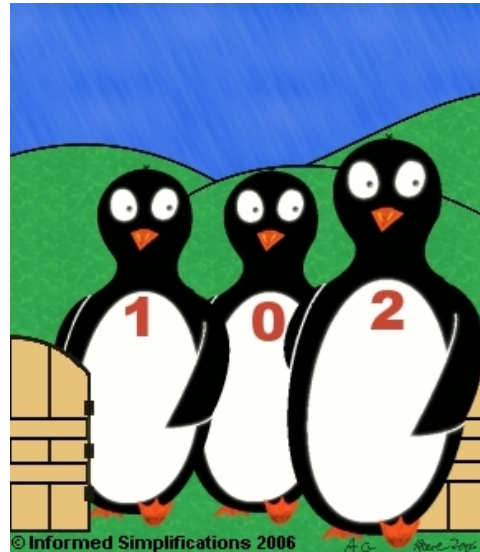
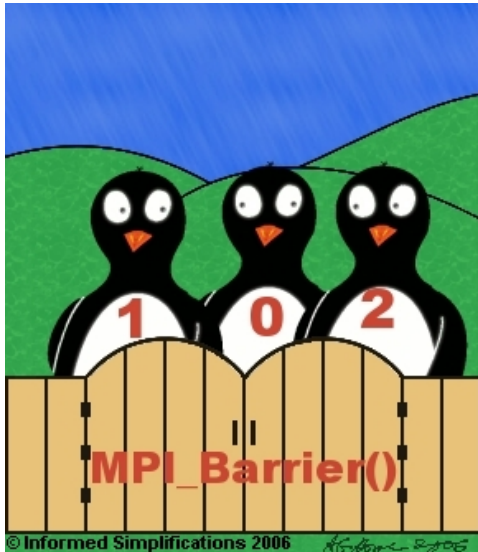


# Supercomputing Simplified

The Bare Necessities for Parallel C Programming with MPI



William B Levy and Andrew G. Howe

Table of Contents

<b>Preface.....</b>	<b>4</b>
<i>Who Should Read This Booklet?.....</i>	<i>4</i>
Why the Message Passing Interface, MPI?.....	5
<i>How to Use This Booklet.....</i>	<i>6</i>
Computer Science Terminology.....	6
Not Computer Science Terminology.....	7
<i>Software and Hardware Requirements.....</i>	<i>8</i>
<i>Legal Information.....</i>	<i>9</i>
<b>Chapter 1.....</b>	<b>1</b>
<i>Compiling and Executing an MPI-based Program.....</i>	<i>1</i>
A. Activating the Message Passing Daemon.....	1
B. Executing a Compiled C/MPI Program.....	3
C. Compiling a C/MPI Program.....	3
D. Using a C/MPI Template for Programming.....	4
E. Running Multiple Processes on a Single CPU.....	5
F. The Four MPI Commands of Chapter 1 (optional reading).....	6
G. Using Process Rank.....	7
Code Appendix : ch1.1.cpp - The MPI Template.....	10
Code Appendix : ch1.2.cpp - Template with Three printf()'s .....	11
Code Appendix : ch1.3 .cpp - Generate Process Specific Array Values.....	12
Code Appendix : New Function - printf_oneP_nobarrier().....	13
<b>Chapter 2.....</b>	<b>14</b>
<i>Using MPI_Barrier() To Enforce Parallel Execution.....</i>	<i>14</i>
A. Introduction.....	14
B. Collective Commands, Barriers, and Parallel Computation.....	15
C. MPI_Barrier() is the Simplest Collective Command.....	16
D. How to Achieve Approximate Synchronization Across Processes.....	17
E. Installing MPI_Barrier() into the source code.....	19
F. Summing up.....	24
Code Appendix : ch2.1.cpp - Chapter 2 Template.....	26
Code Appendix : ch2.2.cpp - One Barrier Improves Output .....	27
Code Appendix : ch2.3.cpp - Two Barriers Pause the Program for Input.....	28
Code Appendix : New Function - keyb_entry_nonneg_int().....	30
<b>Chapter 3.....</b>	<b>31</b>
<i>Transmitting Data Between Processes with MPI_Bcast().....</i>	<i>31</i>
A. Sending the Keyboard Entry to All Processes.....	31
B. Parsing the Arguments of MPI_Bcast( ).....	33
C. Sending a Data Array with MPI_Bcast( ).....	34
D. A Function To Order Your Output, ourmpi_printf( ).....	37
E. Summary.....	41
F. Answers for Chapter 3.....	42

