ANL/MCS-TM-ANL-98/xx

User's Guide for `MPE`: Extensions for MPI Programs

by

Anthony Chan, William Gropp, and Ewing Lusk

ARGONNE NATIONAL LABORATORY • UNIVERSITY OF CHICAGO •

MATHEMATICS AND
COMPUTER SCIENCE
DIVISION

Contents

Abstract

The MPE extensions provide a number of useful facilites for MPI programmers. These include several profiling libraries to collect information on MPI programs, including logfiles for post-mortum visualization and real-time animation. Also included are routines to provide simple X window system graphics to parallel programs. MPE may be used with any implemenation of MPI.

1 Introduction

The Message Passing Interface (MPI) [4] provides a strong basis for building parallel programs. One of its design goals was to enable the construction of parallel software libraries, thus helping to solve the problem of developing large parallel applications. The MPE (Multi-Processing Environment) library exploits the features of MPI to provide a number of useful facilities, including performance and correctness debugging, graphics, and some common utility routines.

The MPE library was developed for the `MPICH` [1] implementation of MPI (and is included with the `MPICH` distribution), but can and has been used with any MPI implementation. Installation instructions for MPE are in Appendix A.

2 The MPE library of useful extensions

Currently the main components of the `MPE` are

- A set of routines for creating logfiles for examination by various graphical visualization tools : `upshot`, `nupshot`, `Jumpshot-2` or `Jumpshot-3`.
- A shared-display parallel X graphics library.
- Routines for sequentializing a section of code being executed in parallel.
- Debugger setup routines.

2.1 Logfile Creation

MPE provides several ways to generate logfiles that describe the progress of a computation. These logfiles can be viewed with one of the graphical tools distributed with MPE. In addition, you can customize these logfiles to add application-specific information.

The easiest way to generate logfiles is to link your program with a special MPE library that uses the profiling feature of MPI to intercept all MPI calls in an application.

You can create customized logfiles for viewing by calls to the various `MPE` logging routines. For details, see the `MPE` man pages. An example is shown in Section 2.5.3.

2.2 Logfile Formats

MPE currently provides three different logfile formats: ALOG, CLOG and SLOG. ALOG is provided for backward compatibility purposes only, and stores events as ASCII text. CLOG is similar to ALOG, but stores data in a binary format (essentially the same as "external32" in MPI-IO or as the format specified for Java). SLOG is an abbreviation for Scalable LOGfile format and stores data as states (essentially an event with a duration) in a special binary format chosen to help visualization programs handle very large (multi-Gigabyte) log files.

Each of these log file formats has one or more visualization programs associated with it. The ALOG format is understood by `nupshot`. The CLOG format is understood by `nupshot` and `jumpshot`, a Java-based visualization tool. SLOG and `Jumpshot-3` are capable of handling logfiles containing gigabytes of data [5].

2.3 Parallel X Graphics

MPE provides a set of routines that allows you to display simple graphics with the X Window System. In addition, there are routines for input, such as getting a region defined by using the mouse. A sample of the available graphics routines are shown in Table 1. For arguments, see the `man` pages.

	Control Routines
`MPE_Open_graphics`	(collectively) opens an X display
`MPE_Close_graphics`	Closes a X11 graphics device
`MPE_Update`	Updates an X11 display
	Output Routines
`MPE_Draw_point`	Draws a point on an X display
`MPE_Draw_points`	Draws points on an X display
`MPE_Draw_line`	Draws a line on an X11 display
`MPE_Draw_circle`	Draws a circle
`MPE_Fill_rectangle`	Draws a filled rectangle on an X11 display
`MPE_Draw_logic`	Sets logical operation for new pixels
`MPE_Line_thickness`	Sets thickness of lines
`MPE_Make_color_array`	Makes an array of color indices
`MPE_Num_colors`	Gets the number of available colors
`MPE_Add_RGB_color`	Add a new color
	Input Routines
`MPE_Get_mouse_press`	Get current coordinates of the mouse
`MPE_Get_drag_region`	Get a rectangular region

Table 1: MPE graphics routines.

You can find an example of the use of the `MPE` graphics library in the directory `mpich/mpe/contrib/mandel`. Enter

```
make
mpirun -np 4 pmandel
```

to see a parallel Mandelbrot calculation algorithm that exploits several features of the MPE graphics library.

2.4 Other MPE Routines

Sometimes during the execution of a parallel program, you need to ensure that only a few (often just one) processor at a time is doing something. The routines MPE_Seq_begin and MPE_Seq_end allow you to create a "sequential section" in a parallel program.

The MPI standard makes it easy for users to define the routine to be called when an error is detected by MPI. Often, what you'd like to happen is to have the program start a debugger so that you can diagnose the problem immediately. In some environments, the error handler in MPE_Errors_call_dbx_in_xterm allows you to do just that. In addition, you can compile the MPE library with debugging code included. (See the -mpedbg configure option.)

2.5 Profiling Libraries

The MPI profiling interface provides a convenient way for you to add performance analysis tools to any MPI implementation. We demonstrate this mechanism in MPICH, and give you a running start, by supplying three profiling libraries with the MPICH distribution. MPE users may build and use these libraries with any MPI implementation.

