Beginner's Guide to MPI (MPICH-v1.2.5) on the University of Delaware CIS Linux Cluster

Dixie Hisley and Lori Pollock
Department of Computer and Information Sciences
University of Delaware
(Revised September 2006 by Lori Pollock)

August 30, 2006

Contents

1 Introduction

This is a manual for MPI, with local information for using MPI on the University of Delaware's CIS linux cluster. The cluster consists of one single processor host node (porsche) and 8 dual CPU rack mounted PC's. All are pentium III's and all run linux.

MPI, the *Message Passing Interface*, is a library, and a software standard developed by the MPI Forum to make use of the most attractive features of existing message passing systems for parallel programming. Important contributions have come from the IBM T. J. Watson Research Center, Intel's NX/2, Express, nCUBE's Vertex, p4, PARMACS, Zipcode, Chimp, PVM, Chameleon, and PICL.

An MPI process consists of a C, C++, or Fortran 77 program which communicates with other MPI processes by calling MPI routines. The MPI routines provide the programmer with a consistent interface across a wide variety of different platforms. The MPI specification is based on a message passing model of computation where processes communicate and coordinate by sending and receiving messages. These messages can be of any size, up to the physical memory limit of the machine. MPI provides a variety of message passing options, offering maximal flexibility in message passing.

MPI is a specification (like C or Fortran) and there are a number of implementations. This guide describes the basic use of the MPICH implementation of MPI. Other implementations include LAM and CHIMP versions of MPI. These implementations are freely available by anonymous ftp from www-unix.mcs.anl.gov/mpi.

The MPICH implementation is a library of several hundred C and Fortran routines that will let you write programs that run in parallel and communicate with each other. Few completely understand all that any MPI implementation offers, but that's okay, because this class will only be using some ten (maybe a few more) routines out of the bunches available.

This guide is designed to give the user a brief overview of some of the basic and important routines of MPICH with emphasis on getting up and running on the Dell dual Intel pentium III's at the University of Delaware. The machine porsche.cis.udel.edu is the host and the only system accessible from the Internet. The other 8 machines are backend nodes and have two CPUs each. They are honda, toyota, subaru, nissan, hyundai, acura, mazda, and suzuki. They may also be referred to by aliases node1, node2, ..., node8, respectively. Porsche is aliased to node0.

This guide is not meant to be a replacement for the official manual or user's guide. You should follow the various links on the course web page for the manual, user's guide, MPI Standard, and MPI FAQ.

2 Getting Started With MPI on the Cluster

This section contains the steps necessary to configure your MPI environment, and to compile and run an MPI program.

2.1 Editing Files on porsche

There are three different editors available on the Cluster: `emacs, vi,` and `pico`. Pick your favorite editor for your course projects.

2.2 Logging In & Setting Up Your Environment

2.2.1 Logging In

To log in to porsche from any machine on UDelNet, type the following at the command prompt:

```
ssh porsche.cis.udel.edu
```

Once logged in, ssh sets up your DISPLAY environment and uses X-Authentication to handle any X-Windows you might create during your session. If you have never used ssh to connect with the machine, you will be asked about an authentication key. It is okay to type "yes" at this prompt. This will only happen the first time you ssh to connect to porsche. After the authentication prompt (if any), ssh should prompt for your password on porsche. The first time you execute a parallel program on the cluster, you will most likely also get prompted for your password on every machine in the cluster.

2.2.2 .cshrc/.tcshrc file modifications

To be sure that mpirun, mpicc, mpif77, and mpiCC are all in your search path, put `/usr/local/bin /usr/local/mpich/bin` in your path.

2.2.3 Avoiding the Annoying Password Prompt

While you are on porsche, type:

```
cd ~/.ssh
ssh-keygen -t rsa
```

It will ask for a file to store the key in. Take the default (id_rsa) It will then ask you for passphrase (which can be longer than a password). Here just hit enter if you want passwordless entry. It will ask you to verify the passphrase. Just hit enter again.

```
cp id_rsa.pub authorized_keys
```

ssh-keygen will create two files: id_rsa and id_rsa.pub. This is a public/private key pair with the public key being in the id_rsa.pub file (in case that wasn't obvious). When you do the cp above make sure you use the .pub file and not the other one.

If you already have a authorized_keys file then add the contents of id_rsa.pub to it rather than just overwriting it.

Now anytime you ssh from porsche it will use the key to identify you. Since that key matches what we have in authorized_keys it will allow you in without a password.

One more note: If you mess up the permissions of any file in .ssh then this passwordless entry will stop working (ssh is paranoid). You'll be able to tell because ssh will scream about it if when you try to log in. If you've botched then do the following:

```
cd .ssh
chmod 600 *
chmod 644 authorized_keys id_rsa.pub
```

and then things should be fine.

2.2.4 Setting Up to Display Graphics to Your Local Screen

There is nothing you need to do special for this. ssh takes care of this for you.

2.3 Compilation

MPI allows you to have your source code in any directory. For convenience, you should probably put them in subdirectories under `~yourusername/mpi`.