Code Appendix : ch3.1.cpp - ch2.4.cpp Revised.....	43
Code Appendix : ch3.2.cpp - Broadcast Keyboard Entry.....	44
Code Appendix : ch3.3.cpp - Broadcasting a Data Array.....	45
Code Appendix : ch3.4.cpp - Fully Ordered Printing of ch3.2.cpp.....	46
Code Appendix : ch3.5.cpp - Fully Ordered Printing of ch3.3.cpp.....	47
Code Appendix : New Function - ourmpi_print( ).....	48
<b>Chapter 4.....</b>	<b>49</b>
<i>Transmitting Data Between Processes with MPI_Scatterv( ).....</i>	<i>49</i>
A. Comparing MPI_Scatterv() to MPI_Bcast().....	49
B. Sending a Partitioned Data Array.....	52
C. A Better Program for Partitioning.....	56
D. Summary.....	58
E. Answers for Chapter 4.....	58
Code Appendix : ch4.1.cpp - Replacing MPI_Bcast from ch3.3.cpp with MPI_Scatterv().	59
Code Appendix : ch4.2.cpp - Partitioning & Transmitting an Array with MPI_Scatterv( )	60
Code Appendix : ch4.3.cpp - Generalized Partitioning and Scattering.....	61
Code Appendix : New Function - scv_partition().....	62
<b>Chapter 5.....</b>	<b>63</b>
<i>MPI_Gatherv() Complements MPI_Scatterv().....</i>	<i>63</i>
A. Reconstituting a Partitioned Array with MPI_Gatherv().....	63
B. The Nine Arguments of MPI_Gatherv().....	69
C. Normalizing the length of a vector and some more linear algebra.....	71
Code Appendix : ch5.1.cpp - Generalized Partitioning and Scattering Template.....	78
Code Appendix : ch5.2.cpp - Reconstructing Partitioned and Scattered Data with MPI_Gatherv().....	79
Code Appendix : ch5.3.cpp - Generalization of Concatenation Technique with MPI_Gatherv().....	81
Code Appendix : ch5.4.cpp - Parallel Vector Normalization with MPI.....	83
<b>Things We Almost Forgot to Mention.....</b>	<b>85</b>
The Underestimated Importance of the Collective Commands.....	85
More MPI and MPICH Commands.....	86
Bug.....	86
Reminders.....	86
Hints.....	86
Why We Expect to be Using MPI More in the Future.....	87
<b>Appendix I : Troubleshooting.....</b>	<b>89</b>
Compiler Problems.....	89
Apple/Mac OS X Issues.....	90
mpiexec Warning.....	90
What to do if an MPI Program Crashes.....	90
Other Errors.....	90
<b>Appendix II – The Complete Header File.....</b>	<b>91</b>

<i>supercomputing_simplified.hpp</i> .....	91
<b>Appendix III – The Provided Functions</b> .....	<b>93</b>
<i>ScS_functions.cpp</i> .....	93
<b>Appendix IV : Some MPI Data Types</b> .....	<b>98</b>
<b>Appendix V : Handy Linux Commands</b> .....	<b>99</b>
Quick Reference.....	99
A. Introduction & Formatting Notes.....	100
B. The File System.....	100
C. Essential Commands (Elementary Navigation).....	101
D. Options.....	101
E. The Manual.....	102
F. Directory Navigation & File Management.....	103
G. Reading Files & Creating Output.....	104
H. Managing Executing Programs.....	104
I. Complex Commands.....	105
J. File Permissions.....	106
K. Secure Communication with Other Linux Computers.....	108
<b>Appendix VI: References &amp; Suggested Further Reading</b> .....	<b>110</b>

# Preface

## Who Should Read This Booklet?

This booklet is written for novice C programmers who want to use, understand and perhaps even write programs that run on supercomputers, clustered PC's, or any system of linked multiprocessors that can be harnessed by the Message Passing Interface, MPI. We assume that the motivation for using such hardware arises from a need for greater speed of execution and, behind this motivation, the strategic thought that time-savings can be produced by some form of parallel execution.

