_id);

MPI_Finalize();

return 0;
}

```

We'll skip the preliminaries and land right where the file space and the memory space are defined:

```

#define NX      8
#define NY      4
#define CH_NX   4
#define CH_NY   2

```

```

...
dimsf[0] = NX;
dimsf[1] = NY;
chunk_dims[0] = CH_NX;
chunk_dims[1] = CH_NY;
filesystem = H5Screate_simple(RANK, dimsf, NULL);
memspace = H5Screate_simple(RANK, chunk_dims, NULL);

```

So, the filesystem itself is going to be  $8 \times 4$  and the memory space for each process is going to be  $4 \times 2$ . Now we create the dataset and observe that we request that this dataset be chunked with chunks of the same size as the memory datasets. But this chunking of the dataset, i.e., its physical partitioning, is quite unrelated to our logical chunking, which we will have to implement by selecting an appropriate hyperslab.

```

#define DATASETNAME "IntArray"
...
plist_id = H5Pcreate(H5P_DATASET_CREATE);
H5Pset_chunk(plist_id, RANK, chunk_dims);
dset_id = H5Dcreate(file_id, DATASETNAME, H5T_NATIVE_INT, filesystem,
plist_id);
H5Pclose(plist_id);
H5Sclose(filespace);

```

Now we have to define counts, strides, block sizes and offsets for each hyperslab:

```

count[0] = 1;
count[1] = 1;
stride[0] = 1;
stride[1] = 1;
block[0] = chunk_dims[0];
block[1] = chunk_dims[1];
if(mpi_rank == 0) {
    offset[0] = 0;
    offset[1] = 0;
}
if(mpi_rank == 1) {
    offset[0] = 0;
    offset[1] = chunk_dims[1];
}
if(mpi_rank == 2) {
    offset[0] = chunk_dims[0];
    offset[1] = 0;
}
if(mpi_rank == 3) {
    offset[0] = chunk_dims[0];
    offset[1] = chunk_dims[1];
}
}

```

Each process is going to write just one block. The stride between the blocks is going to be 1 in each direction, i.e., no striding – but here we could probably default, since you cannot talk about a stride if you are going to write just one block. The block size is going to be  $4 \times 2$  for each process. Process of rank 0 writes its block at (0,0), process of rank 1 writes its block at (0,2), process of rank 2 writes its block at (4,0) and process of rank 3 writes its block at (4,2).

Now each process is ready to select its hyperslab:

```

    filespace = H5Dget_space(dset_id);
    status = H5Sselect_hyperslab(filespace, H5S_SELECT_SET, offset, stride, count, block);

```

At this stage the data gets initialized in memory, we add a collective MPI-IO write request to the property list and we write the data, whereupon the previously allocated buffer data gets freed.

```

    data = (int *) malloc(sizeof(int)*chunk_dims[0]*chunk_dims[1]);
    for (i=0; i < (int)chunk_dims[0]*chunk_dims[1]; i++) {
        data[i] = mpi_rank + 1;
    }
    plist_id = H5Pcreate(H5P_DATASET_XFER);
    H5Pset_dxpl_mpio(plist_id, H5FD_MPIO_COLLECTIVE);
    status = H5Dwrite(dset_id, H5T_NATIVE_INT, memspace, filespace,
        plist_id, data);
    free(data);

```

Finally we close all resources and finalize MPI and exit:

```

    H5Dclose(dset_id);
    H5Sclose(filespace);
    H5Sclose(memspace);
    H5Pclose(plist_id);
    H5Fclose(file_id);
    MPI_Finalize();
    return 0;

```

## 7.6 Point Selection

The previous section (or, to be more exact, the section, which was subdivided into two subsections, of which the second was subdivided into 4 sub-sub-sections) talked about selecting hyperslabs, i.e., subsets of a dataset. But what if we want to modify just a specific element of the dataset, not its whole subset? There is an HDF5 dataspace function, `H5Sselect_elements`, that lets us focus the dataspace on individual points. We are going to demonstrate how this function works in this section.

We are going to create two HDF5 files. The first one is going to have just one  $3 \times 4$  data set in it filled with zeros, and the second one is going to have just one  $3 \times 4$  data set in it filled with ones. Then we're going to close the files and open them again for an update. We are going to write 59 in the (0,1) location of each dataset and 53 in the (0,3) location of each dataset. Then the files will get closed again, re-opened for reading, their datasets read and printed on standard output.

Here you can see how the program works on the AVIDD cluster.

```

gustav@bh1 $ pwd
/N/B/gustav/src/I590/HDF5/points
gustav@bh1 $ h5cc -o h5_copy h5_copy.c
gustav@bh1 $ ./h5_copy

```

```

Dataset 'Copy1' in file 'copy1.h5' contains:
  0  59  0  53
  0  0  0  0

```

```

0 0 0 0

Dataset 'Copy2' in file 'copy2.h5' contains:
  1 59 1 53
  1 1 1 1
  1 1 1 1
gustav@bh1 $ ls
copy1.h5 copy2.h5 h5_copy h5_copy.c h5_copy.o
gustav@bh1 $ h5dump copy1.h5
HDF5 "copy1.h5" {
GROUP "/" {
  DATASET "Copy1" {
    DATATYPE H5T_STD_I32LE
    DATASPACE SIMPLE { ( 3, 4 ) / ( 3, 4 ) }
    DATA {
      0, 59, 0, 53,
      0, 0, 0, 0,
      0, 0, 0, 0
    }
  }
}
}
gustav@bh1 $ h5dump copy2.h5
HDF5 "copy2.h5" {
GROUP "/" {
  DATASET "Copy2" {
    DATATYPE H5T_STD_I32LE
    DATASPACE SIMPLE { ( 3, 4 ) / ( 3, 4 ) }
    DATA {
      1, 59, 1, 53,
      1, 1, 1, 1,
      1, 1, 1, 1
    }
  }
}
}
}
gustav@bh1 $

```

This is the listing of the program taken from the NCSA HDF5 Tutorial:

```

/*****
/*
/* PROGRAM: h5_copy.c
/* PURPOSE: Shows how to use the H5SCOPY function.
/* DESCRIPTION:
/* This program creates two files, copy1.h5, and copy2.h5.
/* In copy1.h5, it creates a 3x4 dataset called 'Copy1',
/* and write 0's to this dataset.
/* In copy2.h5, it create a 3x4 dataset called 'Copy2',
/* and write 1's to this dataset.
/* It closes both files, reopens both files, selects two
/* points in copy1.h5 and writes values to them. Then it
/* does an H5Scopy from the first file to the second, and
/* writes the values to copy2.h5. It then closes the
/* files, reopens them, and prints the contents of the
/* two datasets.
/*
*****/

```

```

#include "hdf5.h"
#define FILE1 "copy1.h5"
#define FILE2 "copy2.h5"

#define RANK 2
#define DIM1 3
#define DIM2 4
#define NUMP 2

int main (void)
{
    hid_t  file1, file2, dataset1, dataset2;
    hid_t  mid1, mid2, fid1, fid2;
    hsize_t fdim[] = {DIM1, DIM2};
    hsize_t mdim[] = {DIM1, DIM2};
    hsize_t start[2], stride[2], count[2], block[2];
    int  buf1[DIM1][DIM2];
    int  buf2[DIM1][DIM2];
    int  bufnew[DIM1][DIM2];
    int  val[] = {53, 59};
    hsize_t marray[] = {2};
    hssize_t coord[NUMP][RANK];
    herr_t ret;
    uint  i, j;

    /*
    /* Create two files containing identical datasets. Write 0's to one
    /* and 1's to the other.
    /*
    /*
    /*
    /*
    for ( i = 0; i < DIM1; i++ )
        for ( j = 0; j < DIM2; j++ )
            buf1[i][j] = 0;

        for ( i = 0; i < DIM1; i++ )
            for ( j = 0; j < DIM2; j++ )
                buf2[i][j] = 1;

    file1 = H5Fcreate(FILE1, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
    file2 = H5Fcreate(FILE2, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

    fid1 = H5Screate_simple (RANK, fdim, NULL);
    fid2 = H5Screate_simple (RANK, fdim, NULL);

    dataset1 = H5Dcreate (file1, "Copy1", H5T_NATIVE_INT, fid1, H5P_DEFAULT);
    dataset2 = H5Dcreate (file2, "Copy2", H5T_NATIVE_INT, fid2, H5P_DEFAULT);

    ret = H5Dwrite(dataset1, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT, buf1);
    ret = H5Dwrite(dataset2, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT, buf2);

    ret = H5Dclose (dataset1);
    ret = H5Dclose (dataset2);

    ret = H5Sclose (fid1);

```

```

ret = H5Sclose (fid2);

ret = H5Fclose (file1);
ret = H5Fclose (file2);

/*****
/*
/* Open the two files.  Select two points in one file, write values to
/* those point locations, then do H5Scopy and write the values to the
/* other file.  Close files.
/*
/*
*****/

file1 = H5Fopen (FILE1, H5F_ACC_RDWR, H5P_DEFAULT);
file2 = H5Fopen (FILE2, H5F_ACC_RDWR, H5P_DEFAULT);
dataset1 = H5Dopen (file1, "Copy1");
dataset2 = H5Dopen (file2, "Copy2");
fid1 = H5Dget_space (dataset1);
mid1 = H5Screate_simple(1, marray, NULL);
coord[0][0] = 0; coord[0][1] = 3;
coord[1][0] = 0; coord[1][1] = 1;

ret = H5Sselect_elements (fid1, H5S_SELECT_SET, NUMP, (const hssize_t **)coord);

ret = H5Dwrite (dataset1, H5T_NATIVE_INT, mid1, fid1, H5P_DEFAULT, val);

fid2 = H5Scopy (fid1);

ret = H5Dwrite (dataset2, H5T_NATIVE_INT, mid1, fid2, H5P_DEFAULT, val);

ret = H5Dclose (dataset1);
ret = H5Dclose (dataset2);
ret = H5Sclose (fid1);
ret = H5Sclose (fid2);
ret = H5Fclose (file1);
ret = H5Fclose (file2);
ret = H5Sclose (mid1);

/*****
/*
/* Open both files and print the contents of the datasets.
/*
/*
*****/

file1 = H5Fopen (FILE1, H5F_ACC_RDWR, H5P_DEFAULT);
file2 = H5Fopen (FILE2, H5F_ACC_RDWR, H5P_DEFAULT);
dataset1 = H5Dopen (file1, "Copy1");
dataset2 = H5Dopen (file2, "Copy2");

ret = H5Dread (dataset1, H5T_NATIVE_INT, H5S_ALL, H5S_ALL,
               H5P_DEFAULT, bufnew);

printf ("\nDataset 'Copy1' in file 'copy1.h5' contains: \n");
for (i=0;i<DIM1; i++) {
    for (j=0;j<DIM2;j++) printf ("%3d ", bufnew[i][j]);
    printf("\n");
}

```

```

printf ("\nDataset 'Copy2' in file 'copy2.h5' contains: \n");

ret = H5Dread (dataset2, H5T_NATIVE_INT, H5S_ALL, H5S_ALL,
              H5P_DEFAULT, bufnew);

for (i=0;i<DIM1; i++) {
    for (j=0;j<DIM2;j++) printf ("%3d ", bufnew[i][j]);
    printf("\n");
}
ret = H5Dclose (dataset1);
ret = H5Dclose (dataset2);
ret = H5Fclose (file1);
ret = H5Fclose (file2);
}

```

This is not an MPI program (just in case you got used to the HDF5/MPI-IO mixture). Its first part is completely standard HDF5: we create the files, we create simple dataspace, then we create the datasets, we write zeros on one and ones on the other one, then close it all.

In the second part of the program new things happen. We open both files and the datasets within them. Then we extract the dataspace from the first dataset, create a simple dataspace for the two numbers {53, 59}, and call `H5Sselect_elements` to narrow the dataspace to just the two selected locations:

```

#define NUMP 2
...
    fid1 = H5Dget_space (dataset1);
    mid1 = H5Screate_simple(1, marray, NULL);
    coord[0][0] = 0; coord[0][1] = 3;
    coord[1][0] = 0; coord[1][1] = 1;
    ret = H5Sselect_elements (fid1, H5S_SELECT_SET, NUMP, (const hssize_t **)coord);

```

The first parameter in the call to `H5Sselect_elements` is the dataset. This is followed by the mode of selection, in this case we use the same mode as in our previous examples with hyperslabs, i.e., we are going to replace the original dataspace with the one that focuses on the two points in the dataspace only. The third parameter, here it is `NUMP`, specifies the number of points and the last parameter is the array that provides the point coordinates. Here they are (0,3) and (0,1).

Now we can write our data on just these two locations:

```

    int val[] = {53, 59};
...
    ret = H5Dwrite (dataset1, H5T_NATIVE_INT, mid1, fid1, H5P_DEFAULT, val);

```

Observe that number 53 corresponds to (0,3) and 59 corresponds to (0,1).

Because the datasets on both files have the same geometry and we want to write the data on exactly the same points in the second dataset, we don't need to re-create the dataspace for this operation. We can simply copy `fid1` onto a new dataspace `fid2` by calling function `H5Scopy`:

```

    fid2 = H5Scopy (fid1);

```

Having done this we complete the write on the second file:

```
ret = H5Dwrite (dataset2, H5T_NATIVE_INT, mid1, fid2, H5P_DEFAULT, val);
```

and then close everything again.

The last part of the program opens the files, the datasets, reads them, prints them, and then closes it all. There is nothing new in it, so I'll just skip it.

## 7.7 Compound Datatypes

All our HDF5 examples so far were concerned with writing very elementary data on simple datasets. And the data were all 32-bit integers to make things simpler, the only touch of excitement being small-endian versus big-endian. But what if we have some richly structured data, a mixture of characters, floats, integers and what not? Following the example of MPI HDF5 provides a rich interface for creation of new HDF5 data types. All functions in this family have their names beginning with H5T, where T stands for *type*. You cannot perform any computations on these HDF5 datatypes though. Their purpose is similar to the purpose of MPI datatypes. By defining HDF5 datatypes you tell it how it should extract data from your C-language structures and how it should write the data back on them.

The following program is going to create a simple one-dimensional dataset, which is going to hold an array of length 10, on an HDF5 file, but each element of this dataset is going to store a 3-element structure, whose C-language definition is:

```
typedef struct s1_t {
int    a;
float  b;
double c;
} s1_t;
```

The program will initialize an array of such structures in memory, then it will provide appropriate definitions to HDF5, and will transfer the content of the memory array to the HDF5 dataset.

In the second part of the program we are going to read the data back, in a special way that lets us select certain components of the structure only, and we'll print the results on standard output.

Here is how the program compiles, links and runs on the AVIDD cluster:

```
gustav@bh1 $ pwd
/N/B/gustav/src/I590/HDF5/compound
gustav@bh1 $ h5cc -o h5_compound h5_compound.c
gustav@bh1 $ ./h5_compound

Field c :
1.0000 0.5000 0.3333 0.2500 0.2000 0.1667 0.1429 0.1250 0.1111 0.1000

Field a :
0 1 2 3 4 5 6 7 8 9

Field b :
0.0000 1.0000 4.0000 9.0000 16.0000 25.0000 36.0000 49.0000 64.0000 81.0000
gustav@bh1 $ ls
```



```
SDScompound.h5 h5_compound h5_compound.c h5_compound.o
gustav@bh1 $
```

And here for the more inquisitive minds, is the `h5dump` of the file itself, followed by its `h5ls` listing:

```
gustav@bh1 $ h5dump SDScompound.h5
HDF5 "SDScompound.h5" {
  GROUP "/" {
    DATASET "ArrayOfStructures" {
      DATATYPE H5T_COMPOUND {
        H5T_STD_I32LE "a_name";
        H5T_IEEE_F64LE "c_name";
        H5T_IEEE_F32LE "b_name";
      }
      DATASPACE SIMPLE { ( 10 ) / ( 10 ) }
      DATA {
        {
          0,
          1,
          0
        },
        {
          1,
          0.5,
          1
        },
        {
          2,
          0.333333,
          4
        },
        {
          3,
          0.25,
          9
        },
        {
          4,
          0.2,
          16
        },
        {
          5,
          0.166667,
          25
        },
        {
          6,
          0.142857,
          36
        },
        {
          7,
          0.125,
          49
        },
        {
```

```

        8,
        0.111111,
        64
    },
    {
        9,
        0.1,
        81
    }
}
}
}
}
gustav@bh1 $ h5ls -r -v SDScompound.h5
Opened "SDScompound.h5" with sec2 driver.
/ArrayOfStructures      Dataset {10/10}
  Location: 0:1:0:976
  Links: 1
  Modified: 2003-11-30 17:56:34 EST
  Storage: 160 logical bytes, 160 allocated bytes, 100.00% utilization
  Type: struct {
        "a_name"          +0  native int
        "c_name"          +8  native double
        "b_name"          +4  native float
    } 16 bytes
gustav@bh1 $

```

You can clearly see that information about the structure is provided. The components of the structure are written on fields named "a\_name", "c\_name" and "b\_name". These names are important. They are not just decorative labels. You will use these names in order to read data from selected *fields* into memory. The program itself comes from the NCSA HDF5 Tutorial:

```

/*
 * This example shows how to create a compound data type,
 * write an array which has the compound data type to the file,
 * and read back fields' subsets.
 */

#include "hdf5.h"

#define FILE          "SDScompound.h5"
#define DATASETNAME  "ArrayOfStructures"
#define LENGTH       10
#define RANK         1

int
main(void)
{
    /* First structure and dataset*/
    typedef struct s1_t {
int    a;
float  b;
double c;
    } s1_t;
    s1_t    s1[LENGTH];

```

```

hid_t      s1_tid;      /* File datatype identifier */

/* Second structure (subset of s1_t) and dataset*/
typedef struct s2_t {
double c;
int      a;
} s2_t;
s2_t      s2[LENGTH];
hid_t      s2_tid;      /* Memory datatype handle */

/* Third "structure" ( will be used to read float field of s1) */
hid_t      s3_tid;      /* Memory datatype handle */
float      s3[LENGTH];

int      i;
hid_t      file, dataset, space; /* Handles */
herr_t      status;
hsize_t      dim[] = {LENGTH}; /* Dataspace dimensions */

/*
 * Initialize the data
 */
for (i = 0; i < LENGTH; i++) {
    s1[i].a = i;
    s1[i].b = i*i;
    s1[i].c = 1./(i+1);
}

/*
 * Create the data space.
 */
space = H5Screate_simple(RANK, dim, NULL);

/*
 * Create the file.
 */
file = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

/*
 * Create the memory data type.
 */
s1_tid = H5Tcreate (H5T_COMPOUND, sizeof(s1_t));
H5Tinsert(s1_tid, "a_name", HOFFSET(s1_t, a), H5T_NATIVE_INT);
H5Tinsert(s1_tid, "c_name", HOFFSET(s1_t, c), H5T_NATIVE_DOUBLE);
H5Tinsert(s1_tid, "b_name", HOFFSET(s1_t, b), H5T_NATIVE_FLOAT);

/*
 * Create the dataset.
 */
dataset = H5Dcreate(file, DATASETNAME, s1_tid, space, H5P_DEFAULT);

/*
 * Write data to the dataset;
 */
status = H5Dwrite(dataset, s1_tid, H5S_ALL, H5S_ALL, H5P_DEFAULT, s1);

```

```

/*
 * Release resources
 */
H5Tclose(s1_tid);
H5Sclose(space);
H5Dclose(dataset);
H5Fclose(file);

/*
 * Open the file and the dataset.
 */
file = H5Fopen(FILE, H5F_ACC_RDONLY, H5P_DEFAULT);

dataset = H5Dopen(file, DATASETNAME);

/*
 * Create a data type for s2
 */
s2_tid = H5Tcreate(H5T_COMPOUND, sizeof(s2_t));

H5Tinsert(s2_tid, "c_name", HOFFSET(s2_t, c), H5T_NATIVE_DOUBLE);
H5Tinsert(s2_tid, "a_name", HOFFSET(s2_t, a), H5T_NATIVE_INT);

/*
 * Read two fields c and a from s1 dataset. Fields in the file
 * are found by their names "c_name" and "a_name".
 */
status = H5Dread(dataset, s2_tid, H5S_ALL, H5S_ALL, H5P_DEFAULT, s2);

/*
 * Display the fields
 */
printf("\n");
printf("Field c : \n");
for( i = 0; i < LENGTH; i++) printf("%.4f ", s2[i].c);
printf("\n");

printf("\n");
printf("Field a : \n");
for( i = 0; i < LENGTH; i++) printf("%d ", s2[i].a);
printf("\n");

/*
 * Create a data type for s3.
 */
s3_tid = H5Tcreate(H5T_COMPOUND, sizeof(float));

status = H5Tinsert(s3_tid, "b_name", 0, H5T_NATIVE_FLOAT);

/*
 * Read field b from s1 dataset. Field in the file is found by its name.
 */
status = H5Dread(dataset, s3_tid, H5S_ALL, H5S_ALL, H5P_DEFAULT, s3);

/*
 * Display the field
 */

```

```

printf("\n");
printf("Field b : \n");
for( i = 0; i < LENGTH; i++) printf("%.4f ", s3[i]);
printf("\n");

/*
 * Release resources
 */
H5Tclose(s2_tid);
H5Tclose(s3_tid);
H5Dclose(dataset);
H5Fclose(file);

return 0;
}

```

Let us analyze the program.

The program begins by initializing the array of structures:

```

#define LENGTH      10
...
typedef struct s1_t {
int    a;
float  b;
double c;
} s1_t;
s1_t    s1[LENGTH];
...
for (i = 0; i < LENGTH; i++) {
    s1[i].a = i;
    s1[i].b = i*i;
    s1[i].c = 1./(i+1);
}

```

Then we create a *simple* dataspace. Observe that there is no information provided at this stage about the structure of the entities the dataspace is made of:

```

#define LENGTH      10
#define RANK        1
...
hsize_t    dim[] = {LENGTH};
...
space = H5Screate_simple(RANK, dim, NULL);

```

We create the HDF5 file:

```
file = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
```

and now we have to tell HDF5 more about the structure that the dataset is going to be made of:

```

s1_tid = H5Tcreate (H5T_COMPOUND, sizeof(s1_t));
H5Tinsert(s1_tid, "a_name", HOFFSET(s1_t, a), H5T_NATIVE_INT);
H5Tinsert(s1_tid, "c_name", HOFFSET(s1_t, c), H5T_NATIVE_DOUBLE);
H5Tinsert(s1_tid, "b_name", HOFFSET(s1_t, b), H5T_NATIVE_FLOAT);

```

We have two new functions here. The first one `H5Tcreate` creates the new type. The first parameter in this call is an HDF5 constant `H5T_COMPOUND`, which tells

HDF5 that this is going to be a compound object. The other possible choices here are `H5T_OPAQUE` and `H5T_ENUM`. The second parameter specifies the size of the whole object.

The second function is `H5Tinsert` and it is used to tell HDF5 how exactly the new type, whose identifier is `s1_tid`, is made.

The first call specifies that the first component is of type `H5T_NATIVE_INT`, it should be inserted into the field of name `"a_name"`, and it should be picked from the memory location that corresponds to the offset of the `a` component in the C-language structure `s1_t`. A special HDF5 macro `HOFFSET` can be used to get it. You may recall that we had a function in MPI, called `MPI_Get_address`, whose purpose was, similarly, to find offsets of various structure components in order to provide `MPI_Type_struct` with displacements.

The second call tells HDF5 that the second component of the structure is of type `H5T_NATIVE_DOUBLE` and its memory location corresponds to the location of the `c` component of the `s1_t` structures. It should be inserted into the field named `"c_name"`.

Finally the third call tells HDF5 that the third component is to be inserted into the field called `"b_name"`, that it is an `H5T_NATIVE_FLOAT` and that its memory location corresponds to the location of the `b` component in the `s1_t` structure.

Now we are finally ready to create the dataset:

```
#define DATASETNAME "ArrayOfStructures"
...
dataset = H5Dcreate(file, DATASETNAME, s1_tid, space, H5P_DEFAULT);
```

The combination of the two parameters, (`s1_tid`, `space`), tells `H5Dcreate` what the space should really be like.

Now we can perform the simple HDF5 write and release all resources:

```
status = H5Dwrite(dataset, s1_tid, H5S_ALL, H5S_ALL, H5P_DEFAULT, s1);
H5Tclose(s1_tid);
H5Sclose(space);
H5Dclose(dataset);
H5Fclose(file);
```

In the second part of the program we open the file for reading, open the dataset, and then read the data. But we read it a little differently than we have written it originally. This is because in the first read we are going to read the `"c_name"` and `"a_name"` fields of the dataset only. To do this we create a new type:

```
#define LENGTH 10
...
typedef struct s2_t {
double c;
int a;
} s2_t;
s2_t s2[LENGTH];
...
s2_tid = H5Tcreate(H5T_COMPOUND, sizeof(s2_t));
H5Tinsert(s2_tid, "c_name", HOFFSET(s2_t, c), H5T_NATIVE_DOUBLE);
H5Tinsert(s2_tid, "a_name", HOFFSET(s2_t, a), H5T_NATIVE_INT);
```

and use it to read the selected fields from the dataset – the data goes then directly into `s2` into slots specified by `HOFFSET(s2_t,c)` and `HOFFSET(s2_t,a)` respectively:

```
status = H5Dread(dataset, s2_tid, H5S_ALL, H5S_ALL, H5P_DEFAULT, s2);
```

Having read the data we print it on standard output:

```
printf("\n");
printf("Field c : \n");
for( i = 0; i < LENGTH; i++) printf("%.4f ", s2[i].c);
printf("\n");

printf("\n");
printf("Field a : \n");
for( i = 0; i < LENGTH; i++) printf("%d ", s2[i].a);
printf("\n");
```

Now we create an HDF5 type for reading the "b\_name" field:

```
s3_tid = H5Tcreate(H5T_COMPOUND, sizeof(float));
status = H5Tinsert(s3_tid, "b_name", 0, H5T_NATIVE_FLOAT);
```

The offset in this case is zero, because we are going to read the data onto a plain array of floats, `s3`. Then we read the field and print the array `s3` on standard output:

```
#define LENGTH      10
...
float      s3[LENGTH];
...
status = H5Dread(dataset, s3_tid, H5S_ALL, H5S_ALL, H5P_DEFAULT, s3);
printf("\n");
printf("Field b : \n");
for( i = 0; i < LENGTH; i++) printf("%.4f ", s3[i]);
printf("\n");
```

Having done all this, we close the shop and exit.

```
H5Tclose(s2_tid);
H5Tclose(s3_tid);
H5Dclose(dataset);
H5Fclose(file);
return 0;
```

## 7.8 Iterating over HDF5 Groups

You are obviously familiar with the `-R` switch to the `ls` command and with the `-r` switch to the `h5ls` command, which, in both cases, invoke recursive traversal of the whole directory tree. There is a function in HDF5 called `H5Giterate`, which can be used to iterate over the whole group and all its subgroups, picking up on every object within the group and applying a specified function, presumably made of other HDF5 functions, to it. Program `h5ls` when invoked with the `-r` option, is an example of how to use this function.

In this section we are going to write our own much simpler version of `h5ls`. The program is called `h5_iterate`. It creates an HDF5 file and endows it with

a certain structure. Then it proceeds to iterate over the members of the "/" group and tells us what the members it finds are.

Here is how this program works on our AVIDD cluster.

