22 Parallel Computing with MPI

In this chapter, we present a tutorial on the use of MPI on a small Beowulf cluster composed of Unix or Linux computers. This follows our philosophy of "learning while doing." Our presentation is meant to help a user from the ground up, something that might not be needed if you were working at a central computing center with a reasonable level of support. Although your **problem** is still to take the program you wrote to generate the bifurcation plot for bug populations and run different ranges of μ values simultaneously on several CPUs, in a more immediate sense, your task is to get the experience of running MPI, to understand some of the MPI commands within the programs, and then to run a timing experiment. In Section 22.7, at the end of the chapter we give a listing and a brief description of the MPI commands and data types. General information about MPI is at [52], detailed information about the syntax of MPI commands is at [53], while other useful materials are at [54]. The standard reference on the C language is [63], although we prefer [64]. [1]MPI is very much the standard software protocol for parallel computing, and is at a higher level than its predecessor PVM.

Nevertheless, we have found that the software for parallel computing is not as advanced as the hardware. There are just too many details for a computational scientist to worry about, and much of the coding is at a much lower, or more elementary level than that with which we usually deal. This we view as a major step backward from the high-level programming that was possible on the shared memory supercomputers of the past. This, apparently, is the price to be paid for the use of inexpensive clusters. Indeed, John Dongarra, a leading HPC scientist has called the state of parallel software a crisis, with programming still stuck in the 1960s.

While in the past we have run Java programs with a version of MPI, the different communication protocols used by MPI and Java have led to poor performance, or additional complications needed to improve performance [65].

[1] These materials were developed with the help of Kristopher Wieland, Kevin Kyle, Dona Hertel, and Phil Carter. Some of the other materials derive from class notes from the Ohio Super Computer Center, which were supplied to us, and written in part, by Steve Gordon.

Computationyal Physics. Problem Solving with Computers (2nd edn).
Rubin H. Landau, Manuel José Páez, Cristian C. Bordeianu
Copyright © 2007 WILEY-VCH Verlag GmbH & Co. KGaA, Weinheim
ISBN: 978-3-527-40626-5

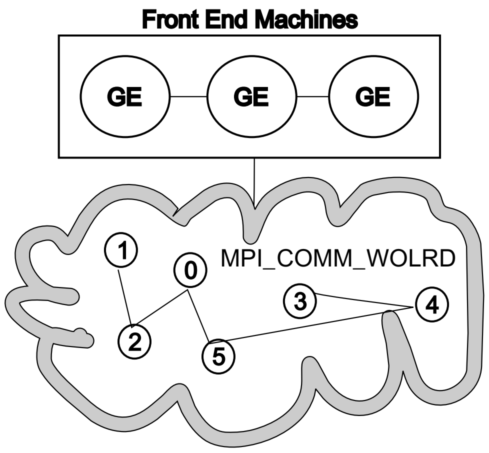

Fig. 22.1 A schematic view of a cluster (cloud) connected to front end machines (box).

In addition, you usually would not bother parallelizing a program unless if it requires very large amounts of computing time or memory, and those types of programs are usually written in Fortran or C (both for historical reasons and because Java is slower). So it makes sense for us to use Fortran or C for our MPI examples. We will use C because it is similar to Java.

22.1 Running on a Beowulf

A Beowulf cluster is a collection of independent computers each with its own memory and programs that are connected to each other by a fast communication network over which the messages are sent. MPI stands for *Message Passing Interface*. It is a library of commands that make communication between programs running on different computers possible. The messages sent are usually data contained in arrays. Because different processors cannot directly access the memory on some other computer, when a variable is changed on one computer, it does *not* get changed in the other copies of the program running on other processors. This is an example of where MPI comes into play.

In Fig. 22.1 we show a typical, but not universal, configuration for a Beowulf cluster. Almost all clusters have the common feature of using MPI for communication among computers and Unix/Linux for the operating system. The cluster in Fig. 22.1 is shown within a cloud. The cloud symbolizes the grouping and connection of what are still independent computers communicating via MPI (the lines). The `MPI_COMM_WORLD` within the cloud is an MPI data type containing all the processors that are allowed to communicate with each other (in this case six).

The box in Fig. 22.1 represents the *front end* or *submit hosts*. These are the computers from which users submit their jobs to the Beowulf and later work with the output from the Beowulf. We have placed the front-end computers outside of the Beowulf cloud, although it could be within. This type of configuration frees the Beowulf cluster from administrative chores so that they may concentrate on number crunching, and is useful when there are multiple users on the Beowulf. However, some installations have the front-end machines as part of the Beowulf, which lets them be used for number crunching as well. Other installations, particularly those with only a single user, may have no front end and permit the users to submit their jobs on any computer.

Finally, note that we have placed the letters *GE* within the front-end machines. This represents a configuration in which these computers are also running some type of a *grid engine* or queuing system that oversees the running of jobs submitted to MPI by a number of users. For instance, if we have a cluster of 20 computers and user A requests 10 machines and user B requests 8 machines, then the grid engine would permit both users to run simultaneously and assign their jobs to different computers. However, if user A had requested 16 machines, then the grid engine would make one of the users wait while the other user gets their work done.

Some setup is required before you can run MPI on several computers at once. If someone has already done this for you, then you may skip the rest of this section and move on Section 22.2.2. Our instructions have been run on a cluster of Sun computers running Solaris Unix. You will have to change the computer names and such for your purposes, but the steps should remain the same.

- First you need to have an active account on each of the computers in the Beowulf cluster. Even if you usually run on a set of networked computers, it is likely that the Beowulf is set up to be separate.

- Open a shell on one of the Beowulf machines designated for users to sign on to (a *front end* machine). You probably do not have to be sitting at the front end to sign on, but instead can use `ssh` or *telnet*. Make the directory `mpi` in your home directory:

```
> cd ~                                   Change to home directory
> mkdir output                 Screen output gets stored here first
> mkdir output/error                 Place to store error messages
> mkdir mpi                        A place to store your mpi stuff
```

- You need to have your Beowulf account configured so that Unix can find the MPI commands that you issue from the command line or from your programs. When you log onto the computer, the operating system reads a configuration file `.cshrc` file residing in your home directory. It contains the places where the operating system looks for commands. (We are assuming here that you are using either the `cshell` or `tcshell`, if not, then modify your `.login`, which should work regardless of the shell.) When a file begins with a "dot," it is usually hidden from view when you list files, but it can be seen with the command `ls -la`. The list of places where Unix looks for commands is an *environmental variable* called your `PATH`, and it needs to include the current version of the `mpich-n.m/bin` directory where the scripts for MPI reside, e.g.,

```
/usr/local/cluster/mpich-1.2.6/bin                This should be in your PATH
```

Here the `1.2.6` may need to be replaced by the current version of MPI that is installed on your computer, as well as possibly the directory name `cluster`.

• Because your `.cshrc` file controls your environment, having an error in this file can lead to a nonfunctional computer for you. And since the format is rather detailed and unforgiving, it is easy to make mistakes. So before you mess with your existing `.cshrc` file, make a backup copy of it:

```
> cp .cshrc .cshrc_bk
```

You can use this backup file as reference, or copy it back to `.cshrc` if things get to be too much of a mess. If you have really messed things up, then your system administrator may have to copy the file back for you.