The C/C++ programs themselves are purposely quite simple to avoid distracting the student from the central focus, the MPI part of the lessons. In this same vein, the neophyte C-programmer is bound to have trouble with pointers, so we incorporated them as explicit declarations as gradually as we could. We also filled a gap or two, in terms of specific, precisely relevant examples, that seem to be left as exercises in elementary C/C++ texts but that are critical for using the MPI functions.

The simple programming lessons and terse exposition found here should appeal to a particular audience, those who use computers as computational tools: scientists like ourselves; engineers who write their own simulations; scientists and engineers in training; and last but far from least, teachers of pre-college computer science students and pre-college students themselves, particularly those with the discipline and interest to teach themselves. On the other hand, university-level computer scientists and professional programmers will find the presentation here far too elementary. These two groups will likely prefer a textbook such as Quinn's (2004), which we really like, and the primary source for MPI, MPI-The Complete Reference vols I, II, and III. We continually returned to these sources as our guides. However, it would be hard to underestimate how much information we ignored, information that a computer scientist and professional programmer would want, perhaps demand, to know. Not only does this booklet ignore almost two hundred MPI functions, but it also ignores lengthy discussions about using these commands, and it skips over the sketches of the underlying philosophy, issues, and the compromises with which the creators of MPI grappled. If this is your meat and potatoes, then take the time to enjoy the books just cited.

The presentation is expressly aimed at the novice programmer. To take advantage of the lessons requires the smallest experience with C or C++. No special types or classes are created. The most complicated aspects of the computer code are calls to functions and passing variable values

with pointers.

Compared to the available texts, the sequenced development of the source code here provides the shortest and simplest path from a serial programming ability and mindset to the creation of parallel C/MPI2-based programs. In particular, this path is specifically laid out to install an awareness of the fundamental issues of parallel programming with multiple computers.

To accomplish an appropriately streamlined presentation, the bywords of this booklet are 'minimal but sufficient' and 'less is more'. These bywords directed our selection of the MPI functions and the design of the programming exercises. For us, minimal but sufficient expertise is the quickest path for our students to become comfortable using the custom parallel code of the laboratory, to decide if they want to go farther and learn more, perhaps even create their own parallel programs. Quite importantly, if they insist on programming, we want them thinking in terms of code simplicity relative to the task of parallel execution of their algorithms.

Although it will depend on your programming background, we foresee just a couple of hours to read through, to modify, to compile, and to execute the programs in Chapter 1 through Chapter 5. At that point you will be equipped with the essentials needed to program a Beowulf system or other forms of multiprocessor computers that can be run within the MPI format.

### **Why the Message Passing Interface, MPI?**

For reasons of program simplicity, portability across systems, and costs associated with testing and maintenance, MPI is today's clear choice for parallel programming. It is particularly appealing for two reasons: its development is supported by the U.S. government, and perhaps most importantly, it is an agreed upon, common standard for companies that produce high performance computers.

Even a very small subset of the MPI functions are enough to harness the power of supercomputers and clustered desktop computers such as Beowulf systems. The current version of MPI is MPI2. This system of functions makes parallelization of C/C++ or Fortran serial programs a relatively simple exercise, whether the parallelized form is embarrassingly parallel or a more sophisticated forms of parallelization.

# How to Use This Booklet

This booklet is designed as a linear and seamless series of programming exercises with interspersed explication. The continuity of the presentation derives from the gradual growth of the source code, little-by-little and chapter-by-chapter.

To gain the deepest understanding of what is being taught, do the programming exercises; edit the source code; and of course, run and test the compiled code whenever suggested. To make this task as easy and simple as possible, the programming exercises are supported by source code templates; these templates minimize busy work and focus attention on the few MPI commands presented here.

With each chapter, there is an initial template file. A second, and sometimes a third, source code file is provided at the end of each chapter. These additional programs are there so you can compare your source code against what we consider the minimum programs for the exercises of that chapter.