2.5.1 Accumulation of Time Spent in MPI routines

The first profiling library is simple. The profiling version of each MPI_Xxx routine calls PMPI_Wtime (which delivers a time stamp) before and after each call to the corresponding PMPI_Xxx routine. Times are accumulated in each process and written out, one file per process, in the profiling version of MPI_Finalize. The files are then available for use in either a global or process-by-process report. This version does not take into account nested calls, which occur when MPI_Bcast, for instance, is implemented in terms of MPI_Send and MPI_Recv. The file 'mpe/src/trc_wrappers.c' implements this interface, and the option -mpitrace to any of the compilation scripts (e.g., mpicc) will automatically include this library.

2.5.2 Automatic Logging

The second profiling library is called MPE *logging* libraries which generate logfiles, they are files of timestamped events for CLOG and timestamped states for SLOG. During execution, calls to MPE_Log_event are made to store events of certain types in memory, and these memory buffers are collected and merged in parallel during MPI_Finalize. During execution, MPI_Pcontrol can be used to suspend and restart logging operations. (By default, logging is on. Invoking MPI_Pcontrol(0) turns logging off; MPI_Pcontrol(1) turns it back on again.) The calls to MPE_Log_event are made automatically for each MPI call.

You can analyze the logfile produced at the end with a variety of tools; these are described in Sections 2.6.1 and 2.6.2.

2.5.3 Customized Logging

In addition to using the predefined MPE *logging* libraries to log all MPI calls, MPE logging calls can be inserted into user's MPI program to define and log states. These states are called *user defined* states. States may be nested, allowing one to define a state describing a user routine that contains several MPI calls, and display both the user-defined state and the MPI operations contained within it. The routine `MPE_Log_get_event_number` should be used to get unique event numbers[1] from the MPE system. The routines `MPE_Describe_state` and `MPE_Log_event` are then used to describe user-defined states. The following example illustrates the use of these routines.

```
int eventID_begin, eventID_end;
...
eventID_begin = MPE_Log_get_event_number();
eventID_end   = MPE_Log_get_event_number();
...
MPE_Describe_state( eventID_begin, eventID_end, "Amult", "bluegreen" );
...
MyAmult( Matrix m, Vector v )
{
    /* Log the start event along with the size of the matrix */
    MPE_Log_event( eventID_begin, m->n, (char *)0 );
    ... Amult code, including MPI calls ...
    MPE_Log_event( eventID_end, 0, (char *)0 );
}
```

The logfile generated by this code will have the MPI routines within the routine `MyAmult` indicated by a containing bluegreen rectangle. The color used in the code is chosen from the file, '`rgb.txt`', provided by X server installation, e.g. '`rgb.txt`' is located in '`/usr/X11R6/lib/X11`' on Linux.

If the MPE logging library, '`liblmpe.a`', is not linked with the user program, `MPE_Init_log` must be called before all other MPE calls and `MPE_Finish_log` must be called after all the MPE calls. The sample programs '`cpilog.c`' and '`fpi.f`', available in MPE source directory '`contrib/test`' or the installed directory '`share/examples`', illustrate the use of these MPE routines.

2.5.4 Real-Time Animation

The third library does a simple form of real-time program animation. The MPE graphics library contains routines that allow a set of processes to share an X display that is not associated with any one specific process. Our prototype uses this capability to draw arrows

[1] This is important if you are writing a library that uses the MPE logging routines.

that represent message traffic as the program runs. Note that MPI programs can generate communication events far faster than most X11 servers can display the arrows that represent those events.

2.6 Logfile Viewers

There are 4 graphical visualization tools distributed with MPE, they are `upshot`, `nupshot`, `Jumpshot-2` and `Jumpshot-3`. Out of these 4 Logfile Viewers, only 3 viewers are built by MPE. They are `upshot`, `Jumpshot-2` and `Jumpshot-3`.

2.6.1 Upshot and Nupshot

One tool that we use is called `upshot`, which is a derivative of Upshot [3], written in Tcl/Tk (version 4). A screen dump of Upshot in use is shown in Figure 1. It shows parallel time