• Edit your `.cshrc` file so that it contains a line in which `setenv PATH` includes `/usr/local/cluster/mpich-1.2.6/bin`. If you do not have a `.cshrc` file, just create one. Find a line containing `setenv PATH`, and add this in after one of the colons, making sure to separate the path names with colons. As an example, the `.cshrc` file for user `rubin` is

```
 # @(#)cshrc 1.11 89/11/29 SMI umask 022
setenv PATH /usr/local/bin:/opt/SUNWspro/bin:/opt/SUNWrtvc/bin:
/opt/SUNWste/bin:/usr/bin/X11:/usr/openwin/bin:/usr/dt/bin:/usr/ucb/:
/usr/ccs/bin/:/usr/bin:/bin:/usr/sbin/:/sbin:
/usr/local/cluster/mpich-1.2.6/bin: setenv PAGER less setenv
CLASSPATH /home/rubin:/home/rubin/dev/java/chapmanjava/classes/:
/home/rubin/dev/565/javacode/:
/home/rubin/dev/565/currproj/:/home/rubin:/home/rubin/mpiJava:
/usr/local/mpiJava/lib/classes:
set prompt="%~::%m> "
```

• If you are editing your `.login` file, enter as the last line in the file:

```
set path = $path /usr/local/cluster/mpich-1.2.6/bin
```

• Because dotfiles are read by the system when you first log on, you will have to log off and back on for your changes to take effect. (You can use the `source` command to avoid logging off and on.) Once you have logged back on, check the values of your `PATH` environmental variable:

```
> echo $PATH                    From Unix shell, tells you what Unix thinks
```

- Let us now take a look at what has been done to the computers to have them run as a Beowulf cluster. On Unix systems the "slash" directory / is the root or top directory. Change directory to /

```
> cd /                          Change to root directory
```

You should see files there, such as the kernel and the devices, that are part of the operating system. You may not be permitted to examine these files, which is a good thing since modifying them could cause real problems (it is the sort of thing that hackers and system administrators do).

- MPI is a *local* addition to the operating system. On our system the MPI and SGE commands and documentation [66] are in the `/usr/local/cluster` directory. Here the first / indicates the root directory and `usr` is the directory name under root. Change directory to `/usr/local/cluster`, or wherever MPI is kept on your system, and notice the directories `scripts` and `mpich-1.2.6` (or maybe just a symbolic link `mpich`). Feel free to explore these directories. The directory `scripts` contains various scripts designed to make running your MPI programs easier. (Scripts are small programs containing system commands that get executed in order when the file is run.)
- In the `mpich-1.2.6` directory, you will notice that there are examples in C, C++, and Fortran. Feel free to copy these to your home directory:

```
> cp -r examples /home/userid/mpi
```

where `userid` is your personal computer name. We encourage you to try out examples, although some may need modification to work on your local system.

- Further documentation can be found in

/usr/local/cluster/mpich-1.2.6/doc/mpichman-chp4.pdf	MPI documentation
/usr/local/cluster/sge/doc/SGE53AdminUserDoc.pdf	SGE documentation
/usr/local/cluster/sge/doc/SGE53Ref.pdf	SGE reference
man qstat	Manual page on `qstat`

- Copy the script `run_mpi.sh` from the `Codes/MPIcodes` on the CD to your personal `mpi` directory. This script contains the commands needed to run a program on a cluster.
- Copy the file `/usr/local/cluster/mpich/share/machines.solaris` to your home directory and examine it. (The `solaris` extender is there because we are using the *Solaris* version of the Unix operating system on our Beowulf, you may need to change this for your local system.) This file contains a list of all of the computers that are on the Beowulf cluster and available to MPI for use (though there is no guarantee that all machines are operative):

```
# Change this file to contain the machines that you want to use # to
run MPI jobs on. Format: 1 host per line, either hostname # or
hostname:n, where n is the number of processors. # hostname should
be the same as output from "hostname" command paul rose tomek manuel
```

22.1.1 An Alternative: BCCD = Your Cluster on a CD

One of the difficulties in learning how to parallel compute is the need for a parallel computer. Even though there may be many computers around that you may be able to use, knitting them all together into a parallel machine takes time and effort. However, if your interest is in learning about and experiencing distributed parallel computing, and not in setting up one of the fastest research machines in the world, then there is an easy way. It is called *bootable cluster CD* (BCCD) and is file on a CD. When you start your computer with the CD in place, you are given the option of having the computer ignore your regular operating system and instead run (boot) from the CD into a preconfigured distributed computing environment. The new system does not change your system, but rather is a nondestructive overlay on top of existing hardware that runs a full-fledged parallel computing environment on just about any workstation-class system, including Macs. You boot up every machine you wish to have on your cluster this way, and if needed, set up a DNS (Domain Name System) and DHCP (Dynamic Host Configuration Protocol) servers, which are also included. Did we mention that the system is free? [67]

22.2 Running MPI

If you are the only one working on a Beowulf cluster, then it may make sense to submit your jobs directly to MPI. However, if there is the possibility that a number of people may be using the cluster, or that you may be submitting a number of jobs to the cluster, then it is a good idea to use some kind of a queue management system to look after your jobs. This can avoid the inefficiency of having different jobs compete with each other for time and memory, or having the entire cluster "hang" because a job requested a processor that is not available. We use the *Sun Grid Engine* (SGE) [66]. This queue management system works on non-Sun machines and is free. When your program is submitted to a cluster via a management system, the system will install a copy of the same program on each computer assigned to run the program.

There are a number of scripts that interpret the MPI commands you give within your programs (the commands are not part of the standard Fortran or

C languages), and then call the standard compilers. These scripts are called *wrappers* because they surround the standard compilers as a way of extending them to include MPI commands

mpicc	C compiler	**mpicxx**	C++ compiler
mpif77	Fortran 77 compiler	**mpif90**	Fortran 90 compiler

Typically you compile your programs on the front end of the Beowulf, or the master machines, but not on the execution nodes. You use these commands just like you would the regular compiler commands, only now you may include MPI commands in your source program:

```
> mpicc  -o name name.c          Compile name.c with MPI wrapper script
> mpif77 -o name name.f          Compile name.f with MPI wrapper script
```

22.2.1 MPI under a Queuing System

Tab. 22.1 Some common SGE commands.

Command	Action
qsub myscript	Submit batch script or job **myscript**
qhost	Show job/host status
qalter <job_id>	Change parameters for job in queue
qdel job_id	Remove job_id
qstat	Display status of batch jobs
qstat -f	Full Listing for qstat
qstat -u <username>	User only for qstat
qmon	X Window Frontend (integrated functionality)
cpug, bradley, emma (local names)	Front end computers

Table 22.1 lists a number of commands that you may find useful if you are using the Sun Grid Engine (SGE). Other queuing systems should have similar commands. Once your program is compiled successfully, it can be executed on the cluster using these commands. The usual method of entering the program name or `a.out` at the prompt only runs it on the local computer; we want it to run on a number of machines. In order to run on the cluster, the program needs to be submitted to the queue management system. Listing 22.1 uses the `run_mpi.sh` script and the `qsub` command to submit jobs to in *batch* mode:

```
> qsub runMPI.sh name                    Submit name to run on cluster
```

This command returns a job ID number, which you should record to keep track of your program. *Note*, in order for this script to work, both `runMPI.sh` and `name` must be in the current directory. If you need to pass parameters to your program, place the program name and parameters in quotes:

```
> qsub run_mpi.sh "name -r 10"           Parameters and program name in quotes
```

Listing 22.1: The script `runMPI.sh` used to run an MPI program.

```
#
#——————————— SHORT COMMENT ———————————————————
# Template script for MPI jobs to run on mphase Grid Engine cluster.
# Modify it for your case and submit to CODINE with
# command "qsub mpi_run.sh".

# You may want to modify the parameters for
# "-N" (job queue name), "-pe" (queue type, numb of requested CPUs),
# "myjob" (your compiled executable).


# Can compile code, for example myjob.c (*.f), with GNU mpicc or
#  mpif77 compilers as follows:
# "mpicc -o myjob myjob.c" or "mpif77 -o myjob myjob.f"

# You can monitor your jobs with command
# "qstat -u your_username" or "qstat -f" to see all queues.
# To remove your job, run "qdel job_id"
# To kill running job, use "qdel -f job_id"

# ————Attention: #$ is a special CODINE symbol, not a comment ———
#
#   The name, which will identify your job in the queue system
#$ -N MPI_job
#
#   Queue request, mpich. You can specify  number of requested CPUs,
#   for example, from 2 to 3
#$ -pe class_mpi 4-6
#
# ————————————————————————
#$ -cwd
#$ -o  $HOME/output/$JOB_NAME-$JOB_ID
#$ -e  $HOME/output/error/$JOB_NAME-$JOB_ID.error
#$ -v  MPIR_HOME=/usr/local/cluster/mpich-1.2.6
# ————————————————————————

echo "Got $NSLOTS slots."


#  Don't modify the line below if you don't know what it is
$MPIR_HOME/bin/mpirun -np $NSLOTS $1
```