When a new function has MPI relevance, it is developed and described within the main text; otherwise, each novel function can be found at the end of the chapter, usually with comments. Typically, a new non-MPI function starts out as a few successive lines of code in `main ()`; in the following chapter it is removed from `main ()` and promoted to `ScS_functions.cpp`. The mature version of this file is available as an appendix at the end of the booklet. Likewise, `supercomputing_simplified.hpp` contains the necessary preprocessor statements such as

- (1) the `#include`'s,
- (2) the declaration using `namespace std`, which takes advantage of the collection of commands and variables in the standard library, and
- (3) the function prototypes.

These two files can be found in appendices at the end of the book.

Just the basics of programming C/C++ are necessary. (There is currently no official MPI for Java-based code.) If more exercises are needed, then almost any linear algebra task can be parallelized and can form the basis of a class exercise or homework assignment.

## Computer Science Terminology

A small but often chaffing issue is the jargon of computer science in juxtaposition with the jargon of people who do simulations and who are substantively outside of computer science. Almost all of the time, the presentation here uses the jargon of computer science. Several words are subject to misinterpretation by those of us outside of computer science. Here, in italics, are some words of computer science being used by, presumably, computer scientists. These usage examples are culled from main entries of Wikipedia including the parenthetical comments found there; these usage examples are followed by some comments of our own in the standard font:

i) *variable* - ... *a variable can be thought of as a place to store a value in*

# Software and Hardware Requirements

The command line statements and the programs here are designed to execute under MPICH and gcc 3.3.

The programs are supported if run on a Baby-B cluster purchased from Informed Simplifications or if the user receives a liveCD that has successfully booted and that falls within the contractual parameters accepted by using this CD.

The programs have been successfully built and executed on single processor desktop computers using standard Intel and AMD processors with Debian Sarge and Fedora Core 4 Linux and also on an Apple iBook with a g4 processor.

This booklet is written under the presumptions that

- (1) a single processor is all that is available,
- (2) MPICH has been properly installed, and
- (3) the appropriate MPICH settings have been made.

In the chapters that follow, we further assume that your MPICH installation and programs are available from the command line without entering a path specification.

The programs may or may NOT work on your cluster (unless it is a Baby-B) since we do not know its hardware/software configuration. If you have your own cluster and find out the programs work, then we would like to hear from you! If have your own cluster and find out the programs do not work, then we would like to hear from you!

If you are using our LiveCD, the minimum recommended equipment is a desktop system with an x86 processor of at least 330 MHz, 64MB of RAM, an 8x CDROM drive, a VGA compatible video card, a standard monitor, a keyboard, a three button PS/2 mouse and properly set boot-from-CD capability. Specialized hardware is not explicitly supported, including, but not limited to, non-PS/2 mice, specialized video card, SATA, RAID or other types of hard drives and any laptop computer.

# Chapter 1

## Compiling and Executing an MPI-based Program

This chapter has two purposes. First, we need to find out if the appropriate software is available in your working directory. Second, you need explicit knowledge of compiling and executing an MPI-based program from the command line. (Baby-B™ owners can use the point and click icons and skip this section if they do not wish to work at the command-line.)

The recommended interface between a Linux operating system and an MPI2-based program is MPICH (Gropp et al, 1998). Specifically, MPICH provides subroutines for compiling and executing MPI-based programs, both MPI1 and MPI2. A so-called Message Passing Daemon (labeled MPD or mpd) mediates the communication for each computational node (in a Beowulf system and similar clusters, a node is a computer with its own IP address). Fortunately, for the purpose of education, software debugging, and portability, MPD can run multiple Processes on a single CPU. Thus, even if you do not have a supercomputer or a Beowulf system, the training lessons can proceed. Indeed, we assume you are working with a single CPU.

At this point we will find out if your MPICH is properly installed and if it can be reached from your working directory. If these two tests are successful, you will also find out how to execute a C/MPI program. So let's arouse a message passing daemon.

### A. Activating the Message Passing Daemon

We assume that you are executing your programs within the `/home/<user>/supercomputing-simplified/` directory and that this directory contains the files which accompany this booklet. In the typing instructions that follow, each line represents an individual command. When you finish typing in an individual command you should press the “Enter” key once. The `$` represents the command line, so you know when we are providing you with commands to type into the shell. You should not enter the `$` at the prompt.