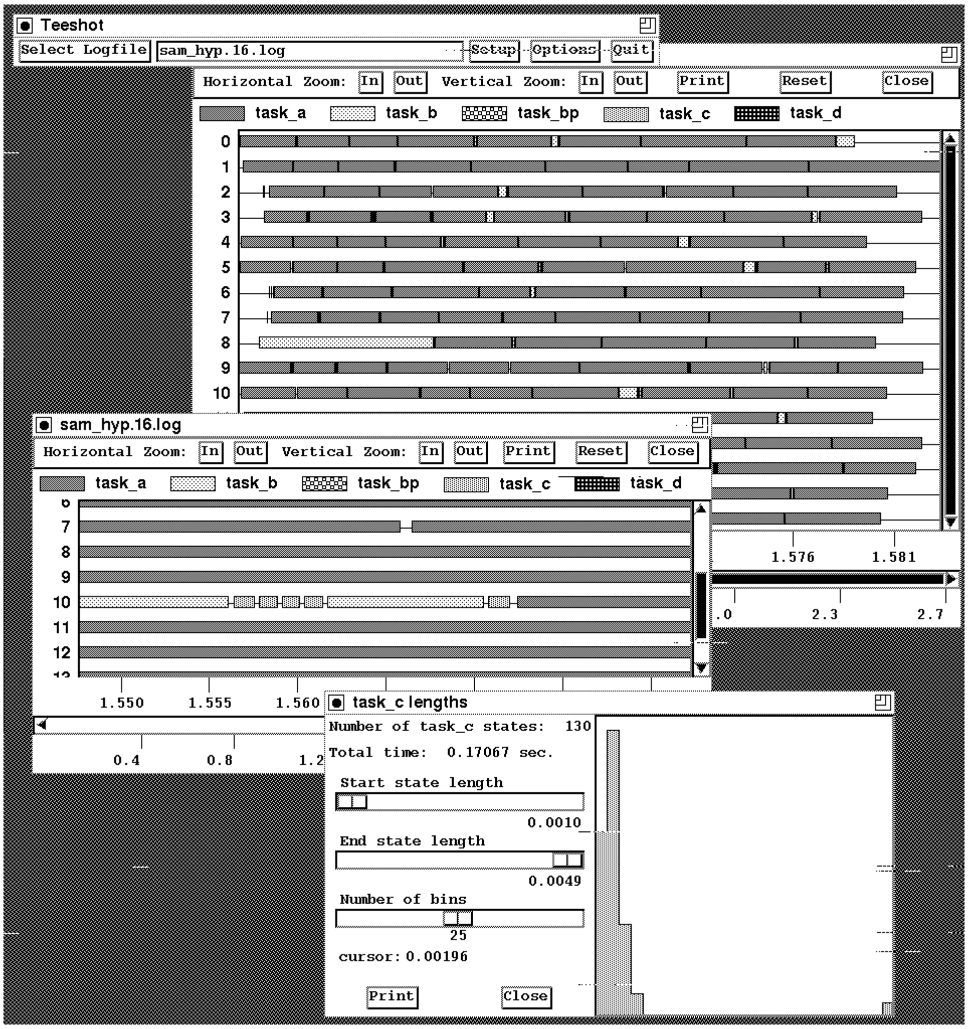

Figure 1: A screendump from `upshot`

lines with process states, like one of the paraGraph [2]. The view can be zoomed in or out, horizontally or vertically, centered on any point in the display chosen with the mouse. In Figure 1, the middle window has resulted from zooming in on the upper window at a chosen point to show more detail. The window at the bottom of the screen show a histogram of state durations, with several adjustable parameters.

`Nupshot` is a version of `upshot` that is faster but requires an older version of Tcl/Tk (version 3, to be precise). Because of this limitation, `Nupshot` is not built by default in current MPE.

2.6.2 Jumpshot-2 and Jumpshot-3

There are 2 versions of Jumpshot distributed with the MPE. They are `Jumpshot-2` and `Jumpshot-3`, which have evolved from Upshot and Nupshot. Both are written in Java and are graphical visualization tools for interpreting binary tracefiles which displays them onto the display, as shown in Figure 2. For `Jumpshot-2`, see [6] for more screenshots and details. For `Jumpshot-3`, See file '`mpe/viewers/jumpshot-3/doc/TourStepByStep.pdf`' for a brief introduction of the tool.

As the size of the logfile increases, `Jumpshot-2`'s performance decreases, and can ultimately result in `Jumpshot-2` hanging while it is reading in the logfile. It is hard to determine at what point `Jumpshot-2` will hang, but we have seen it with files as small as 10MB. When CLOG file is about 4MB in size, the performance of `Jumpshot-2` starts to deterioate significantly. There is a current research effort that will result in the ability to make the Java based display program significantly more scalable. The results of the first iteration of this effort are SLOG which supports scalable logging of data and `Jumpshot-3` which reads SLOG.