22.2.1.1 **Status of Submitted Programs**

After your program is successfully submitted, SGE places it into a queue where it waits for the requested number of processors to become available. It then executes your program on the cluster, and *directs the output to a file* in the `output` subdirectory within your home directory. The program itself uses the MPI and C/Fortran commands. In order to check the status of your submitted program, use `qstat` along with your Job ID number:

```
> qstat 1263                                   Tell me the status of JOB 1263
```

```
job-ID prior name user state submit/start at queue master ja-task-ID
1263 0 Test_MPI_J dhertel qw 07/20/2005 12:13:51
```

The is a typical output that you may see right after submitting your job. The `qw` in the `state` column indicates that the program is in the **q**ueue and **w**aiting to be executed.

```
> qstat 1263                                Same as above, but at later time
job-ID prior name user state submit/start at queue master ja-task-ID
1263 0 Test_MPI_J dhertel t 07/20/2005 12:14:06 eigen11.q MASTER
1263 0 Test_MPI_J dhertel t 07/20/2005 12:14:06 eigen11.q SLAVE
1263 0 Test_MPI_J dhertel t 07/20/2005 12:14:06 eigen12.q SLAVE
1263 0 Test_MPI_J dhertel t 07/20/2005 12:14:06 eigen3.q SLAVE
1263 0 Test_MPI_J dhertel t 07/20/2005 12:14:06 eigen5.q SLAVE
1263 0 Test_MPI_J dhertel t 07/20/2005 12:14:06 eigen8.q SLAVE
```

Here the program has been assigned a set of nodes (`eigenN` is the name of the computers), with the last column indicating whether that node is a master, host, slave, or guest (to be discussed further in Section 22.2.3). At this point the `state` column will have either a `t` indicating `transfer`, or an `r` indicating `running`.

The **output** from your run is sent to the file `Test_MPI.<jobID>.out` in the `output` subdirectory within your home directory. Error messages are sent to a corresponding file in the `error` subdirectory. Of course you can still output to a file in the current working directory, as well as **input** from a file.

22.2.2
Your First MPI Program: MPIhello.c

Listing 22.2 gives the simple MPI program `hello2.c`. It has each of the processors print `Hello World from processor #`, followed by the rank of the processor (we will talk more about rank). Compile `hello.c` using :

```
> mpicc hello.c -o hello                    Compilation via compiler wrapper
```

After successful compilation, an executable file `hello` should be placed in the directory in which you did the compilation. The program is executed via the script `runMPI.sh` (or `run_mpi.sh`), either directly, or by use of the management command `qsub`:

```
> runMPI.sh hello                           Run directly under MPI
> qsub runMPI.sh hello                      Run under management system
```

This script sets up the running of the program on the 10 processors, with processor 0 the host and processors 1–9 the guests.

Listing 22.2: The C program `MPIhellow.c` containing MPI calls that gets each processor to say hellow.

```
// MPIhello.c          has each processor prints hello to screen

#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] ) {
  int myrank;
  MPI_Init( &argc, &argv );                          // Initialize MPI
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);            // Get CPU's rank
  printf( "Hello World from processor %d\n", myrank );
  MPI_Finalize( );                                    // Finalize MPI
  return 0;
}
```

22.2.3
MPIhello.c Explained

Here is what is contained in `MPIhello.c`:

- The inclusion of the MPI headers via the `#include "mpi.h"` statement on lines 2–3. These are short files that assist the C compiler by telling it the type of arguments that MPI functions use for input and output, without giving any details about the functions. (In Fortran we used `include "/usr/local/cluster/mpich-2.1.6/include/mpif.h"` after the `program` line.)
- The main method is declared with `int main(int argc, char *argv[])` statement, where `argc` is a pointer to the number of arguments, and `argv` is a pointer to the argument vector passed to main when you run the program from the a shell. (*Pointers* are variable types that give the locations in memory where the values of the variables reside, rather than the variables' actual values.) These arguments are passed to MPI to tell it how many processors you desire.
- The `int myrank` statement declares the variable `myrank`, which stands for the *rank* of the computer. Each processor running your program under is assigned a unique number called its *rank* by MPI. This is how you tell the difference among the identical programs running on different CPUs.
- The processor that the program is executed on is called the *host* or *master*, and all other machines are called *guests* or *slaves*. The host always has `myrank` = 0, while all the other processors, based on who responds first, have their processor numbers assigned to `myrank`. This means that `myrank` = 1 for the first guest to respond, 2 for the second, an so on. Giving each processor a unique value for `myrank` is critical to how parallel processing works.

- Lines 5 and 8 of `hello2.c` contain the `MPI_init()` and `MPI_Finalize()` commands that initialize and then terminate MPI. All MPI programs must have these lines, with the MPI commands always placed between them. The `MPI_Init(&argv, &argc)` function call takes two arguments, both beginning with a `&` indicating pointers. These arguments are used for communication between the operating system and MPI.
- The `MPI_Comm_rank(MPI_COMM_WORLD, &myrank)` call returns a different value for `rank` for each processor running the program. The first argument is a predefined constant which tells MPI which grouping of processors to communicate with. Unless one has set up groups of processors, just use the default a *communicator* `MPI_COMM_WORLD`. The second argument is an integer which gets returned with the rank of the individual program.

When `hello2.c` is executed, each processor prints its rank to the screen. You should notice that it does not print the ranks in order, and that the order will probably be different each time you run the program. Take a look at the output (in the file output/`MPI_job-xxxx)`. It should look something like this, depending on how many nodes were assigned to your job:

```
"Hello, world!" from node 3 of 4 on eigen3.science.oregonstate.local
"Hello, world!" from node 2 of 4 on eigen2.science.oregonstate.local
Node 2 reporting "Hello, world!" from node 1 of 4 on
eigen1.science.oregonstate.local "Hello, world!" from node 0 of 4 on
eigen11.science.oregonstate.local
```

It might seem that the line `Node 2 reporting` should come last in the output, but it may not. In fact, if you run the program a few times, the lines in the output may appear in different orders each time because each node runs its own code without waiting for any other nodes to catch up. In the output above, node 2 finished outputting two lines before nodes 0 and 1.

If processing order matters to obtain proper execution, call `MPI_Barrier(MPI_COMM_WORLD)` to synchronize the processors. It is similar to inserting a starting line at a race; a processor will stop and wait at this line until all other processors reach it, and then they all set off at the same time. However, modern programming practice suggests that you try to design your programs so that the processors do not have to synchronize often. Having a processor stop and wait obviously slows down the number crunching, and consequently removes some of the advantage of parallel computing. However, as a scientist it is more important to have correct results than fast ones, and so do not hesitate inserting barriers if needed.

Exercise: Modify `hello2.c` so that only the guest processors say hello. *Hint:* What do the guest processors all have in common?