With the first command, you will activate one Message Passing Daemon for the one processor available. With the second command, you will test whether the daemon has started and is working correctly.

```
$ mpdboot -n 1
```

**Code Appendix : ch1.1.cpp - The MPI Template**

```
#include <mpi.h>
#include "supercomputing_simplified.hpp"

int main (int argc, char *argv[])
{
    int P_rank;                //Process identification number
    int worldsize;            //number of processes running

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &P_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &worldsize);

    // ***** //
    // Your MPI program goes here. //
    // ***** //

    MPI_Finalize();
    return 0;
}
```

# Chapter 2

## Using MPI\_Barrier() To Enforce Parallel Execution

### A. Introduction

There are several ways that scientist and engineers can take advantage of computer clusters. The easiest and most efficient way goes by the name of 'embarrassingly parallel' or 'naturally parallel'. The technique is useful when (1) your program works well on a single desktop computer and (2) you need to study the effect of parameter variation on the resulting output. In such cases, you should treat the cluster of computers as if each computer in the cluster is a stand alone desktop computer. If the cluster is all yours and is made up of twenty-five computers, then you run your program twenty-five times, each time with a different set of parameters or initial conditions. We will not comment further on this method except to say that it is usually the most efficient way to use a cluster and that `mpirexec` can be useful for launching such programs.

Here we are interested in computational problems whose execution is unacceptably slow when run on a single desktop machine and on problems that have an inherent parallel-like execution requirement. For example, consider our neuron-based model of hippocampal function. When simulated with twelve thousand neurons and fifteen million synapses, today's desktop computer is adequate for our neuron-based models hippocampal function although faster execution can be obtained when using a cluster of computers for a single simulation. However, for simulations requiring one hundred thousand neurons and one billion synapses, a single desktop machine is inadequate and a supercomputer or a computer cluster running under MPI is a necessity for this research. In addition, a neural network simulation is a good example of a problem with inherent parallel characteristics. That is, each neuron performs its own local transformations independent of all other neurons' states at that instant (although very much dependent on some of the past states).

In such problems where the system being simulated has an inherently parallel characteristic, it is sensible, potentially efficient, and perhaps required, to run one simulation by harnessing a cluster of computers. In such cases, for parallel computation to occur, there is a requirement for approximate synchronization of program execution across CPU's and, at the software level,

# Chapter 3

## Transmitting Data Between Processes with MPI\_Bcast()

This chapter presents the first of the three data transmitting functions taught by this booklet.

`MPI_Bcast()` is the first because it takes the fewest arguments of all the data-transmitting, collective commands and because, having the fewest arguments, it is the easiest to use.

By combining this command with `MPI_Barrier()`, you will gain full control of the output format, and with regard to input, you will be able to communicate with all the Processes from the keyboard or whatever method you eventually choose to communicate with an MPI-based cluster. In fact, let's take care of the input problem that was discussed at the end of the last chapter right now. This problem is fixed by Process 0 sending the keyed entry to all Processes.

### A. Sending the Keyboard Entry to All Processes

Load this chapter's template source code, `ch3.1.cpp`, into your editor and save it as `mych3.2.cpp`. The source code in this file is nearly identical to your last program but we have promoted the keyboard receiving routine into a function, added a little more error trapping, and renamed a variable; the keyboard entry is now stored as the variable named `rootfordata`, which entails replacing the variable `entry` five times.

Add the following instruction to your source code. This instruction sends the keyboard entry stored at Process 0 to all Processes,

```
MPI_Bcast(&kbentry, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Add this instruction just after the `MPI_Barrier()` instruction that follows the keyboard entry function. Here is the source code fragment after inserting the new instruction.

# Chapter 4

## Transmitting Data Between Processes with MPI\_Scatterv()

Like `MPI_Bcast()`, the scatter commands send data from a specified root Process to all the Processes of the communicator, but the scatter commands – particularly `MPI_Scatterv()` – do so in a manner significantly more sophisticated than `MPI_Bcast()`. The scatter commands can send data of one named array to an array of a different name. Thus, the overwriting that characterizes `MPI_Bcast()` need not occur when using `MPI_Scatterv()`, or any other scatter command. Even better, when using `MPI_Scatterv()`, the sending lengths can differ across Processes. Such Process-specific lengths, combined with Process-specific pointers for locations within the original data array on root, make it simple to partition the original array into several smaller arrays, where each of these smaller arrays is sent to one Process in the communications group. Because `MPI_Scatterv()` is so customizable and therefore versatile, it is the only scatter command you need. Thus, it is the only one we teach here. However, its great versatility comes at a price – the arguments are complicated, so there is a lot to learn and remember.

### A. Comparing MPI\_Scatterv() to MPI\_Bcast()

To speed up learning and to emphasize the greater versatility of `MPI_Scatterv()`, we have a teaching trick. You will replace the second `MPI_Bcast()` function of the last program with an `MPI_Scatterv()` function. That is, we are replacing a five argument function with a nine argument function, a poor programming tactic but a useful one for teaching.

Let's make the switch. Load `my_ch3.5.cpp` into your editor, and locate the second `MPI_Bcast()`. The `MPI_Bcast()` being replaced is the one that sends a thirteen element data array.