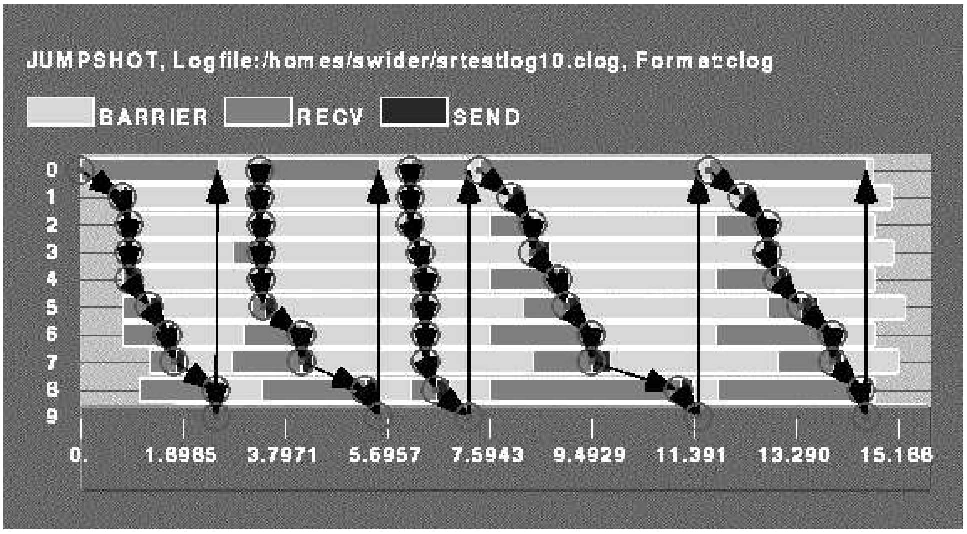

Figure 2: Jumpshot-1 Display

2.7 Accessing the profiling libraries

If the MPE libraries have been built, it is very easy to access the profiling libraries. The configure in the `MPE` directory determines the link path necessary for each profiling library (which varies slightly for each MPI implementation). These variables are first substituted in the Makefile in the directory '`mpe/contrib/test`'. The Makefile is then installed into directory '`share/examples`' during the final installation process. This information is placed in the following variables:

- **PROF_LIBS** - The compiler flag needed to link with the mpe library only. The link path is `-lmpe -lpmpich` or `-lmpe -lpmpi` depending on the MPI implementation.
- **LOG_LIBS** - The compiler flag needed to link with the logging libraries. The logging libraries log all MPI calls and generate log file. The link path is `-llmpe $PROF_LIB`.
- **TRACE_LIBS** - The compiler flag needed to link with the tracing library. The tracing library will trace all MPI calls. Each MPI call is preceded by a line that contains the rank in `MPI_COMM_WORLD` of the calling process, and followed by another line indicating that the call has completed. Most send and receive routines also indicate the values of count, tag, and partner (destination for sends, source for receives). Output is to standard output. The link path is `-ltmpe $PROF_LIB`.
- **ANIM_LIBS** - The compiler flag needed to link with the animation library. The animation library produces a real-time animation of the program. This requires the MPE graphics, and uses X11 Window System operations. You may need to provide a specific path for the X11 libraries (instead of `-lX11`). The link path is `-lampe $PROF_LIB -lX11`.
- **F2CMPI_LIBS** - The compiler flag needed to link Fortran to C MPI wrapper library with all the above mentioned libraries. For `MPICH`, this should be `-lfmpich`. Otherwise, it could be `-lmpe_f2cmpi`, MPE's own Fortran to C MPI wrapper library.
- **FLIB_PATH** - The full compiler flag needed to link Fortran MPI programs with the logging library.

As part of the `make` process, small programs '`cpi.c`' and '`fpi.f`' are linked with each profiling library. The result of each link test will be written as part of the make output. If the link test is successful, then these link paths should be used for user programs as well.

If the MPI implementation being used is `MPICH`, then adding compiler flag `-mpilog` to `MPICH`'s `mpicc`/`mpif77` will automatically link user program with `MPE`'s logging libraries (`-llmpe -lmpe`). Library link path `-lpmpich` is also linked with the MPI profiling interface when `-mpilog` flag is used

If a Fortran MPI program is linked with `MPICH`, it is necessary to include the library '`-lfmpich`' ahead of the profiling libraries. This allows C routines to be used for implementing the profiling libraries for use by both C and Fortran programs. For example, to generate log files in a Fortran program, the library linkage flag is `-lfmpich -llmpe -lmpe -lpmpich`. Using `mpif77 -mpilog` will automatically link with all these libraries.

If the MPI implementation being used is not MPICH, it is necessary to include the library '-lmpe_f2cmpi' (MPE's own Fortran to C MPI wrapper library) instead of library '-lfmpich'. Again this has to be linked before any of the profiling libraries. So the compiler linkage flag will be -lmpe_f2cmpi -llmpe -lmpe -lpmpi -lmpi.

It is possible to combine automatic logging with manual logging. Automatic logging will log all MPI calls and is achieved by linking with $LOG_LIBS. Manual logging is achieved by the user inserting calls to the MPE routines around MPI calls. This way, only the chosen MPI calls will be logged. However, if a combination of these two types of logging is preferred, then the user must NOT call MPE_Init_log and MPE_Finish_log in the user program. Because in the linked logging library, MPI_Init will call MPE_Init_log and MPI_Finalize will call MPE_Finish_log.

2.8 Automatic generation of profiling libraries

For each of these libraries, the process of building the library was very similar. First, profiling versions of MPI_Init and MPI_Finalize must be written. The profiling versions of the other MPI routines are similar in style. The code in each looks like

```
int MPI_Xxx( . . . )
{
       do something for profiling library
    retcode = PMPI_Xxx( . . . );
       do something else for profiling library
    return retcode;
}
```

We generate these routines by writing the "do something" parts only once, in schematic form, and then wrapping them around the PMPI_ calls automatically. It is thus easy to generate profiling libraries. See the README file in mpich/mpe/profiling/wrappergen or Appendix C.