You can compile simple C programs that call MPI routines with:

```
mpicc -o program_name program_name.c -lm
```

(where `-lm` links in the math library)

You can compile simple CC programs that call MPI routines with:

```
mpiCC -o program_name program_name.c -lm
```

(where `-lm` links in the math library)

For simple C programs that use MPE graphics, you can compile with:

```
mpicc -o program\_name program\_name.c -lmpe -lX11 -lm
```

You need to make sure the loader can find the X11 libraries at /usr/local/X11R6/lib. One way to do this is to execute the mpicc command the first time with:

```
mpicc -o program_name program_name.c -lmpe -L/usr/local/X11R6/lib -lX11 -lm
```

Fortran compilation is performed similarly; exchange `mpif77` for `mpicc` and program_name.f for program_name.c. Type `mpicc -help` or `mpif77 -help or mpiCC -help` for additional information.

It might be good to create a makefile for compiling your MPI programs.

2.4 Creating a machinefile for Running Your MPI Program

A `machinefile` is a file that contains a list of the possible machines on which you want your MPI program to run. This file is useful if one of the Alphas is heavily loaded or is having problems. The particular machine you want to avoid can be commented out of the list of possible machines for selection. For example, subaru is not possible for selection below.

```
# sample machine file
honda.cis.udel.edu
toyota.cis.udel.edu
acura.cis.udel.edu
porsche.cis.udel.edu
# subaru.cis.udel.edu
mazda.cis.udel.edu
suzuki.cis.udel.edu
nissan.cis.udel.edu
hyundai.cis.udel.edu
```

For convenience, your `machinefile` should be kept in the same directory as your executable MPI files and named something appropriate like `machines`. The name of your `machinefile` will be used as an argument to the mpirun option `-machinefile`, (see next section).

2.5 Running MPI

In order to run an MPI compiled program, you must type:

```
mpirun -np <number of processors> [mpirun_options] <program name and arguments>
```

where you specify the number of processors on which you want to run your parallel program, the `mpirun` options, and your program name and its expected arguments.

Some examples of mpirun's:

```
mpirun -np 4 hello
mpirun -np 6 -machinefile machines hello
mpirun -np 5 integrate 0 1
```

Type `man mpirun` or `mpirun -help` for a complete list of `mpirun` options.

2.6 Checking and Killing Processes

Bugs? - Just don't write buggy programs! - Simple! Of course, it will clearly never happen that a program written in this class would ever have any sort of problems, but, if, for some reason, a program that you write were to crash unexpectedly, there's something to watch out for.

An MPI program that contains parallelism may start simultaneously on all (or at least, many) of the cluster machines. If one process crashes, and MPI dies, it is quite possible that some of the other processes

might continue living – and, cut off from their MPI connection – may just sort of hang around and use up CPU time. This is a great way to lose friends!

In fact, sit back for a while and imagine the machines, filled to the brim with students, all of them running their programs together on all the machines. One student's program crashes, leaving nine other copies of his program treading water.

Then a second person's program crashes. And a third.

These people try to fix their bugs, recompile, and run their programs again. The twenty-seven floundering processes from their first attempts are still around.

Some other people's programs crash, adding more dead weight. After a second compile-and-run attempt, the Alphas are host to sixty-three floundering processes, each potentially using up a unit of CPU load.

Inexplicably, the machines start to feel sluggish.

Slow, even.

Tempers flare. People start getting out their knives.

Not a good scene!

Soooo, for just such an eventuality, we have provided the commands in `pollock/372porsche06/public/bin`, namely `spy`, `spyall`, `alluptime`, `allusers`, `pings-all`, and `shoot`.

When you type `spy`, `spy` will start a remote shell on each of the machines and issue a `ps` command that will display the current status of all processes on the machines associated with your username. `spyall` will do the same, but show the status of all processes *owned by anyone* on the machines. `alluptime` tells you the time since the each machine was rebooted and load average. `allusers` tells you who is active on each machine. `pings-all` pings each machine for its status, also giving roundtrip times. `shoot` will insure that all your processes (except login shells on porsche) will die.

To access these programs, add these lines somewhere in your .cshrc/.tcshrc:

```
alias spy /usa/pollock/372porsche06/public/bin/spy
alias shoot /usa/pollock/372porsche06/public/bin/shoot
alias spyall /usa/pollock/372porsche06/public/bin/spyall
alias allusers  /usa/pollock/372porsche06/public/bin/allusers
alias alluptime /usa/pollock/372porsche06/public/bin/alluptime
alias pings-all /usa/pollock/372porsche06/public/bin/pings-all
```

Now when you type these commands, they will be found from your alias, and the link will point to my copy of the file which will be executed.

It is suggested that whenever you run an MPI program on a large portion of the cluster, and it crashes unexpectedly in a way that leads you to believe that there may be other, floundering processes left over, you should run `spy` to check out your suspicions and `shoot` to find and kill any processes you have hanging around.