```
gustav@bh1 $ pwd
/N/B/gustav/src/I590/HDF5/iterate
gustav@bh1 $ h5cc -o h5_iterate h5_iterate.c
gustav@bh1 $ ./h5_iterate
Objects in the root group are:

Object with name Dataset1 is a dataset
Object with name Datatype1 is a named datatype
Object with name Group1 is a group
gustav@bh1 $ ls
h5_iterate h5_iterate.c h5_iterate.o iterate.h5
gustav@bh1 $
```

We can have a closer look at what's inside the file created by the program, `iterate.h5`, with `h5dump`:

```
gustav@bh1 $ h5dump iterate.h5
HDF5 "iterate.h5" {
GROUP "/" {
  DATASET "Dataset1" {
    DATATYPE H5T_STD_U32LE
    DATASPACE SIMPLE { ( 4 ) / ( 4 ) }
    DATA {
      0, 0, 0, 0
    }
  }
  DATATYPE "Datatype1" H5T_COMPOUND {
    H5T_STD_I32LE "a";
    H5T_STD_I32LE "b";
    H5T_IEEE_F32LE "c";
  }
  GROUP "Group1" {
  }
}
}
gustav@bh1 $
```

The iterator, `H5Giterate`, will not, by itself, go into subgroups recursively. For this to happen you will have to write a recursive procedure yourself, but this recursive procedure can be based on `H5Giterate`.

Here is the program, which comes from the NCSA HDF5 Tutorial:

```
#include "hdf5.h"

#define FILE "iterate.h5"
#define FALSE 0

/* 1-D dataset with fixed dimensions */
#define SPACE1_NAME "Space1"
#define SPACE1_RANK 1
#define SPACE1_DIM1 4

herr_t file_info(hid_t loc_id, const char *name, void *opdata);
/* Operator function */
```



```

int
main(void) {
    hid_t file; /* HDF5 File IDs */
    hid_t dataset; /* Dataset ID */
    hid_t group; /* Group ID */
    hid_t sid; /* Dataspace ID */
    hid_t tid; /* Datatype ID */
    hsize_t dims[] = {SPACE1_DIM1};
    herr_t ret; /* Generic return value */

    /* Compound datatype */
    typedef struct s1_t {
        unsigned int a;
        unsigned int b;
        float c;
    } s1_t;

    /* Create file */
    file = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

    /* Create dataspace for datasets */
    sid = H5Screate_simple(SPACE1_RANK, dims, NULL);

    /* Create a group */
    group=H5Gcreate(file,"Group1",-1);

    /* Close a group */
    ret = H5Gclose(group);

    /* Create a dataset */
    dataset=H5Dcreate(file,"Dataset1",H5T_STD_U32LE,sid,H5P_DEFAULT);

    /* Close Dataset */
    ret = H5Dclose(dataset);

    /* Create a datatype */
    tid = H5Tcreate (H5T_COMPOUND, sizeof(s1_t));

    /* Insert fields */
    ret=H5Tinsert (tid, "a", HOFFSET(s1_t,a), H5T_NATIVE_INT);

    ret=H5Tinsert (tid, "b", HOFFSET(s1_t,b), H5T_NATIVE_INT);

    ret=H5Tinsert (tid, "c", HOFFSET(s1_t,c), H5T_NATIVE_FLOAT);

    /* Save datatype for later */
    ret=H5Tcommit (file, "Datatype1", tid);

    /* Close datatype */
    ret = H5Tclose(tid);

    /* Iterate through the file to see members of the root group */

    printf(" Objects in the root group are:\n");
    printf("\n");

    H5Giterate(file, "/", NULL, file_info, NULL);

```

```

    /* Close file */
    ret = H5Fclose(file);

    return 0;
}

/*
 * Operator function.
 */
herr_t file_info(hid_t loc_id, const char *name, void *opdata)
{
    H5G_stat_t statbuf;

    /*
     * Get type of the object and display its name and type.
     * The name of the object is passed to this function by
     * the Library. Some magic :- )
     */
    H5Gget_objinfo(loc_id, name, FALSE, &statbuf);
    switch (statbuf.type) {
    case H5G_GROUP:
        printf(" Object with name %s is a group \n", name);
        break;
    case H5G_DATASET:
        printf(" Object with name %s is a dataset \n", name);
        break;
    case H5G_TYPE:
        printf(" Object with name %s is a named datatype \n", name);
        break;
    default:
        printf(" Unable to identify an object ");
    }
    return 0;
}

```

The program begins by creating an HDF file, then we create a simple dataspace, then a group called "Group1". And once the group is created we close it. We are not going to do anything with it. Then we create a dataset using the simple dataspace and close it too. Finally we create a compound type that corresponds to the following C-language structure:

```

typedef struct s1_t {
    unsigned int a;
    unsigned int b;
    float c;
} s1_t;

```

and *commit* it to the file by calling function `H5Tcommit`:

```

ret = H5Tcommit (file, "Datatype1", tid);
ret = H5Tclose(tid);

```

This has the effect that the datatype definition is written on the HDF5 file, but not actually used for anything. It can be retrieved from the file later and used, if need be.

Now, having created all these objects, we invoke `H5Giterate`:

```
printf(" Objects in the root group are:\n");
printf("\n");
H5Giterate(file, "/", NULL, file_info, NULL);
```

We are going to iterate over the group "/" of file `file`. To every object found we are going to apply function `file_info`, which is defined towards the end of the program source. The last parameter, which is `NULL`, can be used to pass additional values to function `file_info` (e.g., a character "v" for "verbose listing"). The third parameter, which is `NULL` too, is the location at which to begin the iteration. If it is set to `NULL`, the iteration begins at the beginning of the group, so to speak.

Normally the passed function, here it is `file_info`, is expected to take three parameters, which will be provided to it by `H5Giterate`. They will be the object location identifier, the object name, and the third parameter is what we have set to `NULL`:

```
herr_t file_info(hid_t loc_id, const char *name, void *opdata)
```

We have not talked about locations in this course yet. Every HDF5 object has, apart from its name, also its location or a reference. The objects can be accessed by providing their locations instead of their names. But this is all you need to know for now.

Function `file_info` calls `H5Gget_objinfo` in order to find what the objects passed to it by `H5Giterate` are:

```
H5Gget_objinfo(loc_id, name, FALSE, &statbuf);
```

The object type is provided in the `statbuf` structure, whose `type` member contains the information. The third parameter, which is set to `FALSE`, tells `H5Gget_objinfo` that it should *not* follow links.

Now, depending on what is returned on `statbuf.type` function `file_info` writes a corresponding message on standard output and returns.

```
switch (statbuf.type) {
case H5G_GROUP:
    printf(" Object with name %s is a group \n", name);
    break;
case H5G_DATASET:
    printf(" Object with name %s is a dataset \n", name);
    break;
case H5G_TYPE:
    printf(" Object with name %s is a named datatype \n", name);
    break;
default:
    printf(" Unable to identify an object ");
}
return 0;
```

After `H5Giterate` has returned, we close the file and exit:

```
ret = H5Fclose(file);
return 0;
```

## 7.9 What Else in HDF5?

What else is there in HDF5?

Plenty. I have skipped the whole issue of accessing HDF5 objects by references (or addresses, or locations). This is useful if you want to create a list of certain objects, store the list on a special dataset, then use it to iterate, for example, over the objects. The objects collected this way don't have to live in the same group, otherwise you could simply use `H5Giterate` to pick them up. Once you have an object you can obtain its address, within the file, by calling function `H5Rcreate`. You can also create lists of references to specific regions within the datasets, also by calling the same function `H5Rcreate`.

I have only showed you three HDF5 tools, `h5dump`, `h5ls` and `h5cc`, but there are more:

**h5diff** identifies and locates differences between two HDF5 files

**h5repart** lets you re-partition an HDF5 file and create a file family out of it

**h5import** lets you import data into an existing or a new HDF5 file

**gif2h5** lets you convert a GIF file to an HDF5 file

**h52gif** lets you extract images from an HDF5 file onto GIF files

**h5perf** lets you measure performance of HDF5 programs

**h5fc** lets you compile Fortran90 programs that call HDF5 functions

**H5c++** lets you compile C++ programs that call HDF5 functions

**h4toh5** lets you convert an HDF4 file to an HDF5 file

**h5toh4** lets you convert an HDF5 file to an HDF4 file

**JHI5** is the Java package that wraps around HDF5

**H5View** is a Java-based tool for browsing and editing HDF5 files

**H5gen** is a Java utility that reads an XML description of an HDF5 file and generates the corresponding HDF5 file

**VisAD** is a Java toolkit for constructing interactive and collaborative visualization and analysis of numerical data

You can find more about HDF5 at <http://hdf.ncsa.uiuc.edu/HDF5/>.

## Chapter 8

# Other Parallel Libraries

HDF5 is an example of a library that lives on top of MPI and lets you carry out quite involved operations on portions (datasets) of structured files in parallel and on a somewhat higher level than what MPI-IO lets you do, although the way that a dataset gets divided into hyperslabs is quite similar to the way that a portion of a file gets divided with file views in MPI-IO.

The whole point behind the development of MPI was to enable construction of parallel libraries in such a way that inter-process communications within the library would not interfere with inter-process communications within the calling program – and this is what the device of an MPI communicator is for.

So, what other parallel libraries, apart from HDF5, are out there, in the free-ware world? The answer is, not that many, but there are some worth mentioning.

We shall start from the Netlib Repository at the University of Tennessee and at the Oak Ridge National Laboratory, which will point you to High Performance Math Software tools at the National HPC Software Exchange (NHSE). HPC-Netlib focuses on four areas at present:

- Numerical linear algebra
- Graph and mesh partitioning
- Parallel PDE solvers
- Optimization

The NHSE Software and Technology page provides numerous links to various Domain Specific Repositories. Of these most reside at various DoD installations, some of which do not distribute their software on-line. You may wish to visit:

- Computational Fluid Dynamics Software Catalog
- Grid Generation Software Catalog
- Performance Case Studies

- Programming Tools Catalog
- Scalable Parallel Programming Tools Catalog
- Signal Image Processing Software Catalog

NCSA distributes HDF5, of course, but also a number of other tools and libraries, most notably:

- Parallel Processing Tools
- Numerical Programs and Routines

And, last but not least, NHSE distributes its own tools and applications, for example:

- Benchmark and Example Programs
- Data Analysis and Visualization
- Distributed Processing Tools
- Numerical Programs and Routines
- Parallel Processing Tools
- Scientific and Engineering Applications

In the last category, i.e., Scientific and Engineering Applications, you will find, amongst others, tools for Financial Modeling, Simulated Annealing, Crystallography, Adaptive Particle-Mesh Methods, Distributed Parallel Multipole Tree Algorithms, Flux-Corrected Transport, Molecular Dynamics, Image Processing, Finite Volume and Finite Element Simulations.

The Princeton Plasma Physics Laboratory distributes numerous applications and tools for scientific programming in general and for solving various complicated plasma problems in particular – *some* parallel. You will find their software distribution site and catalogues at <http://w3.pppl.gov/rib/repositories/NTCC/catalog>

The Argonne National Laboratory is the home of MPI, MPICH, ROMIO and even PVFS nowadays. The Mathematics and Computer Science Division distributes other less know, but equally useful, systems and products. Amongst them:

**ALICE** The toolkit provides three primary functionalities: the visualization of individual data sets, data comparison via the visualization of the differences between data sets, and image comparison via the simultaneous visualization of multiple data sets. For each functionality, one or more standard visualization techniques, such as vector field glyphs, animated streamlines and flow fields, cutting planes, and scalar field height profiles, are used to gain insight into the data. Additional insights are gained through the dynamic selection of the color maps and through data manipulation techniques such as magnification, culling, and exaggeration.

**ChemIO** The toolkit facilitates the development of portable, high-performance computational chemistry codes, by defining a standard I/O API that meets chemistry requirements, and providing high-performance implementations of this API on different high-performance computers.

**FLIC** a Fortran loop and index converter, is a parser-based source translation tool that automates the conversion of program loops and array indices for distributed-memory parallel computers.

**NEOS** The NEOS Server 3.0 is the first network-enabled problem-solving environment for a wide class of applications in business, science, and engineering. Included are state-of-the-art solvers in integer programming, nonlinearly constrained optimization, bound-constrained optimization, unconstrained optimization, linear programming, stochastic linear programming, complementarity problems, linear network optimization, semidefinite programming, and administrative programming.

**PETSc** The Portable, Extensible Toolkit for Scientific computation, is a suite of uni- and parallel-processor codes for solving large-scale problems modeled by partial differential equations. PETSc employs the MPI standard for all message-passing communication. The code is written in a data-structure-neutral manner to enable easy reuse and flexibility.

**Ptools** The Scalable Unix Tools project – which offers parallel, scalable version of common Unix commands for parallel machines with a Unix on each node.

**RSL** RSL is a runtime system library for implementing regular-grid models with nesting on distributed memory parallel computers. RSL provides support for automatically decomposing multiple model domains and for redistributing work between processors at run time for dynamic load balancing. The interface to RSL supports Fortran77 and Fortran90.





## Chapter 9

# Databases: From Small to Huge

*We won't have enough time left in this semester to cover this very important and interesting topic in depth. An introductory lecture, for which the notes are provided in this chapter, is going to be given on the 18th of November, this on account of the AVIDD cluster possibly not being available because of the SC2003 conference.*

A database is a suite of software utilities for maintenance and operations on *tables* of data. If the data is stored as a table, a great deal of optimizations for data searches, updates, insertions, deletions, and other operations, including storing the data itself, can be implemented. A single data base will normally contain numerous tables, but a single table cannot usually belong to more than one data base.

Data bases can store data on disk drives formatted specially for their use – this is normally the most efficient way and this is how corporate computer centers usually go about it – or on the native file system. The latter is almost certainly going to be the case on a research system such as AVIDD, where administrators would loath to sacrifice general purpose disk space and have to format specialized partitions for the data base.

A data base is a typical example of a client-server system. Every data base has a data base server that is always up, waiting for connections, and that takes care of the data base itself. It is quite OK to think of the server and the data it guards, as *the* data base. In order to access the data, or even in order to insert the data into the data base, you have to use a data base client. The client normally runs on a machine other than the machine on which the server runs, although a data base administrator may use a client running on the same machine in order to manage the data base.

Data bases are the darlings of corporate computing. You'll find them in just about every company beginning with a small real estate agent operation, where a single data base would probably run under Windows on a single PC server, all

the way up to major corporations that often maintain hundreds, even thousands, of data bases running on mainframes, parallel computer systems such as IBM SP, and large SMPs like HP Superdome or Sun Starfire.

On the other hand, data bases have not made great strides in scientific computing, because of their cost, fuss associated with setting them up and with maintaining them, inflexibility – once you have defined a table, you are stuck with it, and, besides, not every type of data fits into a table, and *because* they are client-server systems. So, almost exactly what endears data bases to businesses has been a huge turn off for scientists . . . and hackers.

But this is changing.

For many years there were no freeware data bases of quality. Once upon a time there was a data base called Ingress, which was distributed for *almost* free with BSD UNIX – “almost”, because you had to pay some token money for BSD UNIX (about \$500 per campus). But this was a long time ago (about 20 years or so – some of you may not have even been born yet). Ingress programmers went commercial later and even though the earlier version of Ingress remained free, it was very antiquated and unsupported.

There was a good reason for this state of affairs of course. The reason was that writing data bases was such a lucrative, but also difficult, undertaking that no programmer would miss the opportunity to make millions of dollars in the process.

But today we have some freeware data bases, and at least one of them, that I know of, is very good indeed. The data base is called PostgreSQL and it is developed and maintained by an international community of dedicated (usually academic) programmers, much like Linux. PostgreSQL is distributed with Linux and with Cygwin. Even though it is not installed on the AVIDD cluster at the time of this writing, you can always request it and the AVIDD administrators should have no problems adding it to the pool of available software on all AVIDD nodes – although most likely you will just need PostgreSQL clients and client libraries on the computational nodes and the data base itself may run either on a selected AVIDD node or even outside of AVIDD. PostgreSQL has its WWW page at <http://www.postgresql.org/>.

This free availability and ease of use of PostgreSQL made many scientists change their mind about data bases. “Well, if it’s free and I can have it on my PC or on my Linux cluster, then I may just as well have a look at it and think what I could do with it.”

The other reason is that the two *great* data bases, namely, Oracle and DB2 are free to academics, at least in this country, too. For example, if you enroll in the DB2 Scholars’ Program with IBM, you’ll get the whole DB2 shipped to you in a box and you don’t have to pay a cent for it. There is a similar Oracle program too. Both DB2 and Oracle are insanely great products and you should not believe people who will tell you that you should stay away from one or the other. But both can be pretty pricy too, the moment you leave academia. Microsoft’s SQL-Server is not free to academic institutions though (at least at the time I’m writing these words and as far as I know) and it is not quite in the same league as DB2 and Oracle. The latter two run on everything: PCs,

Windows, Linux, Suns, HP systems, IBM systems, you name it. But Microsoft's SQL-Server runs under Windows only (although you can connect to it from non-Windows clients) and it had quite a number of very serious security problems recently.

Next, it ought to be said too that science has evolved recently in the direction where problems acquire great complexity and structural richness. Take, for example, epidemiology – a typical area of medical science where data bases are quite essential. In fact anything to do with medicine involves data bases, because you need to keep patients' records somewhere and you need to carry out searches on the records and other queries against such an information system. The Regenstrief Institute in Indianapolis has built and maintains the *largest* medical information system in the world.

Another example of data bases being used in science are data bases at SLAC, which are used to store data pertaining to high energy physics events observed in SLAC experiments. Every event is described by the means of a table row. The data bases they have are amongst the largest in the world, because even a very short experiment can easily generate trillions of events worth keeping in the data base. The manipulation of and searches on such gigantic data bases, which can very quickly grow to tens, even hundreds of TBs, is an extremely difficult undertaking. Even the largest business data bases seldom grow above some hundreds of GBs. The reason for the latter is that business data normally has to be typed in by human operators, whereas data flows into the SLAC data bases directly from measuring instruments.

Library science is an “in-between” category. Normally it should be more like business, because library catalogues, which are data bases too, have to be constructed by human operators. But more recently methods have been developed to extract abstracts, keywords and other cataloguing information from electronic texts automatically. This makes the task of feeding bibliographic data into the catalogues a little more like the SLAC systems. On the other hand, the texts themselves, still have to be written by humans, which limits the size of the task dramatically, because humans are slow typists. If you took the whole National Library collection and made an electronic copy of every book they have there (but not in terms of images, rather in terms of converting every letter to the corresponding ASCII character) – you would end up with at most a few TBs. On the other hand, the moment you add images, sounds and videos to the collection, the amount of data skyrockets very quickly. But cataloguing images and videos is a tricky business, because it's very difficult to tell a computer “what to look for”.

Between physics and library science just about every other research discipline can benefit from the use of data bases. In most cases the data bases are probably going to be rather small – they should easily fit onto a single PC. Just today I have seen a Dell advertisement in the Sunday newspaper about a small deskside PC with a 140GB drive, 512MB memory and a very fast 2.3GHz CPU for some \$700 or so. A system like this should be more than sufficient for 99% of data bases that small research groups would like to concoct – especially if the data is to be entered manually or semi-automatically.

A very typical personal research data base is a list of citations, which can be accessed from your `TEX`, `troff` or (shudder) Word document. Such data bases are constructed laboriously over years of the researcher's life, with much pain, devotion, and lots of typos. They're treasured like the Holy Grail, but . . . they are trivial by data base standards. They seldom contain more than a few thousand citations, which translates into a mere 5MB or so.

Consequently, when you think about using a data base in your research, the first question you should ask yourself is how large is this data base going to be. You'll be surprised how difficult it will be to make it truly large. But if you arrive at some moderately impressive numbers here, say, tens of GBs, or some quite impressive numbers like hundreds of GBs then you should definitely consider using parallel DB2 on the AVIDD cluster – especially if you expect to run a lot of searches and data mining procedures against it.

In this chapter I'm going to show you first how you can set up and operate on a PostgreSQL data base under Cygwin – and the same should apply to the AVIDD cluster, because it is the same data base.

Then I am going to discuss, rather briefly, what you are going to get if you switch to parallel DB2.

## 9.1 PostgreSQL under Cygwin

PostgreSQL should be already installed on your Laboratory PCs. The server itself lives in the `/usr/bin` directory and is called `postgres`. But normally we run it through a wrapper called `postmaster`, which also lives in `/usr/bin`. In the NTFS notation, this may be something like `C:\cygwin\usr\bin`. There is also a SQL client there, called `psql`.

The way to run PostgreSQL is as follows: first you have to start the server, and then you talk to it using the client. The server and the client may run on the same machine, or they may run on different machines. Both configurations are supported. The same, of course, applies to Linux and therefore to the AVIDD cluster.

But before you can run the PostgreSQL server, you have to configure the IPC daemon, this is the only part that must be performed by the system administrator. All else you can do yourself, simply as a normal Cygwin (or Windows) user.

If you administer your own PC, then the procedure for configuring and running the IPC daemon is as follows. First you have to become the administrator in order to install the service. Because the service is a Cygwin program, it must be configured using the `cygrunsrv` utility. This utility lives in the Cygwin `/usr/bin` too.

Issue the command:

```
# cygrunsrv --install ipc-daemon --path /usr/bin/ipc-daemon2 \
--desc "IPC daemon for PostgreSQL"
```

Then start the IPC daemon with

```
# cygrunsrv --start ipc-daemon
```

You should be able to see the daemon if you run

```
# ps -ef
```

Alternatively, you can go to

```
-> My Computer
    -> Control Panel
        -> Administrative Tools
            -> Services
```

and start the service the Windows way, i.e., by right-clicking on it and then selecting **Start** – this, at least, is how it works on my PC under Windows 2000.

One way or another, you should have the daemon running. Now once the daemon runs, you can switch back to your normal user account.

You have to decide where you want to keep all your PostgreSQL data. If the data base is going to be for your personal use, you should select a directory in your Cygwin home. If the data base is going to be used by your research group and served from a server, you may as well dedicate the whole drive or at least a partition to it.

In this case I am simply going to create the data base in a directory in my Cygwin home:

```
gustav@WOODLANDS:~/gustav 19:39:57 !539 $ pwd
/home/gustav
gustav@WOODLANDS:~/gustav 19:39:58 !540 $ mkdirhier pgsqldata
gustav@WOODLANDS:~/gustav 19:40:03 !541 $
```

At this stage the directory has to be initialized for work with a PostgreSQL server. The utility that does the initialization is called `initdb` and it will initialize the directory not just for one data base, but for the whole data base *cluster*, i.e., a collection of data bases, that are going to have their data stored in this area:

```
gustav@WOODLANDS:~/gustav 19:47:39 !551 $ initdb -D 'pwd'/pgsqldata
The files belonging to this database system will be owned by user "gustav".
This user must also own the server process.
```

The database cluster will be initialized with locale C.

```
Fixing permissions on existing directory /home/gustav/pgsqldata... ok
creating directory /home/gustav/pgsqldata/base... ok
creating directory /home/gustav/pgsqldata/global... ok
creating directory /home/gustav/pgsqldata/pg_xlog... ok
creating directory /home/gustav/pgsqldata/pg_clog... ok
creating template1 database in /home/gustav/pgsqldata/base/1... ok
creating configuration files... ok
initializing pg_shadow... ok
enabling unlimited row size for system tables... ok
initializing pg_depend... ok
creating system views... ok
loading pg_description... ok
creating conversions... ok
setting privileges on built-in objects... ok
vacuuming database template1... ok
copying template1 to template0... ok
```

Success. You can now start the database server using:

```

/usr/bin/postmaster -D /home/gustav/pgsql/data
or
/usr/bin/pg_ctl -D /home/gustav/pgsql/data -l logfile start
gustav@WOODLANDS:~/gustav 19:48:44 !552 $

```

Let us have a look at what some of the files created by `initdb` look like.

```

gustav@WOODLANDS:~/gustav 19:50:28 !561 $ cd postgresql/data
gustav@WOODLANDS:~/data 19:50:31 !562 $ ls -FC
PG_VERSION  global/    pg_hba.conf  pg_xlog/
base/       pg_clog/   pg_ident.conf postgresql.conf
gustav@WOODLANDS:~/data 19:50:33 !563 $

```

There are three configuration files here and then some directories in which PostgreSQL is going to store data and logs.

The file that defines from which hosts you can connect to the data base server and as who, is `pg_hba.conf`. The header of the file explains in detail how to construct the entries. An example entry may look as follows:

```

# TYPE      DATABASE    USER        IP-ADDRESS   IP-MASK      METHOD
host       biblio      adams       129.79.207.223 255.255.255.0 md5

```

which specifies that a user `adams` is allowed to connect to the data base `biblio` from a host of address `129.79.207.223`, and that the user must be authenticated using the `md5` method. By default all users from the local machine are allowed connections to PostgreSQL servers serving all data bases in the cluster. But ... also by default PostgreSQL will ignore this file altogether and allow data base access to the data base owner only. To change this you will have to run `postmaster` with the `-i` option. But before you do this, you should edit this file very carefully indeed.

File `pg_ident.conf` is used for the user identification based on the `ident` protocol. This protocol is very popular, but not very secure. In order to use this protocol you would have to type `ident` in the `METHOD` field in the `pg_hba.conf` file.

Finally there is the `postgresql.conf` file, which is the PostgreSQL configuration file. It has a lot of parameter definitions, including the port number the server is going to listen on, whether SSL should be used for security, what is the maximum number of connections allowed, what is the maximum number of tables per data base, what language should be used in the data base messages (e.g., Lithuanian), and so on. There is usually little need to change much on this file, unless your data base is going to be really big and accessed very frequently.

Now you can start the `postmaster` directing it to the newly created area. The `postmaster` will refuse to run if the directories it's been pointed to have not been prepared with `initdb`. The program does not return and cannot be talked to from the keyboard, although it logs all it does on standard output. So the right way to run it is to start it in the background and have logging redirected to a file. Here is how I do it:

```

gustav@WOODLANDS:~/gustav 20:10:47 !586 $ postmaster \
-D 'pwd'/pgsql/data > postgresql/data/Nov-16-2003.log 2>&1 &

```

```
[1] 2300
gustav@WOODLANDS:~/gustav 20:10:32 !587 $ jobs
[1]+  Running postmaster -D 'pwd'/pgsql/data >pgsql/logs/Nov-16-2003.log 2>&1 &
gustav@WOODLANDS:~/gustav 20:12:06 !588 $
```

Well, it runs! We can go to the logs directory and see the logging:

```
gustav@WOODLANDS:~/gustav 20:12:06 !589 $ cd pgsq/logs
gustav@WOODLANDS:~/logs 20:12:42 !590 $ ls
Nov-16-2003.log
gustav@WOODLANDS:~/logs 20:12:42 !591 $ cat Nov-16-2003.log
LOG: database system was shut down at 2003-11-16 19:48:43 USEST
LOG: checkpoint record is at 0/83B238
LOG: redo record is at 0/83B238; undo record is at 0/0; shutdown TRUE
LOG: next transaction id: 480; next oid: 16976
LOG: database system is ready
gustav@WOODLANDS:~/logs 20:12:44 !592 $
```

and once we're there we can look at the log continuously with `tail -f`. If you are going to do this, you will need another window, in which to run a client application that talks to the server.

But before we are going to connect to the server let us create an empty data base first. This is done with the command `createdb`. The command is going to connect to the first `postmaster` it finds on the default port. If you run more than one server, you will have to provide `createdb` with more information. But the default will do just fine for us. So let us create a data base called `neighbors`:

```
gustav@WOODLANDS:~/gustav 20:25:27 !595 $ createdb neighbors
CREATE DATABASE
gustav@WOODLANDS:~/gustav 20:25:42 !596 $
```

Well, this worked.

Now we can connect to the data base and do various things with it.

If you want to completely delete the whole data base, issue the command (with caution!):

```
gustav@WOODLANDS:~/gustav 20:37:10 !615 $ dropdb neighbors
DROP DATABASE
gustav@WOODLANDS:~/gustav 20:37:18 !616 $
```

To stop the `postmaster` you can simply type `^C`, i.e., control-C, if the `postmaster` runs in the foreground, or you can send it a kill, if it runs in the background:

```
gustav@WOODLANDS:~/gustav 20:37:18 !616 $ jobs
[1]+  Running                postmaster -D 'pwd'/pgsql/data >pgsql/logs/Nov-16-2003.log 2>&1 &
gustav@WOODLANDS:~/gustav 20:38:20 !617 $ kill %1
gustav@WOODLANDS:~/gustav 20:39:29 !618 $ jobs
[1]+  Done                    postmaster -D 'pwd'/pgsql/data >pgsql/logs/Nov-16-2003.log 2>&1
gustav@WOODLANDS:~/gustav 20:39:37 !619 $
```

There are many methods to connect to a PostgreSQL data base (the `postmaster` must be running, of course). First and foremost PostgreSQL provides a C-language interface. There is a direct interface and an embedded SQL interface. There is also a Tcl interface, a Java interface, and a Python interface. And ... last but not least, a SQL (it is often pronounced sea-quell, hence "a" rather than "an" in front of it) interface and a SQL client called `psql`.

How can you find more about PostgreSQL? Abundant documentation is provided with it. You should find it in the `/usr/doc/postgresql-7.3.4` (the number may be different) on your Cygwin or Linux system. There is an `html` subdirectory there and if you point your browser at the `index.html` file in it, you'll get to view the manuals. There should be at least six of them:

1. PostgreSQL 7.3.4 Tutorial
2. PostgreSQL 7.3.4 User's Guide
3. PostgreSQL 7.3.4 Administrator's Guide
4. PostgreSQL 7.3.4 Programmer's Guide
5. PostgreSQL 7.3.4 Reference Manual
6. PostgreSQL 7.3.4 Developer's Guide
7. Index

In the next section I am going to show you how to talk to PostgreSQL using the SQL client.

## 9.2 Talking to a PostgreSQL Data Base with `psql`

`psql` is the SQL interpreter that talks to PostgreSQL. You can use it to interrogate a PostgreSQL data base interactively or to execute SQL programs, in which case you will need to invoke `psql` with the `-f` option followed by the name of the file that contains the program.

In order to connect to PostgreSQL data base with `psql`, the data base must exist already. The command `initdb` creates two template data bases, which are always there in the PostgreSQL data space. You can see them if you issue the `psql -l` command:

```
gustav@WOODLANDS:~/gustav 12:23:06 !508 $ psql -l
      List of databases
  Name      | Owner  | Encoding
-----+-----+-----
 template0 | gustav | SQL_ASCII
 template1 | gustav | SQL_ASCII
(2 rows)
```

```
gustav@WOODLANDS:~/gustav 12:23:10 !509 $
```

The command `createdb`, I have used in the previous section, is just a shell script, which connects to `template1` and then issues the SQL `create database` command. But it's a very convenient wrapper, and so I'm going to use it again to recreate the data base `neighbors` I have *dropped* in the previous section:

```
gustav@WOODLANDS:~/gustav 12:27:15 !510 $ createdb neighbors
CREATE DATABASE
gustav@WOODLANDS:~/gustav 12:27:31 !511 $
```



Now I can connect to it:

```
gustav@WOODLANDS:../gustav 12:29:26 !513 $ psql neighbors
Welcome to psql 7.3.4, the PostgreSQL interactive terminal.
```

```
Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit
```

```
neighbors=#
```

If the `psql` prompt ends with the hash, #, it means that you are the owner of the data base. Right now the data base doesn't have any tables in it yet. But how to define the tables is the SQL thing and I won't dwell on it too much, because you probably know all about it already. I want to focus more on the specific features of `psql`. Still we'll have to create some data if only to illustrate various features of `psql`. But before we get there, let's have a look at what we can do in `psql` without any tables defined.

First you can ask about `psql` specific commands by typing `\?`:

```
neighbors=# \?
\A          toggle between unaligned and aligned output mode
\c[connect] [DBNAME|- [USER]]
            connect to new database (currently "neighbors")
\C [STRING] set table title, or unset if none
\cd [DIR]   change the current working directory
[ ... lots of various commands listed ... ]
\X          toggle expanded output (currently off)
\z [PATTERN] list table access privileges (same as \dp)
\! [COMMAND] execute command in shell or start interactive shell
```

`psql` specific commands all begin with a backslash. Some important ones are

`\c database` connects to the database

`\d name` describes the named object, it may be a table, a view, an index or a sequence

`\e file` edit the query buffer or a file with an external editor

`\h command` help on syntax of SQL commands

`\i file` executes commands from the file

`\l` lists databases accessible to this `psql` session

`\q` quits `psql`

`\r` reset the query buffer

`\s` display history

`\!` execute a command in shell or start interactive shell

`\h` is used to help you with SQL language commands. If you just type `\h` without any arguments, `psql` will list all available SQL commands in three columns:

```
neighbors=# \h
Available help:
ABORT                CREATE TABLE        EXECUTE
ALTER DATABASE       CREATE TABLE AS     EXPLAIN
ALTER GROUP          CREATE TRIGGER        FETCH
ALTER TABLE         CREATE TYPE           GRANT
ALTER TRIGGER        CREATE USER           INSERT
ALTER USER           CREATE VIEW           LISTEN
ANALYZE              DEALLOCATE            LOAD
BEGIN                DECLARE               LOCK
CHECKPOINT           DELETE                MOVE
CLOSE                DROP AGGREGATE        NOTIFY
CLUSTER              DROP CAST             PREPARE
COMMENT              DROP CONVERSION       REINDEX
COMMIT               DROP DATABASE         RESET
COPY                 DROP DOMAIN           REVOKE
CREATE AGGREGATE     DROP FUNCTION         ROLLBACK
CREATE CAST          DROP GROUP            SELECT
CREATE CONSTRAINT TRIGGER DROP INDEX            SELECT INTO
CREATE CONVERSION    DROP LANGUAGE         SET
CREATE DATABASE      DROP OPERATOR CLASS   SET CONSTRAINTS
CREATE DOMAIN         DROP OPERATOR         SET SESSION AUTHORIZATION
CREATE FUNCTION      DROP RULE              SET TRANSACTION
CREATE GROUP          DROP SCHEMA           SHOW
CREATE INDEX         DROP SEQUENCE         START TRANSACTION
CREATE LANGUAGE      DROP TABLE           TRUNCATE
CREATE OPERATOR CLASS DROP TRIGGER          UNLISTEN
CREATE OPERATOR      DROP TYPE             UPDATE
CREATE RULE          DROP USER             VACUUM
CREATE SCHEMA        DROP VIEW
CREATE SEQUENCE      END
neighbors=#
```

In order to refresh your knowledge on a specific SQL command in more detail simply follow `\h` with the name of the command, e.g.,

```
neighbors=# \h select
Command:      SELECT
Description:  retrieve rows from a table or view
Syntax:
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
* | expression [ AS output_name ] [, ...]
[ FROM from_item [, ...] ]
[ WHERE condition ]
[ GROUP BY expression [, ...] ]
[ HAVING condition [, ...] ]
[ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
[ ORDER BY expression [ ASC | DESC | USING operator ] [, ...] ]
[ LIMIT { count | ALL } ]
[ OFFSET start ]
[ FOR UPDATE [ OF tablename [, ...] ] ]
```

where `from_item` can be:

```
[ ONLY ] table_name [ * ]
    [ [ AS ] alias [ ( column_alias_list ) ] ]
|
( select )
    [ AS ] alias [ ( column_alias_list ) ]
|
table_function_name ( [ argument [, ...] ] )
    [ AS ] alias [ ( column_alias_list | column_definition_list ) ]
|
table_function_name ( [ argument [, ...] ] )
    AS ( column_definition_list )
|
from_item [ NATURAL ] join_type from_item
    [ ON join_condition | USING ( join_column_list ) ]
```

neighbors=#

`select` is the SQL command which extract data from the tables and prints them on standard output. It can be also used to print the output of various predefined functions, which, in turn, can be listed with `\df`. For example, there is a function called `version`. If you just type `\df` you will find this function towards the end of a very long listing. To list just the information about it type:

neighbors=# `\df version`

List of functions			
Result data type	Schema	Name	Argument data types
text	pg_catalog	version	

(1 row)

neighbors=#

or, if you want more:

neighbors=# `\df+ version`

List of functions							
Result data type	Schema	Name	Argument data types	Owner	Language	Source code	Description
text	pg_catalog	version		gustav	internal	pgsql_version	PostgreSQL version

(1 row)

neighbors=#

To see what the function actually does type:

neighbors=# `select version();`

version
PostgreSQL 7.3.4 on i686-pc-cygwin, compiled by GCC gcc (GCC) 3.2 20020927 (prerelease)

(1 row)

neighbors=#

You can also use `select` to view the content of variables, e.g., `current_date` or to perform simple arithmetic:

neighbors=# `select current_date;`

date
-----

```

2003-11-17
(1 row)

neighbors=# select 2 + 5;
?column?
-----
              7
(1 row)

neighbors=#

```

So now let us fill the data base `neighbors` with some data. At this stage we get to use the SQL language. The SQL command `create table` creates an empty table with some predefined columns, which you have to specify at this stage, for example:

```

neighbors=# create table dogs (
neighbors(#   name      varchar(80),
neighbors(#   owner     varchar(80),
neighbors(#   race      varchar(80),
neighbors(#   age       int,
neighbors(#   unit      int
neighbors(# );
CREATE TABLE
neighbors=#

```

Having created the table, we can commence inserting data into it with the SQL command `insert`:

```

neighbors=# insert into dogs (name, owner, race, age, unit)
neighbors-#   values ('Barky', 'Abraham Lincoln', 'Poodle', 3, 2387);
INSERT 16980 1
neighbors=# insert into dogs (name, owner, race, age, unit)
neighbors-#   values ('Fiote', 'Margaret Thatcher', 'Border Collie', 4, 2385);
INSERT 16981 1
neighbors=# insert into dogs (name, owner, race, age, unit)
neighbors-#   values ('Cedric', 'Woodrow Wilson', 'Mongrel', 8, 2383);
INSERT 16982 1
neighbors=# select * from dogs;
  name | owner          | race          | age | unit
-----+-----+-----+-----+-----
Barky | Abraham Lincoln | Poodle       | 3   | 2387
Fiote | Margaret Thatcher | Border Collie | 4   | 2385
Cedric | Woodrow Wilson   | Mongrel      | 8   | 2383
(3 rows)

neighbors=#

```

Let us create another table and populate it with some data at the same time:

```

neighbors=# create table cats (
neighbors(#   name      varchar(80),
neighbors(#   owner     varchar(80),
neighbors(#   race      varchar(80),
neighbors(#   age       int,
neighbors(#   unit      int
neighbors(# );
CREATE TABLE
neighbors=# insert into cats (name, owner, race, age, unit)

```

```

neighbors=# values ('Smokey', 'Abraham Lincoln', 'Tabby', 5, 2387);
INSERT 16985 1
neighbors=# insert into cats (name, owner, race, age, unit)
neighbors=# values ('Minnow', 'Benjamin Franklin', 'Bicolor', 3, 2381);
INSERT 16986 1
neighbors=# insert into cats (name, owner, race, age, unit)
neighbors=# values ('Fluffy', 'Frederic Chopin', 'Siamese', 2, 2385);
INSERT 16987 1
neighbors=#

```

Once we have the data in, we can query the data base against a single table or against two (or more) tables at the same time. The latter usually is where interesting discoveries can be made.

Let us begin with querying a single table:

```

neighbors=# select * from dogs
neighbors=# where race = 'Poodle';
 name | owner | race | age | unit
-----+-----+-----+-----+-----
  Barky | Abraham Lincoln | Poodle | 3 | 2387
(1 row)

```

```
neighbors=#
```

Well, this tells us that my neighbor, Abraham Lincoln, has a poodle named Barky. This query:

```

neighbors=# select * from cats
neighbors=# where age > 2;
 name | owner | race | age | unit
-----+-----+-----+-----+-----
  Smokey | Abraham Lincoln | Tabby | 5 | 2387
  Minnow | Benjamin Franklin | Bicolor | 3 | 2381
(2 rows)

```

```
neighbors=#
```

tells us that there are two cats nearby that are more than 2 years old.

This query against two tables at the same time is more interesting:

```

neighbors=# select dogs.owner, dogs.unit
neighbors=# from dogs, cats
neighbors=# where dogs.owner = cats.owner
neighbors=# and dogs.unit = cats.unit;
 owner | unit
-----+-----
 Abraham Lincoln | 2387
(1 row)

```

```
neighbors=#
```

This tells us that Abraham Lincoln has both a dog and a cat. Such a query made against two tables at the same time is called a *join* query in the data base parlance.

But now we are going to make a much fancier query:

```

neighbors=# select dogs.unit, dogs.owner, cats.owner
neighbors-#   from dogs, cats
neighbors-#   where dogs.unit = cats.unit
neighbors-#   and dogs.owner not like cats.owner;
  unit |      owner      |      owner
-----+-----+-----
  2385 | Margaret Thatcher | Frederic Chopin
(1 row)

```

```
neighbors=#
```

It turns out that Margaret Thatcher cohabits with Frederic Chopin! In my neighborhood! The world must have gone to dogs!

This is an example of *data mining* and startling discoveries that it may yield. Verily I say unto you, great is the power of this technology.

But observe the following. Let us corrupt the `cats` data base and mistype the name of Abraham Lincoln, something that is more than likely to happen in real life.