Examples of how to write wrapper templates are located in the mpe/profiling/lib subdirectory. There you will find the source code (the .w files) for creating the three profiling libraries described above. An example Makefile for trying these out is located in the mpe/profiling/examples directory.

2.9 Tools for Profiling Library Management

The sample profiling wrappers for MPICH are distributed as wrapper definition code. The wrapper definition code is run through the wrappergen utility to generate C code (see Section 2.8. Any number of wrapper definitions can be used together, so any level of profiling wrapper nesting is possible when using wrappergen.

A few sample wrapper definitions are provided with MPICH:

timing Use MPI_Wtime() to keep track of the total number of calls to each MPI function, and the time spent within that function. This simply checks the timer before and after the function call. It does not subtract time spent in calls to other functions.

logging Create logfile of all pt2pt function calls.

vismess Pop up an X window that gives a simple visualization of all messages that are passed.

allprof All of the above. This shows how several profiling libraries may be combined.

Note: These wrappers do not use any mpich-specific features besides the MPE graphics and logging used by 'vismess' and 'logging', respectively. They should work on any MPI implementation.

You can incorporate them manually into your application, which involves three changes to the building of your application:

- Generate the source code for the desired wrapper(s) with wrappergen. This can be a one-time task.
- Compile the code for the wrapper(s). Be sure to supply the needed compile-line parameters. 'vismess' and 'logging' require the MPE library ('`-lmpe`'), and the 'vismess' wrapper definition requires MPE_GRAPHICS.
- Link the compiled wrapper code, the profiling version of the mpi library, and any other necessary libraries ('vismess' requires X) into your application. The required order is:

```
$(CLINKER)   <application object files...> \
<wrapper object code> \
<other necessary libraries (-lmpe)> \
<profiling mpi library (-lpmpi)> \
<standard mpi library (-lmpi)>
```

To simplify it, some sample makefile sections have been created in '`mpich/mpe/profiling/lib`':

```
Makefile.timing - timing wrappers
Makefile.logging - logging wrappers
Makefile.vismess - animated messages wrappers
Makefile.allprof - timing, logging, and vismess
```

To use these Makefile fragments:

1. (optional) Add `$(PROF_OBJ)` to your application's dependency list:

```
myapp:  myapp.o $(PROF_OBJ)
```

2. Add `$(PROF_FLG)` to your compile line `CFLAGS`:

   ```
   CFLAGS = -O $(PROF_FLG)
   ```

3. Add `$(PROF_LIB)` to your link line, after your application's object code, but before the main MPI library:

   ```
   $(CLINKER) myapp.o -L$(MPIR_HOME)/lib/$(ARCH)/$(COMM) $(PROF_LIB) -lmpi
   ```

4. (optional) Add `$(PROF_CLN)` to your clean target:

   ```
   rm -f *.o *~ myapp $(PROF_CLN)
   ```

5. Include the desired Makefile fragment in your makefile:

   ```
   include $(MPIR_HOME)/mpe/profiling/lib/Makefile.logging
   ```

   (or

   ```
   #include $(MPIR_HOME)/mpe/profiling/lib/Makefile.logging
   ```

   if you are using the wildly incompatible BSD 4.4-derived make)

3 Using MPE

The Multi-Processing Environment (MPE) attempts to provide programmers with a complete suite of performance analysis tools for their MPI programs based on post processing approach. These tools include a set of profiling libraries, a set of utility programs, and a set of graphical tools.

The first set of tools to be used with user MPI programs is profiling libraries which provide a collection of routines that create log files. These log files can be created manually by inserting MPE calls in the MPI program, or automatically by linking with the appropriate MPE libraries, or by combining the above two methods (see Section 2.5.3). Currently, MPE offers the following three profiling libraries.

- **Tracing Library:** This library traces all MPI calls. Each MPI call is preceded by a line that contains the rank in `MPI_COMM_WORLD` of the calling process, and followed by another line indicating that the call has completed. Most send and receive routines also indicate the values of count, tag, and partner (destination for sends, source for receives). Output is to standard output.
- **Animation Libraries:** A simple form of real-time program animation that requires X window routines.

- **Logging Libraries:** The most useful and widely used profiling libraries in MPE. They form the basis to generate log files from user MPI programs. There are currently three different log file formats allowed in MPE. The default log file format is CLOG. It is basically a collection of events with single timestamps. And there is ALOG which is provided for backward compatibility reason and it is not being developed. And the most powerful one is SLOG, stands for Scalable LOGfile format, which can be converted from CLOG file after CLOG file has been generated (preferred approach), or can be generated directly when MPI program is executed (through setting the environmental variable `MPE_LOG_FORMAT` to SLOG).

The set of utility programs in MPE includes log format converter (e.g. `clog2slog`), logfile print (e.g. `slog_print`) and logfile viewer wrapper, `logviewer`, which selects the correct graphical tool to display the logfile based on the logfile's file extension.

Currently, MPE's graphical tools includes three display programs, `upshot` for ALOG, `Jumpshot-2` for CLOG and `Jumpshot-3` for SLOG. The logviewer script eliminates the need for user to remember the relationship between logfile formats and display programs.