It is **strongly suggested** that you issue a `shoot` command immediately before logging off porsche to help keep the peace. You should use `spyall` just before you run a program for performance numbers to be sure that no one else is running a job that will affect your performance numbers. You want to make sure that you are the only one using the cluster when you are doing performance runs.

2.7 Printing your Programs and Other Files

You can print directly from porsche onto any of the CIS Department printers, by typing `lpr -P<printer> <filename>` directly from porsche. However, the CIS Department printers are located in rooms typically not accessible by undergraduates, and the main department printer in the CIS Department office (103 Smith) is NOT to be used for printing coursework. So, you should print your files by copying your files to strauss and then printing from there to a printer on campus in which you have access. The best way to copy your files is via scp, or you can copy your files by ftp.

2.8 Copying Files via scp

Scp utilizes ssh to transmit your files, thus you will not be sending your password in clear text for the world to see (as you would with ftp). To copy files from porsche to strauss, type the following while on porsche,

```
scp <path1><file> <username>@strauss.udel.edu:<path2><file>
```

Where `<path1><file>` is the local machine path and filename and `<path2><file>` is the path and filename for the destination machine. When prompted for your password, enter it and press return. Your file will then be copied for you.

Due to security concerns, do not ftp files.

3 The Basics of Writing MPI Programs

You should now be able to successfully compile and execute MPI programs, check the status of your MPI processes, and halt MPI programs that have gone astray. This section gives an overview of the basics of parallel programming with MPI. For a more in-depth discussion of basic and advanced constructs, see the references or man pages on the web.

3.1 Fortran versus C versus C++ with MPI

MPI provides for programming in Fortran (mpif77), C (mpicc), and C++ (mpiCC). The MPI commands for C++ adhere to the C API. All names of MPI routines and constants in both C and Fortran begin with the prefix `MPI_` to avoid name collisions. For the remainder of this guide, only the C versions of the MPI routines will be presented. However, the primary differences between the C and Fortran routines are:

- Error codes are returned in a separate argument for Fortran as opposed to the return value for C functions.
- Fortran-compatible MPI routine names are totally uppercase (e.g., MPI_INIT), whereas C-compatible MPI routine names are upper and lowercase (e.g., MPI_Init).
- The arguments to most C-compatible MPI functions are more strongly typed than they are in Fortran, having specific C types such as MPI_Comm and MPI_Datatype where Fortran has integers.
- The *include* files are different: in C, `mpi.h`, in Fortan, `mpif.h`.
- The arguments to MPI_Init are different, so that a C program can take advantage of command-line arguments.

3.2 Initialization, Communicators, Handles, and Clean-Up

The first MPI routine called in any MPI program *must* be the initialization routine MPI_INIT. Every MPI program must call this routine *once*, before any other MPI routines. Making multiple calls to MPI_INIT is erroneous. The C version of the routine accepts the arguments to `main`, namely `argc` and `argv` as arguments.

MPI_INIT defines something called MPI_COMM_WORLD for each process that calls it. MPI_COMM_WORLD is a *communicator*. All MPI communication calls require a communicator argument and MPI processes can only communicate if they share a communicator.

Every communicator contains a *group* which is a list of processes. Secondly, a group is in fact *local* to a particular process. The group contained within a communicator has been previously agreed across the processes at the time when the communicator was set up. The processes are ordered and numbered consecutively from zero, the number of each process being known as its *rank*. The rank identifies each process within the communicator. The group of MPI_COMM_WORLD is the set of all MPI processes.

MPI maintains internal data structures related to communications etc. and these are referenced by the user through *handles*. Handles are returned to the user from some MPI calls and can be used in other MPI calls.

An MPI program should call the MPI routine MPI_FINALIZE when all communications have completed. This routine cleans up all MPI data structures etc. It does NOT cancel outstanding communications, so it is the responsibility of the programmer to make sure all communications have completed. Once this routine is called, no other calls can be made to MPI routines, not even MPI_INIT, so a process cannot later re-enroll in MPI.

3.3 MPI Indispensable Functions

This section contains the basic functions needed to manipulate processes running under MPI. It is said that MPI is small and large. What is meant is that the MPI standard has many functions in it, approximately 125. However, many of the advanced routines represent functionality that can be ignored until one pursues added flexibility (data types), robustness (nonblocking send/receive), efficiency ("ready mode"), modularity (groups, communicators), or convenience (collective operations, topologies). MPI is said to be small because there are six indispensable functions from which many useful and efficient programs can be written.

The six functions are:

```
MPI_Init - Initialize MPI
MPI_Comm_size - Find out how many processes there are
MPI_Comm_rank - Find out which process I am
MPI_Send - Send a message
MPI_Recv - Receive a message
MPI_Finalize - Terminate MPI
```

You can add functions to your working knowledge incrementally without having to learn everything at once. For example, you can accomplish a lot by just adding the collective communication functions MPI_Bcast and MPI_Reduce to your repertoire. These functions will be detailed below in addition to the six indispensable functions.