22.2.4 Send/Receive Messages: MPImessage2.c

Listing 22.3: The C program `MPImessage2.c` uses MPI commands to both send and receive messages. Note the possibility of blocking, in which the program waits for a message.

```
// MPImessage2.c: source node sends message to dest

#include "mpi.h"
#include <stdio.h>

int main(int argc,char *argv[]) {

  int rank;
  int msg_size = 6;
  MPI_Status status;
  int tag = 10;
  int source = 0, dest = 1;
  MPI_Init(&argc,&argv);                                  // Initialize MPI
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);                    // Get CPU's rank
  if (rank == source) {
    char *msg = "Hello";
    printf("Host about to send message: %s\n",msg);
              // Send message, may block till dest recieves message
    MPI_Send(msg, msg_size, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }
    else if (rank == dest) {
      char buffer[msg_size+1];
            // Receive message, may block till dest receieves message
      MPI_Recv( &buffer, msg_size, MPI_CHAR, source, tag,
                MPI_COMM_WORLD, &status );
      printf("Message recieved by %d: %s\n", rank, buffer);
    }
      printf("NODE %d done.\n", rank);                    // All nodes print
      MPI_Finalize();                                      // Finalize MPI
      return 0;
}
```

Sending and receiving data constitute the heart of parallel computing. The guest processors need to transmit the data they have processed back to the host, and the host has to assemble the data and then assign new work to the guests. An important aspect of MPI communication is that if one processor sends data, another processor must receive those same data. Otherwise the sending processor waits indefinitely for a signal that its data have been received! Likewise, after a processor executes a `receive` command, it waits until those data arrive.

There is a basic MPI command `MPI_Send` to send a message from a *source* node, and another basic command `MPI_Recv` needed for a *destination* node to receive it. The message itself must be an array, even if there is only one element in the array. We see these commands in use in `message2.c` in Listing 22.3. This program accomplishes the same thing as `hello.c`, but with send and receive

Tab. 22.2 The arguments for MPI_Send and MPI_Recv.

Argument name	Description
msg	Pointer (& in front) to array to send/receive
msg_size	Size of array sent; may be bigger than actual size
MPI_TYPE	Predefined constant indicating variable type within array other possible constants: MPI_INTEGER, MPI_DOUBLE
dest	Rank of processor receiving message
tag	A number that uniquely identifies message
comm	A *communicator*, e.g., predefined constant MPI_COMM_WORLD
source	Rank of processor sending message; if receiving messages from any source, use predefined constant MPI_ANY_SOURCE
status	Pointer to variable type MPI_Status containing status info

commands. The host sends the message and prints out a message, while the guests also print when they receive a message. The forms of the send/receive commands are:

```
MPI_Send(msg, msg_size, MPI_TYPE, dest, tag, MPI_COMM_WORLD);        Send
MPI_Recv(msg, msg_size, MPI_TYPE, source, tag, comm, status);     Receive
```

The arguments and their descriptions are given in Tab. 22.2.4. The criteria for successfully sending and receiving a message are as follows.

1. The sender must specify a valid destination rank, and the processor of that rank must call MPI_recv.
2. The receiver must specify a valid source rank or MPI_ANY_SOURCE.
3. The send and receive *communicators* must be the same.
4. The tags must match.
5. The receiver's message array must be large enough to hold the array.

Exercise: Modify message2.c so that all processors say hello. □

22.2.5 Receive More Messages: MPImessage3.c

Listing 22.4: The C program MPImessage3.c containing MPI's commands that have each guest processor send a message to the host processor, who then prints that guest's rank.

```
//  MPImessage3.c: guests send rank to the host, who prints them

#include "mpi.h"
#include <stdio.h>

int main(int argc,char *argv[])  {
  int rank, size;
  int msg_size = 6;
  MPI_Status status;
  int tag = 10;
  int host = 0;
  MPI_Init(&argc,&argv);                                    // Initialize MPI
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);                      // Get CPU's rank
  MPI_Comm_size(MPI_COMM_WORLD,&size);                // Get number of CPUs
  if (rank != host) {
    int n[1];                                         // Array of 1 integer
    n[0] = rank;
    printf("node %d about to send message\n", rank);
    MPI_Send(&n, 1, MPI_INTEGER, host, tag, MPI_COMM_WORLD);
  }
  else  {
    int r[1];
    int i;
    for(i=1; i < size; i++)  {
      MPI_Recv(&r, 1, MPI_INTEGER, MPI_ANY_SOURCE, tag,
                                           MPI_COMM_WORLD, &status);
      printf("Message recieved:  %d\n", r[0]);
    }
  }
  MPI_Finalize();                                             // Finalize MPI
  return 0;
}
```

A bit more advanced use of message passing is given by `message2.c` in Listing 22.4. It has each guest processor sending a message to the host processor, who then prints out the rank of the guest which sent the message. Observe how we have the host looping through all the guests; otherwise it would stop looking for more messages after the first one arrives. In order to know how the number of processors, the host calls `MPI_Comm_size`.

22.2.6
Broadcast Messages: MPIpi.c

If we used the same technique to send a message from one node to several other nodes, we would have to loop over calls to `MPI_Send`. In `MPIpi.c` in Listing 22.5 we see an easy way to send to all other nodes.
This is a simple program that computes π in parallel using the "stone throwing" technique discussed in Chap 11. Notice the new MPI commands:

- **`MPI_Wtime`** is used to return the wall time in seconds (the time as given by a clock on the wall). This is useful when computing speedup curves (Figs. 22.2 and 22.3).

- **`MPI_Bcast`** *broadcasts* sends out data from one processor to all others. In our case the host broadcasts the number of iterations to the guests, which, in turn, replace their current values of `n` with the one received from the host.

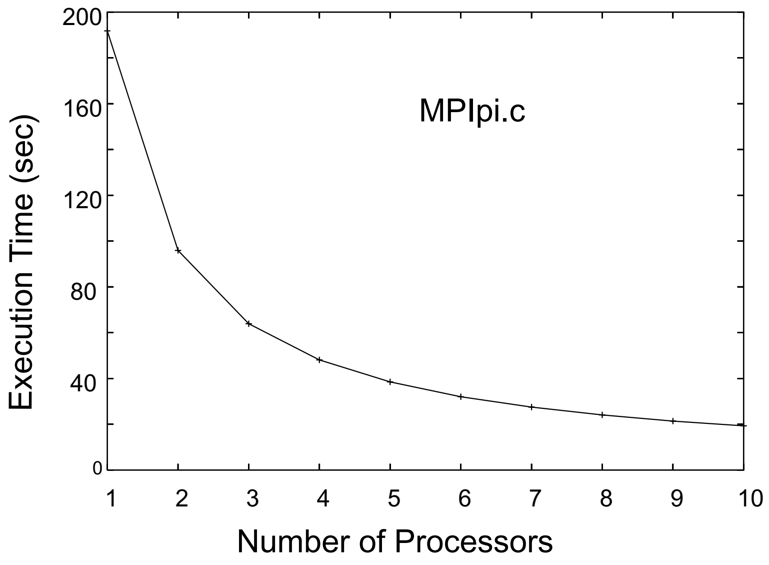

Fig. 22.2 Execution time versus number of processors for the calculation of π with `MPIpi.c`.

• **`MPI_Allreduce`** is a glorified broadcast command. It collects the values of the variable `mypi` from each of the processors, performs an operation on them with `MPI_SUM`, and then broadcasts the result via the variable `pi`.

Listing 22.5: The C program `MPIpi.c` uses a number of processors to compute π by a Monte Carlo rejection technique (stone throwing).

```
// MPIpi.c computes pi in parallel by Monte Carlo Integration
#include "mpi.h"  #include <stdio.h>  #include <math.h>
double f(double);

int main(int argc,char *argv[]) {

  int    n, myid, numprocs, i;
  double PI25DT = 3.141592653589793238462643;
  double mypi, pi, h, sum, x, startwtime=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. / (double) n;
  sum = 0.;
  for (i = myid + 1; i <= n; i += numprocs) {  // Better to work back
    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;
}

double f(double a) { return (4./(1. + a*a));}        // Function f(a)
```

22.2.7 Exercise

In Fig. 22.2 we show our results for the speedup obtained by calculating π in parallel with `MPIpi.c`. This exercise leads you through the steps in obtaining your own speedup curve:

1. Record how long each of your runs takes and how accurate the answers are. Does the roundoff error enters in? What could you do to get a more accurate value for π?
2. There are two version of a parallel program possible here. In the *active host* version the host acts just like a guest and does some work. In the *lazy host* version the host does no work but instead just controls the action. Does `MPIpi.c` contain an active or lazy host? Change `MPIpi.c` to the other version and record the difference in execution times.
3. Make a plot of the time versus number of processors for the calculation of π.
4. Make a *speedup* plot, that is, a graph of computation time divided by the time for one processor, versus the number of processors. ☐