3.1 Directory Structure

The final install directory contains the following subdirectories. In terms of usage of MPE, user usually only need to know about the files that have been installed in '`include/`', '`lib/`' and '`bin/`'.

include/ contains all the include files that user program needs to read.

lib/ contains all the libraries that user program needs to link with.

bin/ contains all the utility programs that user needs to use.

sbin/ contains the MPE uninstall script to uninstall the installation.

share/ contains user read-only data. Besides '`share/examples/`', user usually does not need to know the details of other subdirectories.

3.2 Example Makefile

'`share/examples`' contains some very useful and simple example programs and '`Makefile`' which illustrates the usage of MPE routines and the linkage of MPE libraries to generate logfiles. In most cases, users can simply copy the '`share/examples/Makefile`' to their home directory, and do a `make` to compile the suggested targets. Users don't need to copy the '`.c`' and '`.f`' files when MPE has been compiled with a `make` that has VPATH support. The created executables can be launched with `mpirun` provided by the MPI implementation to generate sample logfiles.

3.3 Environmental Variables

There are 2 environmental variables, `TMPDIR` and `MPE_LOG_FORMAT`, that user may need to set before the generation of logfiles :

`MPE_LOG_FORMAT`**:** determines the format of the logfile generated from the execution of application linked with MPE logging libraries. The allowed value for `MPE_LOG_FORMAT` are CLOG, SLOG and ALOG. When `MPE_LOG_FORMAT` is not set, CLOG is assumed.

TMPDIR : specifies a directory to be used as temporary storage for each process. By default, when `TMPDIR` is not set, '`/tmp`' will be used. When user needs to generate a very large logfile for long-running MPI job, user needs to make sure that `TMPDIR` is big enough to hold the temporary logfile which will be deleted if the merged logfile can be created successfully. In order to minimize the overhead of the logging to the MPI program, it is highly recommended user to use a *local* file system for `TMPDIR`.

Note : The final merged logfile will be written back to the file system where process 0 is running.

3.4 Utility Programs

In '`bin/`', user can find several useful utility programs when manipulating logfiles. These includes log format converters, e.g. `clog2slog`, log format print programs, e.g. `slog_print`, and a script to launch display program, `logviewer`.

3.4.1 Log Format Converters

clog2slog: A CLOG to SLOG logfile converter. Since the automatic generation of SLOG file through setting of environmental variable `MPE_LOG_FORMAT` to SLOG may not work for some non well-behaved MPI programs, using the logfile format converter can generate extra diagnostic information about the condition of the logfile. Also the converter allows one to adjust certain parameters of the logfile, like frame size which is the segment of the logfile to be displayed by `Jumpshot-3`'s Time Line window. For non well behaved MPI program, one may need to increase the frame size from the default 64KB to a bigger value. For more information about the converter, do `clog2slog -h`.

clog2alog: A CLOG to ALOG logfile converter. It is provided here for backward compatibility purpose.

3.4.2 Log Format Print Programs

slog_print: a stdout print program for SLOG file. It serves to check the content of the logfile. If the SLOG file is too big, it may not be useful to use `slog_print`. Also, when slog is not complete, `slog_print` won't work. So it serves as a simple test to check if the SLOG file is generated completely.

clog_print: a stdout print program for CLOG file.

3.4.3 Display Program Selector

logviewer: the script which involves appropriate viewer based on the file extension of logfile. For instance, if the logfile is foo.slog, `logviewer` will invoke `Jumpshot-3` to display the logfile. `Jumpshot-3` resides in '`share/`'. For more information of `logviewer`, do `logviewer -help` to list all available options.

3.5 Using MPE in `MPICH`

MPE has been seamlessly integrated into `MPICH` distribution, so user may find it easier to use MPE when using it with `MPICH`. Here are the differences of using MPE with `MPICH` and with other MPI implementations.

3.5.1 Compilation and Linkage

`MPICH` provides scripts to help users to compile and link C/C++ and F77/F90 programs. They are `mpicc` for C programs, `mpiCC` for C++ programs, `mpif77` for F77 and `mpif90` for F90 programs. In addition, these 4 scripts allows special options to be used to link with MPE profiling libraries. These options are :

-mpitrace to compile and link with tracing library.

-mpianim to compile and link with animation libraries.

-mpilog to compile and link with logging libraries.

For instance, the following command creates executable, `fpilog`, which generates logfile when it is executed.

```
mpif77 -mpilog -o fpilog fpilog.f
```

For other MPI implementations, user needs to compile and link their application with MPE profiling libraries explicitly as shown in the example makefile.