MPI_Init

The call to MPI_Init is required in every MPI program and must be the first MPI call. It establishes the MPI execution environment.

```
int MPI_Init(int *argc, char ***argv)

Input:
       argc - Pointer to the number of arguments
       argv - Pointer to the argument vector
```

MPI_Comm_size

This routine determines the size (i.e., number of processes) of the group associated with the communicator given as an argument.

```
int MPI_Comm_size(MPI_Comm comm, int *size)

Input:
       comm - communicator (handle)
Ouput:
       size - number of processes in the group of comm
```

MPI_Comm_rank

The routine determines the rank (i.e., which process number am I?) of the calling process in the communicator.

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)

Input:
       comm - communicator (handle)
Output:
       rank - rank of the calling process in the group of comm (integer)
```

MPI_Send

This routine performs a basic send; this routine may block until the message is received, depending on the specific implementation of MPI.

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)

Input:
      buf  - initial address of send buffer (choice)
   count - number of elements in send buffer (nonnegative integer)
   datatype - datatype of each send buffer element (handle)
      dest - rank of destination (integer)
      tag  - message tag (integer)
      comm - communicator (handle)
```

MPI_Recv

This routine performs a basic receive.

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Status *status)

Output:
      buf  - initial address of receive buffer
   status - status object, provides information about message received;
          status is a structure of type MPI_Status, the element
          status.MPI_SOURCE is the source of the message received,
          and the element status.MPI_TAG is the tag value.

Input:
   count - maximum number of elements in receive buffer (integer)
   datatype - datatype of each receive buffer element (handle)
   source - rank of source (integer)
   tag  - message tag (integer)
   comm - communicator (handle)
```

MPI_Finalize

This routine terminates the MPI execution environment; all processes must call this routine before exiting.

```
int MPI_Finalize(void)
```

MPI_Bcast

This routine broadcasts data from the process with rank "root" to all other processes of the group.

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root,
              MPI_Comm comm)

Input/Output:
   buffer - starting address of buffer (choice)
   count - number of entries in buffer (integer)
   datatype - data type of buffer (handle)
   root - rank of broadcast root (integer)
      comm - communicator (handle)
```

MPI_Reduce

This routine combines values on all processes into a single value using the operation defined by the parameter op.

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype
               datatype, MPI_Op op, int root, MPI_Comm comm)
```

```
Input:
   sendbuf - address of send buffer (choice)
   count - number of elements in send buffer (integer)
   datatype - data type of elements of send buffer (handle)
   op - reduce operation (handle) (user can create using MPI_Op_create
          or use predefined operations MPI_MAX, MPI_MIN, MPI_PROD, MPI_SUM,
          MPI_LAND, MPI_LOR, MPI_LXOR, MPI_BAND, MPI_BOR, MPI_BXOR,
          MPI_MAXLOC, MPI_MINLOC in place of MPI_Op op.
   root - rank of root process (integer)
   comm - communicator (handle)

Output:
   recvbuf - address of receive buffer (choice, significant only at root )
```

3.4 A Simple MPI Program - hello.c

Consider this demo program:

```
/*The Parallel Hello World Program*/
#include <stdio.h>
#include <mpi.h>

main(int argc, char **argv)
{
   int node;

   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &node);

   printf("Hello World from Node %d\n",node);

   MPI_Finalize();
}
```

In a nutshell, this program sets up a communication group of processes, where each process gets its rank, prints it, and exits. It is important for you to understand that in MPI, this program will start simultaneously on all machines. For example, if we had ten machines, then *running this program* would mean that ten separate instances of this program would start running together on ten different machines. This is a fundamental difference from ordinary C programs, where, when someone said "run the program", it was assumed that there was only one instance of the program running.

The first line,

```
#include <stdio>
```

should be familiar to all C programmers. It includes the standard input/output routines like `printf`. The second line,

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

includes the MPI functions. The file `mpi.h` contains prototypes for all the MPI routines in this program; this file is located in /usr//include/mpi/mpi.h in case you actually want to look at it.

The program starts with the `main...` line which takes the usual two arguments `argc` and `argv`, and the program declares one integer variable, `node`. The first step of the program,

```
MPI_Init(&argc,&argv);
```

calls MPI_Init to initialize the MPI environment, and generally set up everything. This should be the first command executed in all programs. This routine takes pointers to argc and argv, looks at them, pulls out the purely MPI-relevant things, and generally fixes them so you can use command line arguments as normal.

Next, the program runs MPI_Comm_rank, passing it an address to `node`.

```
MPI_Comm_rank(MPI_COMM_WORLD, &node);
```

MPI_Comm_rank will set `node` to the rank of the machine on which the program is running. Remember that in reality, several instances of this program start up on several different machines when this program is run. These processes will each receive a unique number from MPI_Comm_rank.

Because the program is running on multiple machines, each will execute not only all of the commands thus far explained, but also the *hello world* message printf, which includes their own rank.