Listing 22.6: The program `TuneMPI.c` is a parallel version of our program `Tune.java` used to test the effects of various optimization modifications.

```
/* TuneMPI.c: a matrix algebra program to be tuned for performace
              N X N Matrix speed tests using MPI  */

#include "mpi.h"
#include <stdio.h>
```

```
#include <time.h>
#include <math.h>

int main(int argc,char *argv[]) {

   int N=200, MAX = 15;   //2051
   double ERR = 1.0e-6;
   MPI_Status status;
   double dummy = 2.;
   int h = 1, myrank, nmach;
   int i, j, k, iter = 0;
   time_t systime_i, systime_f;
   double timempi[2];
   double ham[N][N];
   double coef[N];
   double sigma[N];
   double ener[1];
   double err[1];
   double ovlp[1];
   double mycoef[1];
   double mysigma[1];
   double myener[1];
   double myerr[1];
   double myovlp[1];
   double step=0.0;
   long difftime=0l;
                                                    // MPI Initialization
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
   MPI_Comm_size(MPI_COMM_WORLD,&nmach);
   MPI_Barrier(MPI_COMM_WORLD);
   if (myrank == 0) {
     timempi[0] = MPI_Wtime();
                                                     // Store initial time
     systime_i = time(NULL);
   }
                                  // set up Hamiltonian and starting vector
   printf("\n\t Processor %d checking in...\n", myrank);
   fflush(stdout);
   for (i = 1; i < N; i++) {
     for (j = 1; j < N; j++){
       if (abs(j-i) >10) {ham[j][i] = 0.0;}
       else  ham[j][i] = pow(0.3, abs(j-i));
     }
     ham[i][i] = i;
     coef[i] = 0.0;
  }
  coef[1] = 1.0  ;
                                 //  start iterating toward the solution
  err[0] = 1.0;
  iter = 0 ;
  if (myrank==0)  {
    printf("\nIteration #\tEnergy\t\tERR\t\tTotal Time\n");
    fflush(stdout);
  }
  while ((iter < MAX)&&(err[0] > ERR))   {
                                                      // start while loop
```

```
iter = iter + 1;
mycoef[0]=0.0;
ener[0] = 0.0; myener[0] = 0.0 ;
ovlp[0] = 0.0; myovlp[0] = 0.0 ;
err[0] = 0.0  ; myerr[0] = 0.0;
for (i= 1; i < N; i++)  {
  h = (int)(i)%(nmach-1)+1 ;
  if (myrank == h){
    myovlp[0] = myovlp[0]+coef[i]*coef[i];
    mysigma[0] = 0.0;
    for ( j= 1; j < N; j++) {
      mysigma[0] =  mysigma[0] + coef[j]*ham[j][i];
    }
    myener[0] =  myener[0]+coef[i]*mysigma[0] ;
    MPI_Send(&mysigma, 1, MPI_DOUBLE, 0, h, MPI_COMM_WORLD);
  }
  if (myrank == 0)  {
     MPI_Recv(&mysigma,1, MPI_DOUBLE,h,h,MPI_COMM_WORLD,&status);
     sigma[i]=mysigma[0];
  }
}                                            // end of for(i...
MPI_Allreduce(&myener,&ener,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
MPI_Allreduce(&myovlp,&ovlp,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
MPI_Bcast(&sigma, N-1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
ener[0] = ener[0]/(ovlp[0]);
for (  i = 1; i< N; i++) {
  h = (int)(i)%(nmach-1)+1 ;
  if (myrank == h)  {
    mycoef[0] = coef[i]/sqrt(ovlp[0]);
    mysigma[0] = sigma[i]/sqrt(ovlp[0]);
    MPI_Send(&mycoef, 1, MPI_DOUBLE,0,nmach+h+1, MPI_COMM_WORLD);
    MPI_Send(&mysigma,1,MPI_DOUBLE,0,2*nmach+h+1,MPI_COMM_WORLD);
  }
  if (myrank == 0)  {
    MPI_Recv(&mycoef, 1, MPI_DOUBLE, h, nmach+h+1,
    MPI_COMM_WORLD, &status);
    MPI_Recv(&mysigma, 1, MPI_DOUBLE, h, 2*nmach+h+1,
    MPI_COMM_WORLD, &status);
    coef[i]=mycoef[0];
    sigma[i]=mysigma[0];
  }
} // end of for(i...
MPI_Bcast(&sigma, N-1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&coef, N-1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
for (i = 2; i < N ; i++) {
  h = (int)(i)%(nmach-1)+1;
  if (myrank == h) {
    step = (sigma[i] - ener[0]*coef[i])/(ener[0]-ham[i][i]);
    mycoef[0] = coef[i] + step;
    myerr[0] =  myerr[0]+ pow(step,2);
    for ( k= 0; k <= N*N; k++) {                    // slowdown loop
      dummy = pow(dummy,dummy);
      dummy = pow(dummy,1.0/dummy);
    }
    MPI_Send(&mycoef,1,MPI_DOUBLE,0,3*nmach+h+1, MPI_COMM_WORLD);
  }                                          // end of if(myrank..
  if (myrank == 0)  {
        MPI_Recv(&mycoef, 1, MPI_DOUBLE, h, 3*nmach+h+1,
                                          MPI_COMM_WORLD, &status);
        coef[i]=mycoef[0];
      }
    }                                              // end of for(i...
    MPI_Bcast(&coef, N-1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Allreduce(&myerr,&err, 1,MPI_DOUBLE,MPI_SUM, MPI_COMM_WORLD);
    err[0] = sqrt(err[0]);
    if (myrank==0) {
      printf("\t#%d\t%g\t%g\n", iter, ener[0], err[0]);
      fflush(stdout);
    }
  }                                                 // end whileloop
                                               // output elapsed time
  if (myrank == 0) {
     systime_f = time(NULL);
     difftime = ((long) systime_f) - ((long) systime_i);
     printf("\n\tTotal wall time = %d s\n", difftime);
     fflush(stdout);
     timempi[1] = MPI_Wtime();
     printf("\n\tMPItime= %g s\n", (timempi[1]-timempi[0]) );
     fflush(stdout);
  }
  MPI_Finalize();
}
```

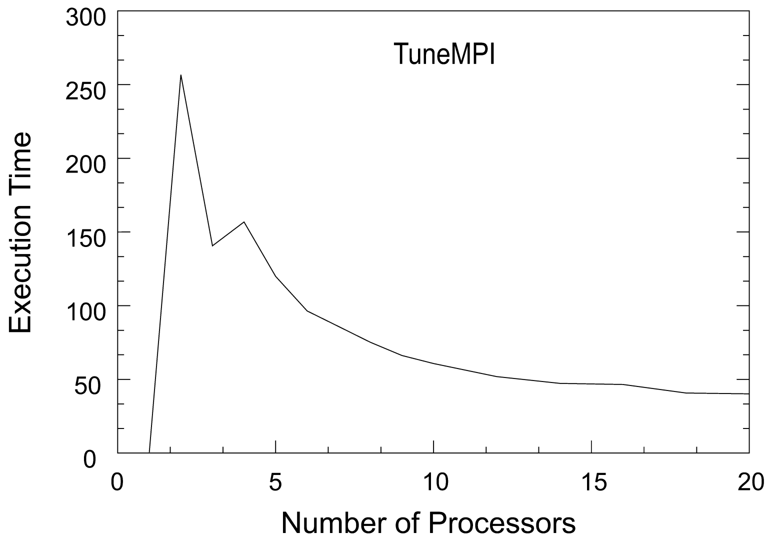

Fig. 22.3 Execution time versus number of processors for the solution of an eigenvalue problem with `TuneMPI.c`. Note, the single processor result here does *not* include the overhead of running MPI.