3.5.2 Inheritance of Environmental Variables

MPE relies on certain environmental variables (e.g. `TMPDIR`). These variables determine how MPE behaves. It is important to make sure that all the MPI processes receive the intended value of environmental variables. The complication of this issue comes from the fact that `MPICH` contains many different devices for different platforms, some of these devices have a different way of passing of environmental variables to other processes. The often used devices, like `ch_p4` and `ch_shmem`, do not require special attention to pass the value of the environmental variable to spawned processes. The spawned process inherits the value from the launching process as long as the environmental variable in the launching process is set. But this is not true for all the devices, for instance, the `ch_p4mpd` device may require a special option of mpirun to be used to set the environmental variables to all processes.

```
mpirun -np N cpilog -MPDENV- MPE_LOG_FORMAT=SLOG
```

In this example, the option `-MPDENV-` is needed to make sure that all processes have their environmental variable, `MPE_LOG_FORMAT`, set to SLOG.

For other MPI implementations, how environmental variables are passed remains unchanged. User needs to get familar with the environment and set the environmental variables accordingly.

3.5.3 Viewing Logfiles

MPE's install directory structure is the same as `MPICH`'s. So all MPE's utility programs will be located in `MPICH`'s '`bin/`' directory. To view a logfile, say '`fpilog.slog`', do

```
logviewer fpilog.slog
```

The command will select and invoke `Jumpshot-3` to display the content of SLOG file if `Jumpshot-3` has been built and installed successfully.

4 Debugging MPI programs with built-in tools

Debugging of parallel programs is notoriously difficult, and we do not have a magical solution to this problem. Nonetheless, we have built into `MPICH` a few features that may be of use in debugging MPI programs.

4.1 Error handlers

The MPI Standard specifies a mechanism for installing one's own error handler, and specifies the behavior of two predefined ones, `MPI_ERRORS_RETURN` and `MPI_ERRORS_ARE_FATAL`. As part of the `MPE` library, we include two other error handlers to facilitate the use of command-line debuggers such as `dbx` in debugging MPI programs.

```
MPE_Errors_call_dbx_in_xterm
MPE_Signals_call_debugger
```

These error handlers are located in the `MPE` directory. A configure option (`-mpedbg`) includes these error handlers into the regular MPI libraries, and allows the command-line argument `-mpedbg` to make `MPE_Errors_call_dbx_in_xterm` the default error handler (instead of `MPI_ERRORS_ARE_FATAL`).

4.2 Contents of the library files

The directory containing the MPI library file ('`libmpich.a`') contains a few additional files. These are summarized here

libmpe.a contains MPE graphics, logging, and other extensions (`PMPE_Xxxx`)

libmpe_nompi.a contains MPE graphics without MPI

libampe.a contains MPE Animation interface

liblmpe.a contains MPE Logging interface

libtmpe.a contains MPE Tracing interface

libmpe_f2cmpi.a contains MPE Fortran to C MPI wrapper interface

mpe_prof.o Sample profiling library (C)

Appendices

A Installing MPE

Users of MPICH do not need to make or install MPE separately; MPE is normally built as part of MPICH. This section is provided primarily to help users who wish to use MPE with other implementations of MPI. In addition, for users who are having trouble building MPE as part of MPICH (e.g., trouble finding an appropriate Java installation), this section provides some suggestions.

A.1 Configuration

MPE can be configured and installed as an extension to most MPI standard compliant MPI implementations, e.g. MPICH, LAM/MPI, SGI's MPI, HP-UX's MPI and IBM's MPI. It has been integrated seamlessly into MPICH distribution, so MPE will be installed automatically during MPICH's installation process.

A.1.1 Configuration Model

MPE is designed to be used as an extension to an existing MPI implementation, so its configuration model assumes a general MPI development environment. Here are the some of the variables that MPE configure reads, some are read as environmental variables and some are read from the command line arguments to configure.

CC: C compiler used to create serial executable, e.g. `xlc` for IBM MPI.

MPI_CC: C compiler used to compile MPI program and to create parallel executable, e.g. `mpcc` for IBM MPI, or `mpicc` for MPICH.

MPE_CFLAGS: `CFLAGS` for `CC` and `MPI_CC`.

F77: F77 compiler used to create serial executable, e.g. `xlf` for IBM MPI.

MPI_F77: F77 compiler used to compile MPI program and to create parallel executables, e.g. `mpxlf` for IBM MPI, or `mpif77` for MPICH.

MPE_FFLAGS: FFLAGS for `F77` and `MPI_F77`.

MPI_INC: compiler's include flag (with prefix `-I`) for `MPI_CC/MPI_F77`, e.g. `-I/usr/include` for '`mpi.h`' on IRIX64.

MPI_LIBS: compiler's library flag (with prefix `-L` for library path and prefix `-l` for each library name) needed by `MPI_CC/MPI_F77`, e.g. `-L/usr/lib -lmpi` for '`libmpi.a`' on IRIX64.

F2CMPI_LIBS: compiler's library flag for Fortran to C MPI wrapper library, e.g. `-lfmpich` when `MPI_CC=mpicc` and `MPI_F77=mpif77` for MPICH.

Among above listed variables, `CC`, `MPI_CC`, `F77` and `MPI_F77` are usually set by the corresponding environmental variables. The rest can be set through command line arguments to configure. In some MPI implementations, like HP-UX's, `MPI_CC` and `MPI_F77` are reserved for use by the MPI implementation, use the configure options to set `MPI_CC` and `MPI_F77` instead.