```
printf("Hello World from Node %d\n",node);
```

If the program is run on ten computers, `printf` is called ten times on ten different machines simultaneously. The order in which each process executes the message is undetermined, based on when they each reach that point in their execution of the program, and how they travel on the network. Your guess is as good as mine. So, the ten messages will get dumped to your screen in some undetermined order, such as:

```
Hello World from Node 2
Hello World from Node 0
Hello World from Node 4
Hello World from Node 9
Hello World from Node 3
Hello World from Node 8
Hello World from Node 7
Hello World from Node 1
Hello World from Node 6
Hello World from Node 5
```

Note that all the printf's, though they come from different machines, will send their output intact to your shell window; this is generally true of output commands. Input commands, like scanf, will only work on the process with rank zero. After doing everything else, the program calls MPI_Finalize, which generally terminates everything and shuts down MPI. This should be the last command executed in all programs.

4 MPE Graphics

4.1 Overview

In addition to the MPI functions, there is also a set of graphics routines located in a library called MPE. These routines are useful for MPI parallel programs that involve displaying graphical images. This library includes a simple interface to X. To execute MPI programs that display graphics to your screen (e.g., the Mandelbrot Renderer), you will need to have both `#include "mpe.h"` and `#include "/usr/lib/mpich/include/mpe_graphics.h"` in your program files. Then, you need load the mpe library on the compilation line (see below). See the mandelbrot program in `pollock/372porsche/public/mandel` directory for an example.

You will also have to compile any program that uses MPE by including the `-lmpe` as part of the compilation line, like this:

```
mpicc lab.c -o lab -lmpe -lX11 -lm  -L/usr/local/X11R6/lib
```

Unfortunately, MPE does not supply the proper functions to create even a semi-reasonable interface. Thus, MPE_Get_mouse_status and MPE_Drag_square are provided for your use. These functions are in the file `~pollock/372porsche06/public/mandel/mouse_status.c` on porsche, with prototypes in the file

~pollock/372porsche06/public/mandel/mouse_status.h on porsche. You should copy these files into your own directory.

Below is a list of the most frequently used MPE routines; the list does not contain all of the MPE graphics routines. For more, look at the file: /usr/lib/mpich/include/mpe_graphics.h. Both this file and the mpe.h files need to be included in programs using the graphics library. (See the compilation section.)

```
MPE_Open_graphics  - create a graphics window
MPE_Close_graphics  - destroy a graphics window
MPE_Draw_point  - draw a point in a window
MPE_Draw_points - draw a series of points in a window.
   (moderately faster than a series of MPI_Draw_point calls)
MPE_Draw_line  - draw a line in a window
MPE_Fill_rectangle  - draw a rectangle in a window
MPE_Update  - flush the buffer for a window
MPE_Get_mouse_press - wait until the user presses a mouse button
   and return the press point.
MPE_Get_mouse_status (in mouse_status.c) - get information about the mouse state
MPE_Drag_square (in mouse_status.c)  - let the user select a square on the screen
MPE_Make_color_array - create a nice spectrum of colors
```

4.2 Details on some MPE Graphics Routines

```
MPE_Open_graphics (MPE_XGraph *window, MPI_Comm comm, char *display, int x,int y,
int width, int height, int is_collective);
```

Open a window at x, y of size width, height. If you pass -1 for x and y, the user will be required to position the window. If NULL is passed for display, then the display will be configured automatically. Always pass 0 for is_collective. MPE_Open_graphics must be called on all nodes in comm. Don't forget to pass the address of your window!

```
MPE_Close_graphics (MPE_XGraph *window);
```

Close the window associated with window. All processes must call this routine. Once any process has called MPE_Close_graphics, no process can call any other MPE routine. Don't forget to pass the address of your window!

```
MPE_Draw_point (MPE_XGraph window, int x, int y, MPE_Color color);
```

Draw a pixel at (x, y). Initially, MPE_Color can be one of: MPE_WHITE, MPE_BLACK, MPE_RED, MPE_YELLOW, MPE_GREEN, MPE_CYAN, and MPE_BLUE. You may change the available colors using MPE_Make_color_array and MPE_Add_RGB_color (see these routines' man pages). Note that the point may not actually be drawn until you call MPE_Update.

```
MPE_Draw_points (MPE_XGraph window, const MPE_Point *points, int npoints);
```

Draws a series of points at once. points should point to an array of MPE_Point structures. Here's the form of an MPE_Point:

```
typedef struct {
  int x, y;
  MPE_Color c;
} MPE_Point;
```

npoints should contain the number of points that are in the array pointed to by points. Note that the points may not actually be drawn until you call MPE_Update.

```
MPE_Draw_line (MPE_XGraph window, int x1, int y1, int x2, int y2, MPE_Color
color);
```

Draw a line from `(x1, y1)` to `(x2, y2)`.

```
MPE_Fill_rectangle (MPE_XGraph window, int x, int y, int width, int height,
MPE_Color color);
```

Fill a rectangle with upper-left corner at `(x, y)` of size `width, height`, in pixels.

```
MPE_Update (MPE_XGraph window);
```

The MPE graphics library buffers the drawing commands that you execute, so that they can be sent to the X server all at once. `MPE_Update` sends the contents of the buffer. You should call `MPE_Update` whenever your process may be idle for a while (so that the window is not partially drawn).