22.3 Parallel Tuning: TuneMPI.c

Recall the `Tune` program that we experimented with in Chap. 13 to determine how memory access for a large matrix affects the running time of programs. You may also recall that as the size of the matrix was made larger, the execution time increased more rapidly than did the number of operations the program had to perform. The extra increase came from the time it took to transfer the needed matrix elements in and out of central memory. Because parallel programming on a multiprocessor also involves a good deal of data transfer, `Tune` program is also a good teaching tool for seeing how communications costs affect parallel computations.

In Listing 22.6 is the program `TuneMPI.c`. In Fig. 22.3 is the speedup curve we obtained by running it. This is a modified version of the `Tune` program in which each row of the large matrix multiplication is performed on a different processor using MPI. Explicitly,

$$[H]_{N\times N} \qquad \times \qquad [\Psi]_{N\times 1} \tag{22.1}$$

$$\begin{bmatrix} \Rightarrow & \text{rank } 1 & \Rightarrow \\ \Rightarrow & \text{rank } 2 & \Rightarrow \\ \Rightarrow & \text{rank } 3 & \Rightarrow \\ \Rightarrow & \text{rank } 1 & \Rightarrow \\ \Rightarrow & \text{rank } 2 & \Rightarrow \\ \Rightarrow & \text{rank } 3 & \Rightarrow \\ \Rightarrow & \text{rank } 1 & \Rightarrow \\ & \ddots & \end{bmatrix}_{N\times N} \times \begin{bmatrix} \psi_1 & \Downarrow \\ \psi_2 & \Downarrow \\ \psi_3 & \Downarrow \\ \psi_4 & \Downarrow \\ \psi_5 & \Downarrow \\ \psi_6 & \Downarrow \\ \psi_7 & \Downarrow \\ \ddots & \end{bmatrix}_{N\times 1} \tag{22.2}$$

where the arrows indicate how each row of H gets multiplied by the single column of Ψ, with the multiplication of each row performed on a different processor (rank). The assignment of rows to processors continues until we run out of processors, and then starts all over again. Since this multiplication gets repeated for a number of iterations, this is the most computationally intensive part of the program, and so it makes sense to parallelize it.

However, even if the matrix is large, the `Tune` program is not computationally intensive enough to overcome the cost of communication inherent in parallel computing. Consequently, to increase computing time we have inserted an inner `for` loop over `k` that takes up time but accomplishes nothing (we have all had days like that). Slowing down the program should help make the speedup curve more realistic.

22.3.0.1 TuneMPI.c Exercise

1. Compile `TuneMPI.c`:

```
> mpicc TuneMPI.c -lm -o TuneMPI
```

Compilation

Here `-lm` loads the math library and `-o` places the object in `TuneMPI`. This is your base program. It will use one processor as the host and another one to do the work.

2. To determine the speedup with multiple processors, you need to change the `runMPI.sh` script. Open it with an editor and find the line of the form:

```
#$ -pe class_mpi 1-4                    A line in runMPI.sh script
```

The last number on this line tells the cluster the maximum number of processors to use. Change this to the number of processors you want to use. Use a number from 2 up to 10; starting with one processor leads to an error message as that leaves no processor to do the work. After changing `runMPI.sh`, run the program on the cluster. With the SGE management system this is done via

```
> qsub runMPI.sh TuneMPI                Submit to queue via SGE
```

3. You are already familiar with the scalar version of the `Tune` program. Find the scalar version of `Tune.c` (and add the extra lines to slow the program down), or modify the present one so that it runs on only one processor. Run the scalar version of `TuneMPI` and record the time it takes. Because there is overhead associated with running MPI, we would expect the scalar program to be faster than an MPI program running on a single processor.

4. Open another window and watch the processing of your MPI jobs on the host computer. Check that all temporary files are removed.

5. You now want to collect data for a plot of running time versus number of machines. Make sure your matrix size is large, say with `N=200` and up. Run `TuneMPI` on a variable number of machines, starting at 2, until you find no appreciable speedup (or an actual slowdown) with an increasing number of machines.

6. *Warning:* While you will do no harm running on the Beowulf when others are also running on it, in order to get meaningful and repeatable speedup graphs, you need to have the cluster all to yourself. Otherwise, the time it takes to switch around jobs and to setup and drop communications may slow down your runs significantly. A management system should help with this. If you are permitted to log in directly to the Beowulf machines, you can check what is happening via `who`:

```
> rsh rose who                          Who is running on rose?
> rsh rubin who                         Who is running on rubin?
> rsh emma who                          Who is running on emma?
```

7. Increase the matrix size in steps and record how this affects the speedups. Remember, once your code is communications bound due to memory access, distributing it over many processors probably will make things worse!

22.4 A String Vibrating in Parallel

Listing 22.7: The solution of the sting equation on several processors via MPIstring.c.

```
// Code listing for MPIstring.c

#include <stdio.h>
#include <math.h>
#include "mpi.h"
#define maxt 10000                     // Number of time steps to take
#define L 10000                    // Number of divisions of the string
#define rho 0.01                          // Density per length (kg/m)
#define ten 40.0                                          // Tension (N)
#define deltat 1.0e-4                                       // Delta t (s)
#define deltax .01                                         // Delta x (m)
#define skip 50        // Number of time steps to skip before printing

/* Need sqrt(ten/rho) <= deltax/deltat for a stable solution
        Decrease deltat for more accuracy, c' = deltax/deltat   */

main(int argc, char *argv[]) {

  const double scale = pow(deltat/deltax,2)*ten/rho;
  int i, j, k,myrank, numprocs, start, stop, avgwidth, maxwidth, len;
  double left, right, startwtime, init\_string(int index);

  FILE *out;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);                  // Get my rank
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);      // Number of processors
  MPI_Status status;
    if (myrank == 0)  {
      startwtime = MPI_Wtime();
      out = fopen("eqstringmpi.dat","w");
    }
    // assign a string to each node
    // 1st and last points (0 and L-1) -must =0, else error
    // Thus L-2 segments for numprocs processors
    avgwidth = (L-2)/numprocs;
    start = avgwidth*myrank+1;
    if (myrank < numprocs - 1)  stop = avgwidth*(myrank+1);
    else stop = L-2;
    if (myrank == 0)   maxwidth = L-2 - avgwidth*(numprocs-1);
    else maxwidth = 0;
    double results[maxwidth];                // Holds print for master
    len = stop - start;                     // Length of the array - 1
    double x[3][len+1];
    for (i=start; i <= stop; i++)   x[0][i-start] = init_string(i);
    x[1][0] = x[0][0]+0.5*scale*(x[0][1]
                               +init_string(start-1)-2.*x[0][0]);
    x[1][len] = x[0][len]                             // 1st time step
        + 0.5*scale*(init_string(stop+1)+x[0][len-1]-2.0*x[0][len]);
    for (i=1; i < len; i++)
      x[1][i] = x[0][i]+0.5*scale*(x[0][i+1]+x[0][i-1]-2.0*x[0][i]);
    for(k=1; k<maxt; k++) {                          //  Later time steps
      if (myrank == 0) {                         // Send to R, get from L
```

```
        MPI_Send(&x[1][len],1,MPI_DOUBLE,1,1,MPI_COMM_WORLD);
        left = 0.0;
      }
      else if (myrank < numprocs - 1) MPI_Sendrecv(&x[1][len],1,
      MPI_DOUBLE, myrank+1,1,&left,1,MPI_DOUBLE,myrank-1,1,
                                            MPI_COMM_WORLD, &status);
      else MPI_Recv(&left,1,MPI_DOUBLE,myrank-1,
                                         1,MPI_COMM_WORLD,&status);
      if (myrank == numprocs - 1) {            // Send to L & get from R
        MPI_Send(&x[1][0],1,MPI_DOUBLE,myrank-1,2,MPI_COMM_WORLD);
        right = 0.0;
      }
      else if (myrank>0) MPI_Sendrecv(&x[1][0],1,MPI_DOUBLE,myrank-1,
             2,&right,1,MPI_DOUBLE,myrank+1,2,MPI_COMM_WORLD,&status);
      else MPI_Recv(&right,1,MPI_DOUBLE,1,2,MPI_COMM_WORLD,&status);
      x[2][0] = 2.0*x[1][0]-x[0][0]+scale*(x[1][1]+left-2.0*x[1][0]);
      for(i = 1; i < len; i++)  x[2][i] = 2.0*x[1][i]-x[0][i]
                            + scale*(x[1][i+1]+x[1][i-1]-2.0*x[1][i]);
      x[2][len] = 2.0*x[1][len]-x[0][len]
                             + scale*(right+x[1][len-1]-2.0*x[1][len]);
      for(i = 0; i <= len; i++)  {
        x[0][i] = x[1][i];
        x[1][i] = x[2][i];
      }
      if((k%skip) == 0) {          // Print using gnuplot 3D grid format
        if (myrank != 0)
              MPI_Send(&x[2][0],len+1,MPI_DOUBLE,0,3,MPI_COMM_WORLD);
        else {
          fprintf(out,"%f\n",0.0);            // Left edge of (always 0)
          for (i=0; i < avgwidth; i++) fprintf(out,"%f\n",x[2][i]);
          for (i=1; i < numprocs-1; i++) {
            MPI_Recv(results,avgwidth,MPI_DOUBLE,i,3,MPI_COMM_WORLD,
                                &status);
            for (j=0;j<avgwidth;j++) fprintf(out,"%f\n",results[j]);
          }
          MPI_Recv(results,maxwidth,MPI_DOUBLE,numprocs-1,3,
                                              MPI_COMM_WORLD,&status);
          for (j=0; j < maxwidth; j++)
              fprintf(out,"%f\n",results[j]);
          fprintf(out,"%f\n",0.0);                            // R edge
          fprintf(out,"\n");                 // Empty line for gnuplot
        }
      }
    }
    if (myrank == 0)
      printf("Data in eqstringmpi.dat\nComputation time: %f s\n",
                                              MPI_Wtime()-startwtime);
      MPI_Finalize();
      exit(0);
}

  double init_string(int index) {
    if (index < (L-1)*4/5) return 1.0*index/((L-1)*4/5);
    return 1.0*(L-1-index)/((L-1)-(L-1)*4/5);
    // Half of a sine wave
  }
```