```

neighbors=# update cats
neighbors-#   set owner = 'Abrahan Lincoln'
neighbors-#   where owner = 'Abraham Lincoln';
UPDATE 1
neighbors=# select * from cats;
  name |      owner      | race | age | unit
-----+-----+-----+-----+-----
Minnow | Benjamin Franklin | Bicolor | 3 | 2381
Fluffy | Frederic Chopin | Siamese | 2 | 2385
Smokey | Abrahan Lincoln | Tabby | 5 | 2387
(3 rows)

```

```
neighbors=#
```

Now let us repeat our earlier query:

```

neighbors=# select dogs.owner, dogs.unit
neighbors-#   from dogs, cats
neighbors-#   where dogs.owner = cats.owner;
  owner | unit
-----+-----
(0 rows)

```

```
neighbors=#
```

As you see, because of the stupid typo, we have just missed the fact that Abraham Lincoln has a dog and a cat – although in this case we could do the query differently, namely:

```

neighbors=# select dogs.owner, dogs.unit
neighbors-#   from dogs, cats
neighbors-#   where dogs.unit = cats.unit;
  owner | unit
-----+-----
Margaret Thatcher | 2385
Abraham Lincoln | 2387
(2 rows)

```

```
neighbors=#
```

and this correctly tells us that there are a dog and a cat living in units 2385 and 2387.

So, this, in short is how you can talk to PostgreSQL using a SQL client, `psql`.

Although in this example I have run `psql` and `postmaster` on the same machine, you would normally have them running on separate machines, reserving a more powerful system, e.g., a research group server, for the `postmaster` and the data directory itself, and running `psql` on, e.g., computational nodes of the AVIDD cluster or on a desktop PC in your office.

### 9.3 Talking to a PostgreSQL Data Base from C

Suppose you have the following project. You have a C-language program that can read documents, e.g., memos, articles, books, and that can generate automatically abstracts, keywords and bibliographic entries for these documents – without any human intervention. If you have a lot of documents to process you may consider running the program on the AVIDD farm. You would simply submit a PBS job that would distribute multiple instances of the program over, say, 32 or 48 nodes. Because the instances of the program have no need to communicate with each other, this is all that you really need to do to “parallelise” the application.

The only snag is coordination. Suppose that all the documents you need to process live on GPFS in one of your directories. How are you to decide, which instance of your program should get which document? The other issue is how you are going to collect the results of the processing. Each instance of the program is going to generate bibliographic entries for the documents it has processed, where are these entries to go?

This kind of a problem is not restricted to generation of bibliographic entries. When high energy physicists process their experimental data, what they do is almost exactly the same. The only difference in the type of calculations their programs do. They store their data on multiple files, which are then distributed, somehow, to instances of data reduction programs that run on a CPU farm. The programs process the data and return the results on various files, which are then collated and prepared for further processing, e.g., visualization.

In order to coordinate the exercise you could write an MPI program and then incorporate your C program as a subroutine in the MPI program. But this is really quite unnecessary, because you already have a first class coordinator and data collector in the form of . . . yes, in the form of a PostgreSQL data base.

PostgreSQL data base transactions, i.e., operations such as `update`, or `insert`, or `delete`, are *atomic*. Various complex combinations of these and other transactions can be made atomic too using a special SQL bracketing in the form of `begin`, `commit` and `rollback` commands. Atomicity of a transaction means that the transaction, however complex and lengthy, runs to completion or not at all. If anything happens while the transaction runs that makes it impossible to complete it, the transaction is unwound and the state of the data base re-

turns to the condition the data base was in before the transaction began. This is accomplished by the means of sophisticated locking, queuing, logging, backtracking and various other devices that would be quite difficult to implement in a small MPI program.

Additionally SQL provides the command `lock`, which lets you lock the table you work on explicitly and in various ways. The way to use this command is to lock the table in the first command line of the transaction. Then you proceed with the transaction and the moment you `commit` it, the lock is automatically lifted. There is no `unlock` command in SQL.

In summary you could begin by constructing a table that would represent the job queue. Your workers, once they've been instantiated by PBS on various nodes of the AVIDD cluster would contact the data base server running on one of the AVIDD nodes or even on your laboratory server and would perform a transaction with the data base. They could, for example, ask the data base for the top entry in the job queue table that has not been worked on yet. The way to ask for such table entry is similar to the following:

```
neighbors=# select * from dogs
neighbors-#   where age not like 3
neighbors-#   limit 1;
  name |      owner      |      race      | age | unit
-----+-----+-----+-----+-----
  Fiote | Margaret Thatcher | Border Collie |   4 | 2385
(1 row)
```

```
neighbors=#
```

Now think of `age = 3` representing a “worked-on” flag and here you have the solution that gives you the next *available* entry in the table. Once the worker gets the entry, the worker would set the “worked-on” flag on this row by doing something like:

```
neighbors=# update dogs
neighbors-#   set age = 3
neighbors-#   where name = 'Fiote' and unit = 2385;
UPDATE 1
neighbors=#
```

Having reserved the row, the worker would then go to process the corresponding document. Then once the bibliographic entry for the document has been constructed, the worker would again contact the data base and it would update the relevant row in this or some other table, by adding the full bibliographic entry to it.

The only thing you need to learn in order to implement your system this way is how to talk to a PostgreSQL data base from your C-language program. The easiest way to do this is to use the `ecpg` preprocessor for C with embedded SQL statements. The preprocessor recognizes the SQL statements and converts them automatically to C-language constructs. The program is then compiled and linked with the `ecpg` library and ... you have it.

Here is a simple example. Consider the following program:

```
gustav@WOODLANDS:~/gustav 18:08:23 !525 $ cat tryme.pgc
#include <stdio.h>
main()
```



```

{
    printf ("Connecting to the data base ... \n");
EXEC SQL CONNECT TO neighbors;
EXEC SQL UPDATE dogs SET age = 7 WHERE unit = 2383 AND name = 'Cedric';
EXEC SQL COMMIT;
EXEC SQL DISCONNECT CURRENT;
    printf ("Done it.\n");
}
gustav@WOODLANDS:~/gustav 18:09:40 !526 $

```

The SQL commands begin with the EXEC SQL string, which is then followed by a command itself and the terminating semicolon. You can interleave SQL commands with normal C-language statements. SQL transactions executed this way are *not* committed unless COMMIT is requested explicitly.

I am going to preprocess this program first in order to generate a genuine C-language program:

```

gustav@WOODLANDS:~/gustav 18:09:40 !526 $ ecpg tryme.pgc
gustav@WOODLANDS:~/gustav 18:10:30 !527 $

```

The C-language program I have just generated looks as follows:

```

gustav@WOODLANDS:~/gustav 18:10:30 !527 $ cat tryme.c
/* Processed by ecpg (2.10.0) */
/* These four include files are added by the preprocessor */
#include <ecpgtype.h>
#include <ecpglib.h>
#include <ecpgerrno.h>
#include <sqlca.h>
#line 1 "tryme.pgc"
#include <stdio.h>
main()
{
    printf ("Connecting to the data base ... \n");
{ ECPGconnect(__LINE__, "neighbors" , NULL,NULL , NULL, 0); }
#line 5 "tryme.pgc"

{ ECPGdo(__LINE__, NULL, "update dogs set age = 7 where unit = 2383 and name
= 'Cedric'", ECPGt_EOIT, ECPGt_EORT);}
#line 6 "tryme.pgc"

{ ECPGtrans(__LINE__, NULL, "commit");}
#line 7 "tryme.pgc"

{ ECPGdisconnect(__LINE__, "CURRENT");}
#line 8 "tryme.pgc"

    printf ("Done it.\n");
}
gustav@WOODLANDS:~/gustav 18:11:02 !528 $

```

Now I have to link this program with the ecpg library:

```

gustav@WOODLANDS:~/gustav 18:11:02 !528 $ gcc -o tryme tryme.c -lecpg
gustav@WOODLANDS:~/gustav 18:11:57 !529 $

```

and I have the binary.

Let me demonstrate to you that it actually works!

```

gustav@WOODLANDS:~/gustav 18:08:18 !559 $ psql --command "select * from dogs;" neighbors
 name | owner | race | age | unit
-----+-----+-----+-----+-----
 Barky | Abraham Lincoln | Poodle | 3 | 2387
 Cedric | Woodrow Wilson | Mongrel | 8 | 2383
 Fiote | Margaret Thatcher | Border Collie | 3 | 2385
(3 rows)

gustav@WOODLANDS:~/gustav 18:08:25 !560 $ ./tryme
Connecting to the data base ...
Done it.
gustav@WOODLANDS:~/gustav 18:08:32 !561 $ psql --command "select * from dogs;" neighbors
 name | owner | race | age | unit
-----+-----+-----+-----+-----
 Barky | Abraham Lincoln | Poodle | 3 | 2387
 Fiote | Margaret Thatcher | Border Collie | 3 | 2385
 Cedric | Woodrow Wilson | Mongrel | 7 | 2383
(3 rows)

gustav@WOODLANDS:~/gustav 18:08:37 !562 $

```

Also, observe the use of the `--command` (or `-c`) command line switch to `psql`. It lets you execute `psql` commands without actually entering an interactive session with it.

In the example above I have performed certain SQL action, but I have not really interfaced SQL variables with C variables and this, clearly, is going to be necessary if you want to do anything more elaborate with your C program. `ecpg` lets you declare variables that are going to be used both in the SQL and C parts of the program. This is done in the following way, e.g.,

```

EXEC SQL BEGIN DECLARE SECTION;
    int x;
    char foo[16], bar[16];
EXEC SQL END DECLARE SECTION;

```

But here things may get tricky, e.g., if you want to capture the output of a `select` operation into a C-language string. The `ecpg` documentation is not very clear on this point.

So here you may wish to turn to the full C-language interface, called `libpq`, which is also provided with PostgreSQL. The full interface is, of course, much more complex and more difficult to use than `ecpg`, but here you will have no problems with capturing the output of `select`. There is the whole section, “1.3.4. Retrieving SELECT Result Information”, in the “PostgreSQL 7.3.4 Programmer’s Guide, I. Client Interfaces” that talks about it. This specific section’s location on my PC is

```
C:\cygwin\usr\doc\postgresql-7.3.4\html\libpq-exec.html
```

Briefly speaking, there is a function `PQexec`, which is used to submit queries to the data base. The function returns the result as an object of type `PGresult`. This object is really a table. It can be queried with various functions provided by `libpq` in order to find about the number of rows and columns, column names, column types, the actual values in the fields of the table. There is also a special function that can be used to print the returned table on a file – including standard output.

The same manual, i.e., the Programmer's Guide, provides example programs. On my PC the examples live in:

```
C:\cygwin\usr\doc\postgresql-7.3.4\html\libpq-example.html
```

and they are quite invaluable if you want to make sense of it all.

If you are a Java programmer, then you'll find Java interface to PostgreSQL discussed in Chapter 5 of the Programmer's Guide, still part I, Client Interfaces. This chapter lives on my PC in