```
MPE_Get_mouse_press (MPE_XGraph window, int *x, int *y, int *button);
```

Blocks until a mouse button is pressed in `window`. Then, the mouse position (relative to the upper-left corner of the window) is returned in `x` and `y`. The number of the button that was pressed is returned in `button`.

```
MPE_Get_mouse_status (MPE_XGraph window, int *x, int *y, int *button, int
*wasPressed);
/* in mouse_status.h. */
```

Does exactly the same thing as `MPE_Get_mouse_press`, buy returns immediately even if no button is pressed. `wasPressed` will be non-zero if any button was pressed at the time of the call.

```
MPE_Drag_square (MPE_XGraph window, int *startx, int *starty, int *endx,
int *endy);
/* In mouse_status.h. */
```

Wait for the user to drag out a square on the screen. It is OK if a button is already pressed when you call `MPE_Drag_square`; for instance, you might call `MPE_Get_mouse_press` to wait for a mouse press, and then call `MPE_Drag_square` only if a certain button was pressed. If the button is already pressed when you call `MPE_Drag_square`, `*startx` and `*starty` should contain the point at which the mouse was pressed. `*endx` and `*endy` will always be greater than `*startx` and `*starty`, even if the user drags the square from right to left.

```
MPE_Make_color_array (MPE_XGraph window, int ncolors, MPE_Color *colors);
```

This function creates a nice rainbow spectrum of `ncolors` colors. It places these colors into the array pointed to by `colors`; this array should have at least `ncolors` elements. If not enough colors are available, then some of the returned colors will be random. Mosaic and Netscape tend to hog the colormap, so you might want to quit them before running your program to get the correct colors. The maximum value for `ncolors` is 254. The new colors replace all the standard MPE colors except `MPE_BLACK` and `MPE_WHITE`. You should call `MPE_Make_color_array` from all the nodes that you plan to draw from.

Remember that most of these functions also have man pages.

5 Gathering Performance Data

5.1 Timing Programs

For timing parallel programs, MPI includes the routine `MPI_Wtime()` which returns elapsed wall clock time in seconds. The timer has no defined starting point, so in order to time something, two calls are needed and the difference should be taken between the returned times. As a simple example, we can time each of the processes in the hello world program as below:

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

/*NOTE: The MPI_Wtime calls can be placed anywhere between the MPI_Init
and MPI_Finalize calls.*/

main(int argc, char **argv)
{
   int node;
   double mytime;   /*declare a variable to hold the time returned*/

   MPI_Init(&argc,&argv);
   mytime = MPI_Wtime();  /*get the time just before work to be timed*/
   MPI_Comm_rank(MPI_COMM_WORLD, &node);

   printf("Hello World from Node %d\n",node);

   mytime = MPI_Wtime() - mytime; /*get the time just after work is done
                                    and take the difference */
   printf("Timing from node %d is %lf seconds.\n",node,mytime);
   MPI_Finalize();

 }
```

It might be nice to know what was the least/most execution time spent by any individual process as well as the average time spent by all of the processes. This will give you a vague idea of the distribution of work among the processes. (A better idea can be gained by calculating the standard deviation of the run times.) To do this, in additional to a few calls to get the value of the system clock, you need to add a call to synchronize the processes and a few more calls to collect the results. For example, to time a function called `work()` which is executed by all of the processes, one would do the following:

```
    int myrank,
        numprocs;
    double mytime,   /*variables used for gathering timing statistics*/
           maxtime,
           mintime,
           avgtime;

    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Barrier(MPI_COMM_WORLD);  /*synchronize all processes*/
    mytime = MPI_Wtime();  /*get time just before work section */
    work();
    mytime = MPI_Wtime() - mytime;  /*get time just after work section*/
/*compute max, min, and average timing statistics*/
    MPI_Reduce(&mytime, &maxtime, 1, MPI_DOUBLE,MPI_MAX, 0, MPI_COMM_WORLD);
    MPI_Reduce(&mytime, &mintime, 1, MPI_DOUBLE, MPI_MIN, 0,MPI_COMM_WORLD);
    MPI_Reduce(&mytime, &avgtime, 1, MPI_DOUBLE, MPI_SUM, 0,MPI_COMM_WORLD);
    if (myrank == 0) {
      avgtime /= numprocs;
      printf("Min: %lf  Max: %lf  Avg:  %lf\n", mintime, maxtime,avgtime);
    }
```

Be sure to execute `spyall` to ensure that no one else is using the cluster at the time you want to run a performance execution. Other processes in the system will affect your performance timings. On a network of workstations, the reported times may be off by a second or two due to network latencies as well as interference from the operating system and the jobs of other users. On a tightly coupled machine like the Paragon, these timings should be accurate to within a second. Thus, it's best in general to time things that run for long enough that the system noise isn't significant, and time them several times. If you get an anomalous timing, don't hesitate to run the code a few more times to see if it can be reproduced.