The program `MPIstring.c` given in Listing 22.7 is a parallel version of the solution of the wave equation (`eqstring.c`) discussed in Chap. 25. The algorithm calculates the future ([2]) displacement of a given string element from the present ([1]) displacements immediately to the left and right of that section, as well as the present and past ([0]) displacements of that element. The program is parallelized by assigning different sections of the string to different nodes, and by having the nodes communicate the displacements at their endpoints via the `MPI_Send()` and `MPI_Recv()`. The first argument to `MPI_Send()` are *pointer*s to the data being sent, and the last arguments are of the type `MPI_Status`. A pointer in C, denoted by the ampersand `&`, gives the address of the variable that follows. The `MPI_Recv()` and `MPI_Send()` commands require a pointer as their first argument, or an array element (which in C is the address of the first element in the array). When sending more than one element of an array to `MPI_Send()`, you send a pointer to the first element of the array as the first argument, and then the number of elements as the second argument.

Observe near the end of the program how the `MPI_Send()` call is used to send `len+1` elements of the two-dimensional array `x[][]`, starting with the element `x[2][0]`. MPI will send these elements in the order in which they are stored in memory. In C, arrays are stored in row-major order with the first element of the second row immediately following the last element of the first row, and so on. Accordingly, this `MPI_Send()` call sends `len+1` elements of row 2, starting with column 0, which means all of row 2. If we had specified `len+5` elements instead, MPI would have sent all of row 2, plus the first four elements of row 3.

In `MPIstring.c`, the calculated future position of the string is stored in row 2 of `x[3][len+1]`, with different sections of the string stored in different columns. Row 1 stores the present positions, and row 0 stores the past positions. This is different from column-based algorithm used in the serial program `eqstring.c`, which followed the original Fortran program, which was column, rather than row based. This was changed because MPI reads data by rows.

The initial displacement of the string is given by the user-supplied function `init_string()`. Because the first time step requires only the initial displacement, no message passing is necessary. For later times each node sends the displacement of its rightmost point to the node with the next highest rank. This means that the rightmost node (`rank = numprocs - 1`) does not send data, and that the master (`rank = 0`) does not receive data. Communication is established via the `MPI_Sendrecv()` command, with the different `send`s and `receive`s using tags to ensure proper delivery of the messages.

Next in the program, the nodes (representing string segments) send to and receive data from the segment to their right. All of these sends and receives have a tag of 2. After every 50 iterations, the master collects the displacement of each segment of the string and outputs it. Each slave sends the data for the future time with a tag of 3. The master first outputs its own data and then calls `MPI_Recv()` for each node, one at a time, printing the data it receives.

22.4.1 MPIstring.c Exercise

1. Ensure that the input data (`maxt, L, scale, skip`) in `MPIstring.c` are the same as those in `eqstring.c` (`maxt` in `MPIstring.c` is called `max` in `eqstring.c`).
2. Ensure that `init_string()` returns the initial configuration that is used in `eqstring.c`.
3. Compile and run `eqstring.c` via

   ```
   > gcc eqstring.c -o eqstring -lm          Compile C code
   > ./eqstring                               Run in present directory
   ```

4. Run both programs to ensure that they produce the same output. (This is easy to check in Unix with the `diff` command.)

Listing 22.8: The MPI program `MPIdeadlock.c` illustrates deadlock (waiting for receive). The code `MPIdeadlock-fixed.c` in Listing 22.9 removes the block.

```
// Code listing for MPIdeadlock.c
#include <stdio.h>
#include "mpi.h"
#define MAXLEN 100
main(int argc, char *argv[]) {

    int myrank, numprocs, fromrank, torank;
    char tosend[MAXLEN], received[MAXLEN];
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    if (myrank == 0) fromrank = numprocs - 1;
    else fromrank = myrank - 1;
    if (myrank == numprocs - 1) torank = 0;
    else torank = myrank + 1;
    // Save string to send in tosend:
    sprintf(tosend,"Message from node %d to %d\n", myrank,torank);
    MPI_Recv(received,MAXLEN,MPI_CHAR,fromrank,0,MPI_COMM_WORLD,
                                          &status);
    MPI_Send(tosend,MAXLEN,MPI_CHAR,torank,0,MPI_COMM_WORLD);
    printf("%s",received);    // Print string after successful receive
    MPI_Finalize();
    exit(0);
}
```

22.5 Deadlock

It is important to avoid *deadlock* when using the `MPI_Send()` and `MPI_Recv()` commands. Deadlock occurs when one or more nodes wait for a nonoccuring event to occur. This can arise if each node waits to receive a message that is not sent. Compile and execute `deadlock.c`:

```
> mpicc deadlock.c -o deadlock                    Compile
> qsub run_mpi.sh deadlock                        Execute
```

Take note of the job id returned, which we will call `xxxx`. Wait for a few seconds, and then look at the output of the program

```
> cat output/MPI_job-xxxx                         Examine output
```

The output should list how many nodes ("slots") were assigned to the job. Because all these nodes are now deadlocked, we need to cancel this job:

```
> qdel xxxx                                       Cancel deadlocked job
> qdel all                                        Alternate cancel
```

There are a number of ways to avoid deadlock. The program `MPIstring.c` used the function `MPI_Sendrecv()` to handle much of the message passing, and this will not cause deadlock. It is possible to use `MPI_Send()` and `MPI_Recv()`, but you should be careful to avoid deadlock, as we do in `MPIdeadlock-fixed.c`.