```
C:\cygwin\usr\doc\postgresql-7.3.4\html\jdbc.html
```

This chapter is also illustrated with some examples.

## 9.4 Huge Data Bases

If your data base is going to be tens of GBs large, perhaps even hundreds of GBs, then PostgreSQL will not be enough. In this case you may have to process the data base on multiple CPUs in parallel and PostgreSQL is not a parallel data base.

You will find an example of a very powerful parallel data base described in the article "DB2 Parallel Edition for AIX: Concepts and Facilities".

We may provide parallel DB2 to AVIDD users on specific request, but it is a very large and a very involved system. Think twice about making such a request. Yet, if you have a good justification for it, don't hesitate. This is what the AVIDD cluster is for.

What is a "good justification"? Data bases do not really live entirely on the disk, when the system is up. A data base server always tries to load as much of it into memory as possible, because otherwise responding to queries would be too slow. So you should think of an active data base as living partially on the disk and partially in memory. A very small data base may live in the memory of a computer entirely, with only occasional updates and logs written to the disk. But a very large data base, whose size is well in excess of the available memory floor will have to live on the disk mostly.

Computers still have rather limited memory, because memory is expensive. I can't wait for the day when memory will be so cheap and persistent that the disks, which are clumsy, mechanical and energy hungry devices, will no longer be necessary. Disks slow down computing enormously. They are also a primary break down point. The computational nodes of the AVIDD cluster have 2GBs memory each, of which you can probably get about a GB for your program, the rest being taken by the operating system, PBS, and a "safety margin". This means that you can probably serve a data base with up to 4 times the size, i.e., a 4GB data base from a single node. This assumes that you should have a 25% memory foothold, which is fairly reasonable.

Well, what then if the data base is 40GBs? Then, you should think about distributing it over 10 AVIDD nodes for truly satisfactory performance. If the nodes were more powerful and, just as importantly, if they had more memory, you could run a larger data base from 10 nodes. Consider a system of 4-way

SMPs, for example, each SMP with, say, 8GBs memory, of which 1GB is reserved for the OS. This would give you a 7GB foothold per node, and then you could run a 28GB data base from a single node, or 280GB data base from 10 such nodes. This would be nice and this is how large commercial data bases are usually configured. But let us get back to AVIDD.

You should not expect that the performance of a parallel data base will scale linearly forever. There is a lot of synchronization and communication involved in the data base functioning, and so data bases running on more than 64 CPUs are pretty rare. This could translate into, say, a system of 16 4-way SMPs, or into a single powerful SMP such as the Sun Starfire or HP Superdome.

Since the AVIDD nodes are 2-way SMPs you could think of deploying a data base on up to 32 of these. But the AVIDD nodes have little memory and cannot have much more, because they are 32-bit systems, and so even 32 nodes would probably be too many. But 16 to 24 nodes should be quite OK. This would give you a data base of up to 96GBs, which is plenty as far as data bases goes and more than anything I have seen at Indiana University so far, and more than the largest file we have ever moved to our HPSS (although HPSS operators at some DoE laboratories deal routinely with files that are larger than a TB).

Remember what I said right at the beginning of this chapter: data bases are normally populated by human typists. 96GBs would roughly translate into nearly 30 million pages of text, assuming 60 characters per line and 60 lines per page. Assuming 300 pages per book, this would be 100,000 books. You'd need a lot of typists to type this many pages!

DB2 Parallel Edition is a data base that is designed to live on a cluster. Nowadays we actually no longer use the expression "Parallel Edition", because every new version of DB2 "Enterprise Edition" can be used in this way. It's simply one of the installation options.

Parallelism can manifest itself in various ways on such a system.

The first and the most obvious is *inter-transaction parallelism*. Transactions don't always get into each other's way. If they don't, then they can be executed in parallel on separate nodes. Any SQL lookup with the `select` function doesn't alter the tables, and so you can have multiple lookups going on at the same time. If updates are performed on separate parts of the table, they can be performed in parallel too. Inter-transaction parallelism is especially welcome on relatively small data bases that are accessed very frequently.

A less trivial form of parallelism is *intra-query parallelism*. Here a single query may get split across many processors. This type of parallelism would be used on a very large data base, where a single table may have been striped over multiple nodes. Intra-query parallelism can be implemented in various ways too, for example as *partition parallelism* or *query decomposition*, which is the same thing, or *pipelined parallelism* where data may flow between processes in a pipeline.

Then we may classify parallel operations depending on how we deal with data. In some cases we may ship *functions* from one process to where the data itself resides, in others we may have to fetch the data from other processes and feed them into a function that runs on a select processor. Function shipping

is usually preferable, especially on clusters. Functions are data too, as every Lisp programmer knows, but they are usually rather small compared to the size of data they operate on in large data bases. So it's usually cheaper to send a function across than the whole table or a portion of it.

The architecture of DB2 Parallel Edition is pretty complex. There is a system controller there, a watch dog, a parallel system controller, a communication manager, a master data base logger, a deadlock detector (sic!) and numerous other agents. You will normally have several DB2 listeners running on various nodes, so that connections made to the data base can be distributed over the cluster in order to level the load on the whole system. There is a central agent that maintains maps of how tables are distributed amongst the nodes, but even this function may well be replicated for better performance. In this respect the functioning of DB2 is likely to be quite similar to the functioning of, e.g., the AFS.

The parallel DB2 supports pretty much normal SQL and adds a couple of extensions. Most operations are quite transparent and you don't need to think of parallelism. What is parallelizable DB2 will parallelize automatically (business people wouldn't put up with all the parallel programming fuss otherwise!). In some cases operations will run locally on the nodes. When some central action is required, then the operations may be forwarded to the coordinator node. This is somewhat similar to collective operations in MPI, e.g., `MPI_Reduce` or `MPI_Gather`. There is always a coordinating node there, even if the operation is of the type `MPI_Allreduce`, in which case the node is not explicit, but it is still there.

Parallelism comes to the surface at the level of SQL optimization. Here the optimizer has been extended to generate parallel *plans*. The optimizer then selects the lowest-cost plan of a query. The cost depends on how the table has been partitioned amongst the nodes, on table statistics, on index statistics, data base configuration parameters, and on the SQL query itself.

The actions of the data base can be scrutinized with various so called *explain* tools. These will usually explain why the optimizer made its choices for the query, and how a particular operation is going to be executed. Explain reports can be very detailed, which is important if you work with a huge commercial data base and need to optimize its every aspect.

DB2 provides numerous utilities for executing commands on multiple nodes, managing segments, splitting data, load leveling, adding and dropping nodes, redistributing tables, data base back up and restore, data base recovery, and data base directory utility.

IBM DB2 is a very flexible and quite universal tool. It is used not only in business, but increasingly nowadays in science, engineering, and even in computer management itself. For example, the HPSS data base is DB2 (as of version 5.1 and higher). Another example is the Blue Gene/L supercomputer, which is configured and managed by DB2.

Even though data base pundits predicted a slow demise of relational data bases many years ago, to be replaced, in their opinions, by object oriented data bases, this has never happened. The reason for this is that a table is actually

a very universal device and with suitable extensions it can be used to handle a great variety of data and queries. For example, you may think of storing whole images, even videos as table entries, why not?

Because data bases hold so much useful information, surprising discoveries can sometimes be made by mining this information, looking for unusual and unexpected correlations amongst the millions of rows and columns. Such activities are very difficult and time consuming. Ready-made data mining tools can be purchased from data base companies. Such tools are extremely expensive and people who can operate them skillfully and effectively command extremely high salaries.

Perhaps one day, you'll find yourself amongst them!

# Index

- TEX, 380
- account, 28
- AFS, 21
  - fileset, 280
- ALICE, 374
- Andrew File System, 21
- ANL, 39
- Arctic Region Supercomputer Center, 14
- ARSC, 14
  - Cray X1, 24
- AVIDD, 7, 16
  - GPFS, 36
    - mounting point, 37
    - performance, 37
    - permissions, 37
  - NFS, 36
    - server nodes, 36
  - WWW page, 35
- Birman, Ken, 107
- Boehlert, Sherwood, 15
- C
  - arrays versus pointers, 276
  - case, 43
  - chmod, 136
  - clock, 88
  - CLOCKS\_PER\_SEC, 88
  - command line
    - reading, 44
  - exit, 43
  - exit status, 45
  - fclose, 45
  - feof, 75
  - ferror, 75
  - fopen, 44
  - for, 16, 44
  - fread, 75
  - fscanf, 95
  - fwrite, 44
  - getenv, 94, 142
  - gethostname, 119
  - getopt, 43, 184
  - if, 44
  - INT\_MAX, 75
  - long long, 75
  - macros, 75
  - malloc, 50, 75
  - mkdir, 143
  - optarg, 43
  - perror, 44
  - rand, 44
  - rename, 95
  - sizeof, 44
  - srand, 44
  - sscanf, 43, 50, 75
  - stat, 142
    - S\_ISDIR, 142
  - strcat, 95
  - strcpy, 95
  - switch, 43
  - time, 87
    - wrap around, 88
  - while, 43, 75
- CAE XSH convention, 88
- XSI conformance, 88
- capacity computing, 7, 24
  - car crash analysis, 24
  - distributing jobs, 8
  - SETI, 24
- CEA, 39
- CERN, 23

- Chameleon, 108
- checkpointing, 86
- ChemIO, 375
- Clemson University, 37
- clusters, 8
- competition
  - lack of in IT, 11
- Congress
  - supercomputer hearing, 15
- Connection Machine, 17
- Cornell University, 107
- Cray, 13
  - Cascade, 13
  - Red Storm, 14
  - T3E, 154
  - Tera, 13
  - Unicos
    - checkpointing, 86
  - X1, 14, 144, 154
    - at ARSC, 14
- Cygwin, 29
  - bash, 30
  - cygrunsrv, 380
  - Dfs, 30
  - download, 30
  - Emacs, 33
  - installation, 30
  - NTFS, 29
    - ACLs, 29
  - OpenSSH, 29, 34
    - authorized\_keys, 32
    - DSA keys, 32
    - passphrase, 32
    - slogin, 31
    - ssh-add, 33
    - ssh-agent, 32, 33
    - ssh-keygen, 31
  - SSH2, 34
  - X11R6, 33
    - startxwin.sh, 33
    - starxwin.sh, 33
    - xclock, 33
    - XWin, 33
- DARPA
  - High Productivity Computing Systems Program, 13
- data bases
  - and business, 377
  - and science, 378
  - as client-server systems, 377
  - DB2, 378
    - architecture, 397
    - coordinator node, 397
    - explain tools, 397
    - functions as data, 397
    - on AVIDD, 395
    - Parallel Edition, 36, 380, 395
    - SQL, 397
    - utilities, 397