```
MPI_Bcast(data_array, d_size, MPI_INT, rootfordata, MPI_COMM_WORLD);
```

Replace this instruction with

```
MPI_Scatterv(data_array, blocklengths, blockoffsets, MPI_INT,  
            data_array, rcv_count, MPI_INT,  
            rootfordata, MPI_COMM_WORLD);
```

But do not compile just yet because there are some undeclared and unvalued variables

# Chapter 5

## MPI\_Gatherv() Complements MPI\_Scatterv()

In this chapter, your ultimate task is to write a parallel program for a fundamental computation of linear algebra, the Euclidean normalization of a vector's length. This particular calculation is prototypical of many algorithms in the sense that the full computation occurs in stages where such stages alternate between parallel and serial computations. That is, successive stages alternate between identical arithmetical operations accomplished simultaneously on many processors and arithmetical operations requiring collection of the parallel computations into a single Process where the serial computations are then performed. Extending this alternating form of processing, parallel-serial-parallel-serial-etc, is quite typical. An obvious example is a recursive algorithm in which each pass through the algorithm uses one parallel set and then one serial set of operations.

In order to achieve a parallel execution after a serial execution, scattering data transmissions are necessary, and to achieve serial execution after a parallel execution, gathering transmissions are very useful. As part of such a cycle, you will use the function `MPI_Gatherv()` to bring data scattered over several Processes back to a single root Process.

Like the variable block length scattering command of the last chapter, `MPI_Gatherv()` has nine arguments, and most of these correspond to the arguments of `MPI_Scatterv()`. Moreover, the arguments that are not exactly the same arguments as the ones in `MPI_Scatterv()` are mirror images of them. The two, mirror-like perspectives arise from the movement of data relative to the root Process. In the scattering case, the function sends data concentrated at root to all Processes, while in the gathering case, the root Process concentrates data that is scattered across all Processes.

### A. Reconstituting a Partitioned Array with MPI\_Gatherv()

Let's reverse the partitioning performed by the first program of the last chapter. This is the program named `mych4.2.cpp`, which is the program requiring exactly three Processes. Here you will use the `MPI_Gatherv()` instruction to reverse the partitioning created by `MPI_Scatterv()` in `mych4.2.cpp`. This particular reversing operation is an example of

## Code Appendix : ch5.4.cpp - Parallel Vector Normalization with MPI

```

#include <mpi.h>
#include "supercomputing_simplified.hpp"
#include "ScS_functions.cpp"

int main (int argc, char *argv[])
{
    int P_rank, worldsize, rootfordata=-99, i, myturn;
    const int d_size=13;
    int data_array[d_size];
    float bigtot=0.0, sqsubtot=0.0, checklength=0.0;
    float eulength[]={0.0}, data_arrayf[d_size], tempfileforworkf[d_size];
    for (i = 0; i < d_size; i++){ tempfileforworkf[i] = 0.0;}

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &P_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &worldsize);

    for (i = 0; i < d_size; i++)
        {data_array[i] = i+10*P_rank;data_arrayf[i]=(float)data_array[i];}

    ourmpi_printf(data_array, d_size, P_rank,worldsize,"Process %i's data: ");

    rootfordata = keyb_entry_nonneg_int(P_rank,worldsize,"\nChoose data from a
single Process");
    MPI_Bcast(&rootfordata,1,MPI_INT,0,MPI_COMM_WORLD);