If you are comparing the execution times of a sequential program with a parallel program written in MPI, be sure to use the mpicc compiler for both programs with the same switches, to ensure that the same optimizations are performed. Then, run the sequential version via mpirun -np 1.

5.2 Profiling and Viewing Profile Information

[Note that while this section follows the documentation for MPI and MPE, the logging MPE_Finish_log seems to be improperly working on the linux cluster as of 9/8/03. Stay tuned for an update. Also, upshot.mpich will provide a visualization, but also a floating point error on the linux machines.]

Although timing can provide insight into the performance of a program, it is sometimes desirable to see in detail the sequence of communication and computational events that occurred in a program and the amount of time spent in each phase. This information is usually gained by tracing various events during execution, ie., logging information as the parallel program runs. Files that contain time-stamped communication and computational events are called **logfiles**. The easiest way to understand this data at a glance is with a graphical tool. In the next two subsections, creation of a logfile using MPE logging routines and viewing of the logfile using the program **upshot** are described.

5.2.1 MPE Logging Routines

To log events in a program, you need to have `#include "mpe.h"` and `#include "mpe_log.h"` in your program file. The logging routines in MPE are used to create logfiles of events that occur during the execution of a parallel program. These files can be studied after the program has ended successfully. The following routines allow the user to log events that are meaningful for specific applications rather than relying on automatic logging of MPI library calls. The basic routines are MPE_Init_log, MPE_Log_event, and MPE_Finish_log.

MPE_Init_log must be called by all processes to initialize MPE logging data structures. MPE_Finish_log collects the log data from all the processes, merges it, and aligns the timestamps with respect to the times at which MPE_Init_log and MPE_Finish_log were called. Then, the process with rank 0 in MPI_Comm_world writes the log into the file whose name is given as an argument to MPE_Finish_log.

A single event is logged with the MPE_Log_event routine. The routines MPE_Describe_event and MPE_Describe_state allow one to add event and state descriptions and to define states by specifying a starting and ending event for each state. Finally, MPE_Start_log and MPE_Stop_log can be used to dynamically turn logging on and off, respectively. By default, logging is on after MPE_Init_log is called. For the specific syntax of these routines, you can also consult the man pages, e.g., `man MPE_Describe_state`. The following sample program demonstrates some of these logging routines. The program can be found on porsche in

`~pollock/372porsche06/public/cpilog.c`

```
/* Sample Program with Logging Commands*/
#include "mpi.h"
#include "mpe.h"
#include <math.h>
#include <stdio.h>

double f(a)
double a;
{
```

```
    return (4.0 / (1.0 + a*a));
}

int main(argc,argv)
int argc;
char *argv[];
{
  int done = 0, n, myid, numprocs, i, rc, repeat;
  double PI25DT = 3.14159265358979323846264 3;
  double mypi, pi, h, sum, x, a;
  double startwtime, endwtime;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);

  MPE_Init_log();
  if (myid == 0) {
    MPE_Describe_state(1, 2, "Broadcast", "red:vlines3");
    MPE_Describe_state(3, 4, "Compute",   "blue:gray3");
    MPE_Describe_state(5, 6, "Reduce",    "green:light_gray");
    MPE_Describe_state(7, 8, "Sync",      "yellow:gray");
  }

  while (!done)
    {
      if (myid == 0)
{
  printf("Enter the number of intervals: (0 quits) ");
  scanf("%d",&n);
  startwtime = MPI_Wtime();
}
      MPI_Barrier(MPI_COMM_WORLD);
      MPE_Start_log();

      MPE_Log_event(1, 0, "start broadcast");
      MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
      MPE_Log_event(2, 0, "end broadcast");

      if (n == 0)
done = 1;
      else
{
  for (repeat=5; repeat; repeat--) {
    MPE_Log_event(7,0,"Start Sync");
    MPI_Barrier(MPI_COMM_WORLD);
    MPE_Log_event(8,0,"End Sync");
    MPE_Log_event(3, 0, "start compute");
    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs)
      {
x = h * ((double)i - 0.5);
sum += f(x);
```

```
      }
    mypi = h * sum;
    MPE_Log_event(4, 0, "end compute");
    fprintf( stderr, "[%d] mypi = %lf\n", myid, mypi );

    MPE_Log_event(5, 0, "start reduce");
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    MPE_Log_event(6, 0, "end reduce");
    if (myid == 0)
      {
printf("pi is approximately %.16f, Error is %.16f\n",
       pi, fabs(pi - PI25DT));
endwtime = MPI_Wtime();
printf("wall clock time = %f\n", endwtime-startwtime);
      }
  }
        }
      MPE_Stop_log();
    }
  MPE_Finish_log("cpilog.log");
  MPI_Finalize();
}
```

5.2.2 Profile Visualization with Upshot

The logfile viewing program that is distributed with MPI is called `upshot`. It is a simple graphical display of parallel time lines and state durations. Upshot is a Tcl/Tk script, so it can be customized and extended. Once you have created a logfile by inserting the MPE logging routines into your program and compiling and executing the program, the logfile can be viewed by invoking upshot. Simply type

```
upshot.mpich filename.log
```

at the UNIX prompt on porsche. `upshot.mpich` is found in /usr/bin. Try this on the sample logfile cpilog.log provided for you on porsche in

```
~pollock/372porsche06/public/cpilog.log
```

When the window titled Upshot appears, click on `Setup`. `cpilog.log` will be automatically read, and upshot will display parallel time lines for each process, with states indicated by colored bars. Timestamp values, adjusted to start at 0 are shown along the bottom of the time lines. Upshot provides zooming capability for magnified views of the time lines.