  - epidemiology, 379
  - high energy physics
    - SLAC, 379
  - Ingress, 378
  - medical information systems
    - Regenstrief Institute, 379
    - National Library, 379
  - Oracle, 378
  - personal
    - list of citations, 380
  - PostgreSQL, 378
    - C interface, 394
    - createdb, 383
    - documentation, 384
    - ecpg, 392
    - Embedded SQL preprocessor, 392
    - initdb, 381
    - IPC daemon, 380
    - Java interface, 395
    - libpq, 394
    - pg\_hba.conf, 382
    - pg\_ident.conf, 382
    - postgres.conf, 382
    - postmaster, 382
    - psql, 383, 384
    - templates, 384
  - SQL
    - atomicity, 391
    - begin, 391
    - commit, 391



- create database, 384
- create table, 388
- delete, 391
- insert, 388, 391
- lock, 392
- rollback, 391
- select, 387
- update, 390, 391
- SQL Server, 378
- storage methods, 377
- tables, 377
- data parallel programming
  - scalability, 20
- data parallelism, 16
- deadlock, 19
- Dell, 379
- disk IO
  - caching, 38
  - data compression, 38
  - writes versus reads, 38
- Distributed Storage Systems Groups,
  - 57
- DSSG, 57
- Dynix, 19
  
- Earth Simulator, 13, 24
- ECMWF, 39
- Emacs
  - default.el, 35
  - info, 35
- Express, 107
  
- FLIC, 375
- fork, 19
- Fortran 90, 17
  - and MPI, 19
  - and shared memory, 19
- Fox, Geoffrey, 107
- FTP, 29
- fuddy daddies, 15
- Fujitsu, 12
  - AP-3000, 154
  
- Gateway, 8
- girlfriend
  - upset, 24
  
- GNU
  - compilers, 34
  - info, 35
- GPFS
  - ACLs, 34
  - documentation, 34
  - IO
    - dependence on block size, 50
    - memory caching, 37
    - on AVIDD, 36
- Grid, 20
  - .NET Password as grid, 21
  - AFS as grid, 21
  - and distributed computing, 21
  - Globus, 20
  - Legion, 21
  - Unicore, 21
- Gropp, William, 108
  
- HDF4, 279
- HDF5, 279
  - and Cactus, 279
  - and Chombo, 279
  - and Flash, 279
  - and Intel Array Visualizer, 280
  - and Java, 372
  - and LabVIEW, 280
  - and MPI-IO, 302
  - and Swarm, 279
  - and Vis5D, 279
  - constants
    - H5F\_ACC\_EXCL, 287
    - H5F\_ACC\_RDONLY, 292
    - H5F\_ACC\_RDWR, 292
    - H5F\_ACC\_TRUNC, 287
    - H5FD\_MPIO\_COLLECTIVE,
      - 338
    - H5FD\_MPIO\_INDEPENDENT,
      - 338
    - H5P\_DATASET\_CREATE, 302
    - H5P\_DATASET\_XFER, 302,
      - 325
    - H5P\_DEFAULT, 287, 290, 302
    - H5P\_FILE\_ACCESS, 302
    - H5P\_FILE\_CREATE, 302
    - H5S\_SELECT\_AND, 332

- H5S.SELECT\_NOTA, 332
- H5S.SELECT\_NOTB, 332
- H5S.SELECT\_OR, 332
- H5S.SELECT\_SET, 332
- H5S.SELECT\_XOR, 332
- H5S.UNLIMITED, 289
- H5T.COMPOUND, 365
- H5T.ENUM, 366
- H5T.NATIVE\_DOUBLE, 366
- H5T.NATIVE\_FLOAT, 366
- H5T.NATIVE\_INT, 293, 366
- H5T.OPAQUE, 366
- H5T.STD\_I32BE, 281
- H5Z.FILTER\_FLETCHER32, 322
- Folk, Michael J., 280
- Frequently Asked Questions, 280
- gif2h5, 372
- h4toh5, 372
- h52gif, 372
- H5A.close, 296
- H5A.create, 296
  - attribute creation property list, 296
- H5A.write, 296
- H5c++, 372
- H5D.close, 290
- H5D.create, 290
  - dataset creation property list, 290
- H5D.get\_create\_plist, 317
- H5D.get\_space, 315
- h5diff, 372
- H5D.open, 293
- H5D.read, 293
- h5dump, 284
- H5D.write, 293
- h5fc, 372
- H5F.close, 287
- H5F.create, 287
  - file access property list, 287
  - file creation property list, 287
- H5F.open, 292
  - file access flag, 292
  - file access property list, 293
- H5G.close, 298
- H5gen, 372
- H5G.get\_objinfo, 371
- H5G.iterate, 367
- h5import, 372
- h5ls, 282
- H5P.create, 305
- h5perf, 372
- H5P.get\_chunk, 317
- H5P.get\_edc\_check, 325
- H5P.get\_layout, 317
- H5P.set\_chunk, 314
- H5P.set\_deflate, 322
- H5P.set\_dxpl\_mpio, 338
- H5P.set\_fapl\_family, 307
- H5P.set\_fapl\_mpio, 310
- H5P.set\_filter, 324
- H5P.set\_sizes, 306
- H5R.create, 372
- h5repart, 372
- H5S.close, 290
- H5S.copy, 359
- H5S.create\_simple, 289
- H5S.get\_simple\_extent\_dims, 317
- H5S.get\_simple\_extent\_ndims, 317
- H5S.select\_elements, 355
- H5S.select\_hyperslab, 315, 332
- H5T.commit, 370
- H5T.create, 365
- H5T.insert, 366
- h5toh4, 372
- H5View, 372
- HOFFSET, 366
- JHI5, 372
- MPI-IO
  - hyperslab selection, 325
- object reference, 371
- on AVIDD, 279
- Reference Manual, 280
- Tutorial, 280
- types
  - H5T.STD\_I32BE, 290
  - herr\_t, 288
  - hid\_t, 287
  - hsize\_t, 290
- User's Guide, 280
- versus data bases, 286

- versus flat files, 285
- VisAD, 372
- High Performance Fortran, 18
- High Performance Storage System,
  - 39
- Hitachi, 12
- hogs, 88
- HPC, 7
- HPF, 18
  - and MPI, 18
  - on AVIDD, 158
  - versus MPI, 158
- HPSS, 36, 39
  - and DB2, 397
  - pftp, 50
    - client, 52
    - kerberized client, 50
  - users, 39
- I-Light, 7
- I-light, 22
- IA32, 7
- IA64, 7
- IBM, 7, 8, 13
  - Blue Gene/L
    - and DB2, 397
  - BlueGene/L, 193
  - HPSS, 7
  - PERCS, 13
  - SP
    - at Indiana University, 15
  - Watson Laboratory
    - and MPI-IO, 192
- ICRR, 39
- IEEE-1003, 87
- IN2P3, 39
- India, 23
- Indiana University, 15, 39
  - AVIDD, 16
  - IBM SP, 15
    - GPFS, 16
  - LAM MPI, 18
- IRAF, 23
- ISIS, 107
  - and Microsoft Wolfpack, 107
  - virtual synchrony, 107
- Jacobi, Karl Gustav Jacob, 144
- KEK, 39
- Kerberos, 51
  - kdestroy, 52
  - kinit, 51
  - klist, 51
- KISTI, 39
- LAM, 107
  - MPI, 108
- LANL, 39
- Laplace
  - equation, 143
    - boundary conditions, 144
    - discretized, 144
    - Jacobi iterations, 144
  - operator, 143
- Linux, 9
- LLNL, 39
- Lusk, Ewing, 108
- man
  - configuration file, 34
- Mathematica, 24
- Matlab, 24
- message passing, 18
- Message Passing Interface, 18
- Microsoft, 107
  - .NET Password, 21
  - Wolfpack, 107
  - Word, 380
- MPI, 18, 107
  - and multi-threading, 20
  - Argonne implementation, 35
  - C++, 109
  - collective operations, 124
  - communicator, 109
    - error handler, 190
    - group, 190
  - constants
    - MPI\_2INT, 226
    - MPI\_ANY\_SOURCE, 125
    - MPI\_ANY\_TAG, 125
    - MPI\_BYTE, 225
    - MPI\_CHAR, 125, 225

- MPLCOMM\_NULL, 154
- MPLCOMM\_SELF, 222
- MPLCOMM\_WORLD, 119
- MPLDISTRIBUTE\_BLOCK, 232, 240
- MPLDISTRIBUTE\_CYCLIC, 232, 246
- MPLDISTRIBUTE\_DFLT\_DARG, 232, 241
- MPLDISTRIBUTE\_NONE, 232
- MPLDOUBLE, 130, 225
- MPLDOUBLE\_INT, 226
- MPLERRORS\_ARE\_FATAL, 190
- MPLERRORS\_RETURN, 190
- MPLFLOAT, 225
- MPLFLOAT\_INT, 226
- MPLINFO\_NULL, 202
- MPLINT, 225
- MPLLB, 226, 227
- MPLLONG, 225
- MPLLONG\_DOUBLE, 225
- MPLLONG\_DOUBLE\_INT, 226
- MPLLONG\_INT, 226
- MPLLONG\_LONG, 225
- MPLLONG\_LONG\_INT, 225
- MPLMAX\_INFO\_KEY, 247
- MPLMAX\_INFO\_VAL, 247
- MPLORDER\_C, 232, 241
- MPLORDER\_FORTRAN, 232
- MPLSHORT, 225
- MPLSHORT\_INT, 226
- MPLSUCCESS, 190
- MPLSUM, 130
- MPLUB, 226, 227
- MPLUNSIGNED, 225
- MPLUNSIGNED\_CHAR, 225
- MPLUNSIGNED\_LONG, 225
- MPLUNSIGNED\_SHORT, 225
- MPLWCHAR\_T, 225
- data types, 109
- debugging programs with fprintf, 136
- file handle, 202
- Fortran 90, 109
- instrumenting a program, 254
- LAM MPI, 18, 35, 108
- MPE, 253
  - alog file, 255
  - automatic logging, 254
  - clog file, 254
  - clog2alog, 255
  - clogTOSlog2, 256
  - jumpshot, 256
  - MPE\_Close\_graphics, 265
  - MPE\_Describe\_event, 261
  - MPE\_Describe\_state, 259
  - MPE\_Draw\_circle, 264, 265
  - MPE\_Draw\_line, 265
  - MPE\_Draw\_point, 265
  - MPE\_Draw\_string, 264, 265
  - MPE\_Finish\_log, 261
  - MPE\_Init\_log, 259
  - MPE\_Log\_event, 260
  - MPE\_Log\_get\_event\_number, 259
  - MPE\_Make\_color\_array, 265
  - MPE\_Num\_colors, 265
  - MPE\_Open\_graphics, 263, 265
  - MPE\_Start\_log, 260
  - MPE\_Update, 264, 265
  - slog2 file, 256
- MPI-2, 109
- MPI-IO, 108, 109
- MPIAbort, 191, 199, 342
- MPIAddress, 228
- MPIAllgather, 167
- MPIAllgatherv, 168
- MPIAllreduce, 178, 185, 203
- MPIBarrier, 143, 260
- MPIBcast, 124
- MPICart\_coords, 155
- MPICart\_create, 153
- MPICart\_shift, 155
- MPIComm, 175
- MPIComm\_create, 176
- MPIComm\_free, 179
- MPIComm\_group, 176
- MPIComm\_rank, 110, 119, 154
- MPIComm\_Size, 119
- MPIComm\_size, 110
- MPIDims\_create, 232, 241

- MPI\_Errhandler\_set, 190
- MPI\_Error\_class, 191
- MPI\_Error\_string, 190
- MPI\_File\_close, 203
- MPI\_File\_Delete, 215
- MPI\_File\_delete, 223
- MPI\_File\_get\_info, 250
- MPI\_File\_get\_position, 202
- MPI\_File\_get\_size, 212
- MPI\_File\_iread, 253
- MPI\_File\_iwrite, 253
- MPI\_File\_open, 201
- MPI\_File\_read, 214
- MPI\_File\_read\_all, 232
- MPI\_File\_seek, 202
- MPI\_File\_set\_errhandler, 192
- MPI\_File\_set\_info, 251
- MPI\_File\_set\_view, 229, 232
- MPI\_File\_write, 203
- MPI\_File\_write\_all, 232, 244, 344
- MPI\_Finalize, 110, 118, 142, 342
- MPI\_Gather, 167, 168
- MPI\_Get\_address, 227, 366
- MPI\_Get\_count, 203
- MPI\_Get\_processor\_name, 119
- MPI\_Group, 176
- MPI\_Group\_excl, 176
- MPI\_Group\_free, 176
- MPI\_Info\_create, 247
- MPI\_Info\_get, 247, 250
- MPI\_Info\_get\_nkeys, 247, 250
- MPI\_Info\_get\_nthkey, 247, 250
- MPI\_Info\_set, 247, 251
- MPI\_Init, 110, 118
  - and the command line, 269
- MPI\_Irecv, 252
- MPI\_Isend, 252
- MPI\_Recv, 109, 125, 203
  - status, 138
- MPI\_Reduce, 130, 260
- MPI\_Request, 252
- MPI\_Send, 109, 125
- MPI\_Sendrecv, 157
- MPI\_Ssend, 252
- MPI\_Status, 203
- MPI\_Test, 252
- MPLType\_commit, 157, 167
- MPLType\_contiguous, 166, 228
- MPLType\_create\_darray, 230–232, 240, 241
- MPLType\_create\_struct, 227
- MPLType\_extent, 232, 241
- MPLType\_indexed, 228
- MPLType\_size, 232, 241
- MPLType\_struct, 227, 366
- MPLType\_vector, 157, 228
- MPLWait, 252
- MPLWaitall, 271
- MPLWin\_set\_errhandler, 192
- MPLWtime, 129, 142
- MPICH, 18, 35, 108
- MPICH2, 108
  - .mpd.conf, 111
  - mpd.hosts, 115
  - MPD\_USE\_USER\_CONSOLE, 111, 115
  - mpdboot, 115
  - mpdringtest, 116
  - mpds, 115
  - mpicc, 121
  - mpiexec, 116
- Ohio Supercomputer Center implementation, 35
- on AVIDD, 111
- process rank, 109
- programming examples, 110
- ROMIO, 36, 108
- Standard 1.1, 108
- Standard 2.0, 108
- synchronization
  - cost of, 124
- The Complete Reference, 108
- Using MPI, 108
- Using MPI-2, 108
- windows, 192
- N-cube, 107
- NASA, 39, 108
- NCDC, 39
- NCEP, 39
- NCSA, 14, 22, 37
- NEC, 12

- Cenju-3, 154
- Earth Simulator, 13
- NEOS, 375
- NetMeeting, 9
- Nitzberg, Bill, 109
- NSF, 7, 15
- Ohio Supercomputer Center, 107
- OpenMP, 19
- ORNL, 39, 107
- P4, 107, 108
- parallel programming
  - cost of communication, 20
  - scalability, 20
- parallelism
  - data, 16
- PBS
  - PBS\_NODEFILE, 115
  - qdel, 66
  - qhold, 65
  - qrls, 65
  - qsig, 65
  - qstat, 59, 65
    - f, 85
    - q, 59
  - qsub, 62
    - C, 70
    - I, 57, 67
    - M, 71
    - S, 70
    - V, 82
    - W, 84
    - e, 70
    - j, 79
    - m, 71
    - o, 69
    - q, 70
    - v, 82
  - submitting jobs interactively, 70
  - submitting X11 jobs, 68
  - using here-input in scripts, 69
- qusb
  - V, 115
- PC, 11
  - cluster of, 12
  - memory bottleneck, 12
- PETSc, 375
- PFLOPS, 13
- PGHPF, 18
- PICL, 107
- PIM, 13, 17
- Pittsburgh Supercomputer Center, 14
- Portland Group, 18
- POSIX, 87
- Processor in Memory, 17
- processor in memory, 13
- PSC, 37
- Ptools, 375
- PVFS, 37
- PVM, 107
- Python, 115
- Regenstrief Institute, 379
- RIKEN, 39
- rlogin, 29
- RSL, 375
- SAN, 37
- Sandia, 39
- SC2003 conference, 377
- Scarafino, Vincent, 15
- SDSC, 15, 37, 39
- secure shell, 29
- security bugs, 44
- shared memory, 18
- skills, 8
- Skjellum, Anthony, 108
- skulker, 8
- SLAC, 379
- SMP, 18
- Snir, Marc, 109
- spawn, 18
- speed of light, 22
- Stewart, Craig, 35
- Storage Area Network, 37
- Sun, 13
  - clockless computing, 13
- supercomputing, 7
- Supercomputing 92, 107

- TCGMSG, 107
- telnet, 29
- Thakur, Rajeev, 108
- this course
  - rationale for, 25
- threads, 18
- Trilab, 39
- Turner, George W. M., 28
  
- University of Michigan, 15
- University of Stuttgart, 39
- University of Texas, 15
- UNIX
  - BSD
    - fee per campus, 378
  - chmod, 244
  - install, 46
  - INT\_MAX, 242
  - IO caching, 37
    - writes versus reads, 38
  - make, 45
    - Makefile, 45
  - RCS, 46
    - co, 47, 50
    - rcsclean, 47
  - shell
    - io redirection, 83
  - signals, 66
    - catching, 66
  - sleep, 64
  - tar, 285
  - time, 79, 90
  - troff, 380
- US supercomputer industry
  - near death experience, 12
  
- Windows, 9
  - 2000, 9
  - Dfs, 30
  - NTFS, 29
  - XP, 9
  
- X11, 34
  - mkdirhier, 46
  
- year 2000 bug, 88
- year 2038 bug, 88