Listing 22.9: The code `MPIdeadlock-fixed.c` removes the deadlock present in `MPIdeadlock.c`.

```
// deadlock-fixed.c: deadlock.c without deadlock by Phil. Carter

#include <stdio.h>
#include "mpi.h"
#define MAXLEN 100

main(int argc, char *argv[])  {

  int myrank, numprocs, torank, i;
  char tosend[MAXLEN], received[MAXLEN];
  MPI_Status status;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  if (myrank == numprocs - 1) torank = 0;
  else torank = myrank + 1;
  // Save the string to send in the variable tosend:
  sprintf(tosend,"Message from node %d to %d\n", myrank, torank);
  for (i = 0; i < numprocs; i++)  {
    if (myrank == i)
            MPI_Send(tosend,MAXLEN,MPI_CHAR,torank,i,MPI_COMM_WORLD);
    else if (myrank == i+1 || (i == numprocs - 1 && myrank == 0))
       MPI_Recv(received,MAXLEN,MPI_CHAR,i,i,MPI_COMM_WORLD,&status);
    }
    printf("%s",received);          // Print after successful receive
    MPI_Finalize();
    exit(0);
}
```

22.5.1 Nonblocking Communication

`MPI_Send()` and `MPI_Recv()`, as we have seen, are susceptible to deadlock because they block the program from continuing until the send or receive is finished. This method of message passing, called *blocking communication*. One way to avoid deadlock is to use nonblocking communication such as `MPI_Isend()`, which returns before the send is complete and thus frees up the node. Likewise, a node can call `MPI_Irecv()`, which does not wait for the receive to complete. Note that a node can receive a message with `MPI_Recv()` even if were sent using `MPI_Isend()`, and similarly, receive a message using `MPI_Irecv()` even if were sent with `MPI_Send()`.

There are two ways to determine whether a nonblocking send or receive has finished. One is to call `MPI_Test()`. It is then your choice as to whether you want to wait for the communication to complete (for example, to ensure that all string segments are at current time and not past time). To wait, call `MPI_Wait()`, which blocks execution until communication is complete. When you start a nonblocking send or receive, you get a request handle of data type `MPI_Request` to identify the communication you may need to wait for. When using nonblocking communication, you do have to ensure that you do not use the data being communicated until the communication has completed. You can check for this using `MPI_Test()`, or wait for completion using `MPI_Wait()`.

Exercise: Rewrite `deadlock.c` so that it avoids deadlock by using nonblocking communication. *Hint:* replace `MPI_Recv()` by `MPI_Irecv()`. □

22.5.2 Collective Communication

Several MPI commands that automatically exchange messages among multiple nodes at the same time. This is called collective communication, as opposed to the point-to-point communication between just two nodes. The program `MPIpi.c` already introduced the `MPI_Reduce()` command. It receives data from multiple nodes, performs an operation on the data, and stores the result on one node. The program `tunempi.c` used a similar function `MPI_Allreduce()` that does the same thing, but stores the result on every node. The latter program also used `MPI_Bcast()`, which allows a node to send the same message to multiple nodes.

Collective commands communicate among a group of nodes specified by a communicator, such as `MPI_COMM_WORLD`. For example, in `MPIpi.c` we called `MPI_Reduce()` to receive results from every node, including itself. Other collective communication functions include `MPI_Scatter()` that has one node send messages to every other node. This is similar to `MPI_Bcast()`, but the former sends a different message to each node. Specifically, it breaks an array up into pieces of specified lengths, and sends the different pieces to different nodes. Likewise, `MPI_Gather()` gathers data from every node (including the root node), and places the data into an array, with data from node 0 placed first, followed by node 1, etc. A similar function, `MPI_Allgather()`, stores the data on every node, rather than just the root node.

22.6 Supplementary Exercises

1. **Bifurcation Plot:** If you have not yet done it, take the program you wrote to generate the bifurcation plot for bug populations and run different ranges of μ values simultaneously on several CPUs.
2. **Processor Ring:** Write a program in which
   (a) a set of processes are arranged in a ring,
   (b) each process stores its rank in `MPI_COMM_WORLD` in an integer,
   (c) each process passes this on to its neighbor on the right,
   (d) each processor keeps passing until it receives its rank back.
3. **Ping Pong:** Write a program in which two processes repeatedly pass a message back and forth. Insert timing calls to measure the time taken for one message, and determine how the time taken varies with the size of the message.
4. **Broadcast:** Have processor 1 send the same message to all the other processors and then receive messages of the same length from all the other processors. How does the time taken vary with the size of the messages and the number of processors?

22.7 List of MPI Commands

MPI Data Types and Operators

MPI defines some of its own data types. The following are data types used as arguments to the MPI commands.

- **MPI_Comm:** A communicator, used to specify group of nodes, most commonly `MPI_COMM_WORLD` for all the nodes.
- **MPI_Status:** A variable holding status information returned by functions such as `MPI_Recv()`.
- **MPI_Datatype:** Predefined constant indicating type of data being passed in a function such as `MPI_Send()` (see below).
- **MPI_O:** Predefined constant indicating operation you want performed on data in functions such as `MPI_Reduce()` (see below).
- **MPI_Request:** Request handle to identify a nonblocking `send` or `receive`, for example, when using `MPI_Wait()` or `MPI_Test()`>

Predefined constants: MPI_Op and MPI_Datatype

For a more complete of constants used in MPI, see
`http://www-unix.mcs.anl.gov/mpi/www/www3/Constants.html.`

MPI_OP	Description	MPI_Datatype	C Data Type
MPI_MAX	Maximum	**MPI_CHAR**	char
MPI_MIN	Minimum	**MPI_SHORT**	short
MPI_SUM	Sum	**MPI_INT**	int
MPI_PROD	Product	**MPI_LONG**	long
MPI_LAND	Logical and	**MPI_FLOAT**	float
MPI_BAND	Bitwise and	**MPI_DOUBLE**	double
MPI_LOR	Logical or	**MPI_UNSIGNED_CHAR**	unsigned char
MPI_BOR	Bitwise or	**MPI_UNSIGNED_SHORT**	unsigned short
MPI_LXOR	Logical exclusive or	**MPI_UNSIGNED**	unsigned int
MPI_BXOR	Bitwise exclusive or	**MPI_UNSIGNED_LONG**	unsigned long
MPI_MINLOC	Find node's min and rank		
MPI_MAXLOC	Find node's max and rank		

MPI Commands

Below we list and identify the MPI commands used in this chapter. For the syntax of each command, along with many more MPI commands, see http://www-unix.mcs.anl.gov/mpi/www/. There each command name is a hyperlink to a full description.

Basic MPI Commands

- **MPI_Send.** Sends a message.

- **MPI_Recv.** Receives a message.
- **MPI_Sendrecv.** Sends and receives a message.
- **MPI_Init.** Starts MPI at the beginning of program.
- **MPI_Finalize.** Stops MPI at the end of program.
- **MPI_Comm_rank.** Determine a node's rank.
- **MPI_Comm_size.** Determine number of nodes in communicator.
- **MPI_Get_processor_name.** Determines name of the processor.
- **MPI_Wtime.** Returns wall time in seconds since arbitrary time in the past.
- **MPI_Barrier.** Blocks until all nodes have called this function.

Collective Communication

- **MPI_Reduce.** Performs operation on all copies of variable and stores result on single node.
- **MPI_Allreduce.** Like `MPI_Reduce`, but stores result on all nodes.
- **MPI_Gather.** Gathers data from group of nodes and stores them on one node.
- **MPI_Allgather.** Like `MPI_Gather`, but stores the result on all nodes
- **MPI_Scatter.** Sends different data to all other nodes (opposite of `MPI_Gather`).
- **MPI_Bcast.** Sends same message to all other processes.

Nonblocking Communication

- **MPI_Isend.** Begins a nonblocking `send`.
- **MPI_Irecv.** Begins a nonblocking `receive`.
- **MPI_Wait.** Waits for an MPI `send` or `receive` to complete.
- **MPI_Test.** Tests for the completion of a `send` or `receive`.