6 Debugging MPI Programs

Parallel programs are much more difficult to debug than their serial counterparts. Not only do you have to worry about the things you worry about with serial programs, such as syntax and logic, but you also have to worry about parallel programming problems such as deadlock, nondeterminism, and synchronization.

6.1 Debugging Methods

The following method is suggested for debugging MPI programs. First, if possible, write the program as a serial program. This will allow you to debug most syntax, logic, and indexing errors.

Then, modify the program and run it with 2–4 processes on the same machine. This step will allow you to catch syntax and logic errors concerning intertask communication. A common error found at this point is the use of non-unique message tags. The final step in debugging your application is to run the same

processes on different machines. This will check for synchronization errors caused by message delays on the network.

You should first try to find the bug by using a few `printf` statements. If this does not work, then you may want to try running the program under a debugger such as dbx or gdb which will start the first process under the debugger where possible. Note that the debuggers are not parallel versions, but are only being used by the first process. Also, their interaction with parallel programs was not tested, therefore use at your own risk. You may be able to find some parallel debuggers that are stable, by searching the MPI web pages. Please let us know if you find one that seems to be helpful and stable.

6.2 Common Problems: Descriptions and Tips

This section contains descriptions and fixes for some common problems.

6.2.1 Lost Output

If some or all of your output does not appear, it is the result of one of two things (or a combination of both). Sometimes the output will have disappeared without a trace. The reason for this is that under UNIX, output is placed in a buffer to be printed, but not actually printed yet, to increase the efficiency of the system. Sometimes MPI marks a process as dead when it exits, and therefore the output in the buffer is never read. This is usually the case when some output appears, but not all of it. To correct this, add the statement:

```
fflush(stdout);
```

after each `printf` statement. This will flush the buffer after each `printf` so no output will be in the buffer when a process exits MPI.

If you attempt to write output to a single file from multiple processes, the file will be overwritten; there is no append capability. The file will contain only the output from the last process that writes to it.

Writing to standard output does appear to work okay, except that the information may not appear in the order your program would suggest. Again, adding the following:

```
fflush(stdout);
sleep(1);
```

after every `printf` statement will often force the output to arrive in the proper order, when the real problem is the network latency, and not nondeterminism in your program.

6.2.2 Error Messages

Many messages - When an MPI job crashes, you typically get more than one line of error messages. The FIRST line is the most important and contains the clue to your actual problem. The rest of the messages are usually the system's attempt to clean up the rest of the processes that have been left hanging!

Infinite messages - Occasionally, you will get runaway error messages that appear to be in an infinite loop... You will need to log onto porsche from another window and issue a `shoot` command.

Intermittent messages - The cluster may sometimes issue intermittent error messages on programs that are correct. The errors may have something to do with a network or hardware problem. Gurus are looking at the problem but... unfortunately you may still have to deal with this! Our suggestion... build your program slowly, adding just a few lines at a time. If you do get an error that does not have obvious origins, run the code a couple of times to make sure it is your problem and not the system. Don't forget to do a `shoot` command between runs to clean up leftover jobs. Error messages that are suspicious for being system problems usually contain phrases like:

```
net_send: could not write
unidentified err handler
bad file number
interrupt SIGBUS: 10
```

Uninitialized variables - Another potential problem error could be uninitialized variables. MPI_Init in the main part of your program appears to set uninitialized variables to zero; however, uninitialized variables in subroutines appear to be set to the usual C compiler initialization; that is, garbage. Beware of subroutines bearing garbage! A clue to this problem is a SIGFPE error message.

A reminder of common signals and their explanation:

```
SIGABRT - Abnormal termination of the program (such as a call to abort).
SIGFPE  - An erroneous arithmetic operation, such as a divide-by-zero
                or an operation resulting in overflow
SIGILL -  Detection of an illegal instruction
SIGINT - Receipt of an interactive attention signal
SIGSEGV - An invalid access to storage
SIGTERM - A termination request sent to the program
```

7 World Wide Web Resources

There are some valuable resources online regarding MPI. A good place to start is the Argonne National Labs site:

`http://www.mcs.anl.gov/Projects/mpi/`

8 References

1. Gropp, Lusk, and Skjellum *Using MPI- Portable Parallel Programming with the Message-Passing Interface* The MIT Press: Cambridge, Mass, 1994.

2. Snir, et al. *MPI: The Complete Reference* The MIT Press: Cambridge, Mass, 1994.

3. Kerninghan and Ritchie *The C Programming Language, 2nd Edition* Prentice Hall: Englewood Cliffs, 1988.