    //.....partition and scatter
    int* blocklengths= new int[worldsize];
    int* blockoffsets= new int[worldsize];
    scv_partition(worldsize, d_size,blocklengths,blockoffsets);
    int rcv_count=blocklengths[P_rank];
    MPI_Scatterv(data_arrayf, blocklengths, blockoffsets, MPI_FLOAT,
                tempfileforworkf,rcv_count,MPI_FLOAT,
                rootfordata,MPI_COMM_WORLD);

    //.....local sum of squares
    sqsubtot=innerprod(tempfileforworkf,tempfileforworkf,blocklengths[P_rank]);
    // option to print local sum of squares
    ourmpi_printf(&sqsubtot,1,P_rank,worldsize,
                "Sum of squares at Process %d: ");
    MPI_Barrier(MPI_COMM_WORLD);
    //.....gather subtotals and extract square root of total.....
    int* rcv_singletons= new int[worldsize];
    for (i=0;i<worldsize;i++) {rcv_singletons[i]=1; }
    int* rcv_offsets = new int[worldsize];
    for (i=0;i<worldsize;i++) {rcv_offsets[i]=i;}
    float* subtot = new float[worldsize];
    for (i = 0; i < worldsize; i++){ subtot[i] = 0.0; }
    MPI_Gatherv(&sqsubtot, 1, MPI_FLOAT,
                subtot, rcv_singletons, rcv_offsets, MPI_FLOAT,
                0, MPI_COMM_WORLD);
    delete [] rcv_singletons;

```

```

delete [] rcv_offsets;

if (P_rank==0)
{
    for (i=0;i<worldsize;i++){bigtot = bigtot + subtot[i];}
    eulength[0]=sqrt(bigtot);
    printf("\nTotal sum of squares %d\n", (int) bigtot);
    printf("Euclidean length of Process %d's data_array %f \n\n", //
           rootfordata, eulength[0]);
}
delete []subtot;
//....broadcast euclidean length and normalize scattered elements....
MPI_Bcast(eulength, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
for (i=0;i<blocklengths[P_rank];i++ )
{tempfileforworkf[i] = tempfileforworkf[i]/eulength[0];}
//the printing below is from zero
//.....gather normalized elements and concatenate.....
MPI_Gatherv(tempfileforworkf, rcv_count, MPI_FLOAT,
            data_arrayf, blocklengths, blockoffsets,
            MPI_FLOAT, 0, MPI_COMM_WORLD);
delete []blocklengths;
delete []blockoffsets;
checklength=innerprod(data_arrayf, data_arrayf, d_size);
printf_oneP_nobarrier(0, P_rank, data_arrayf, d_size, "Process %d's
normalized array:\n", rootfordata);
if (P_rank==0)
{
    printf("\n normalized array length check (is it 1?); uses innerprod(): \
n %f\n", checklength);
}
MPI_Finalize();
return 0;
}

```

# Appendix V : Handy Linux Commands

The Linux/Unix operating systems comprise an enormous number of commands. However, very few commands are needed by the typical user.

## Quick Reference

Our three favorite commands are : “ls”, “cd”, and “find”.

Command	Function
\$ ls	List the files in the current directory
\$ ls -lth	List the files in the current directory in order of modification, newest to oldest, with permissions, owner, group, size (human readable), modification date, and modification time.
\$ find / -name " <i>filename</i> " 2> /dev/null	Find the file named <i>filename</i> (case sensitive) by searching through everything contained in / (the directory that contains all other directories). !WARNING! This can take a very long time! The 2> /dev/null part throws away any errors so that the output isn't cluttered with error messages. This is necessary because normal users cannot search a number of directories owned by the superuser (root), so find reports errors.
\$ cd <i>directoryname</i>	Change to the <i>directoryname</i> in the current directory (move “down”)
\$ cd ..	Change to the directory that contains the current directory (move “up”)

Directional words for the file system can be confusing because file systems are described as “trees” because of their hierarchical branching patterns. That is, the words describing the direction of movement away from or toward the beginning of the tree picture an upside down tree: the root directory from which everything else grows is at the top and the branchings ending are at the bottom of the tree. Thus, file systems are more like the root system of a tree, in that the origin of the file system is as far “up” as one can go. File systems grow “down”, away from the origin.