A.1.2 Build Options and Features

MPE's configure is written using autoconf 2, and supports VPATH style install process. It means the actual source directory and the building directory can be in 2 different locations. This allows the same source directory to be used to build multiple versions of MPE with different options and still won't mess up the original source. It is highly recommended that user should do a VPATH build. Also MPE involves several different independent packages, in order to create a tightly integrated environment for user, it is recommended that user should do a `make install` to install the MPE in a separate directory after the build is done. The benefit is that all utility programs will be in '`bin/`', all libraries will be in '`lib/`' and all graphic tools will be nicely organized in '`share/`' ...

There are 2 types of configure options.

1. MPI implementation and User options
2. Generic configure flags supplied by autoconf 2

For a list of flags/switches for type 1 (not type 2) in MPE, use the script `configure--help`.

The following is not a complete list but some of the more important ones. Generic flags:

`--prefix=INSTALL_DIR` Specifies the final install directory for `make install`. All libraries, utility programs, graphic programs and examples are installed in a standard directory structure without files created in the building process.

`--x-includes=X_INC` Specifies the directory where X include files are located. This is used when configure has trouble in locating X in user system.

`--x-libraries=X_LIBS` Specifies the directory where X libraries are located. This is used when configure has trouble in locating X in user system.

MPI implementation Options:

`--with-mpicc=MPI_CC` Specify MPI C compiler to generate parallel executable, e.g. mpcc for AIX.

`--with-mpif77=MPI_F77` Specify MPI F77 compiler to generate parallel executable, e.g. mpxlf for AIX.

`--with-cflags=MPE_CFLAGS` Specify extra CFLAGS to the C and `MPI_CC` compilers, e.g. `-64` for IRIX64 C compiler

`--with-fflags=MPE_FFLAGS` Specify extra FFLAGS to the F77 and `MPI_F77` compilers, e.g. `-64` for IRIX64 F77 compiler

`--with-mpiinc=MPI_INC` Specify compiler's include flag for MPI include directory, e.g. `-I/pkgs/MPI/include` for 'mpi.h'

`--with-mpilibs=MPI_LIBS` Specify compiler's library flag for MPI libraries, e.g. `-L/pkgs/MPI/lib -lpmpich -lmpich`

`--enable-f77` Enable the compilation of routines that require a Fortran compiler. If configuring with `MPICH`, the configure in the top-level `MPICH` directory will choose the appropriate value for you. However, it can be overridden. The default is yes, `--enable-f77`.

`--enable-f2cmpilib` Enable the building of MPE's internal Fortran to C MPI wrapper library. The default is yes, `--enable-f2cmpilib`

`--with-f2cmpilibs=F2CMPI_LIBS` Specify compiler's library flags for Fortran to C MPI wrapper library. Using this option will force `--disable-f2cmpilib`. e.g. `-lfmpich` when configuring MPE for `MPICH`

Other User Options:

`--enable-echo` Turn on strong echoing. The default is no, `--disable-echo`.

`--with-mpelibname=MPE_LIBNAME` Specify the MPE library name instead of the default 'mpe'. e.g. if `MPE_LIBNAME="MPE"`, then the libraries generated will be '`libMPE.a`', '`liblMPE.a`', '`libtMPE.a`', '`libaMPE.a`' and '`libMPE_f2cmpi.a`'. This option is necessary when configuring MPE for a existing and older version of `MPICH` which has MPE installed.

`--enable-mpe_graphics` Enable the building of MPE graphics routines. If disabled, then the MPE routines that make use of X11 graphics will not be built. This is appropriate for systems that either do not have the X11 include files or that do not support X11 graphics. The default is enable=yes.

`--enable-viewers` Enable the build of all the available log viewers. The default is enable=yes

`--with-java=JAVA_HOME` Specify the path of the top-level directory of the Java, JDK, installation. If this option or `--with-javaX` is not given, configure will try to locate JDK for you to build Jumpshot-2 and Jumpshot-3. JDK 1.1.6 to JDK 1.1.8 can be used to build both Jumpshots. It is recommended that user should use this option when there is only one version of JDK installed on the machine.

`--with-java1=JAVA_HOME` Specify the path of the top-level directory of the Java, JDK, installation for Jumpshot-2 only. If this option or `--with-java` is not given, Jumpshot-2's configure will try to locate JDK for you to build Jumpshot-2. For performance reason, it is recommended to use the latest Java 1, i.e. JDK-1.1.8, to build Jumpshot-2.

--with-java2=JAVA_HOME Specify the path of the top-level directory of the Java, JDK, installation for Jumpshot-3 only. If this option or --with-java is not given, Jumpshot-3's configure will try to locate JDK for you to build Jumpshot-3. For performance reason, it is recommended to use the latest Java 2, i.e. JDK-1.3.x, to build Jumpshot-3.

--with-wishloc=WISHLOC This switch specifies the name of Tcl/Tk wish executable. If this switch is omitted, configure will attempt to locate a version. This is used only for upshot. Note: Because Tcl and Tk keep changing in incompatible ways, we will soon be dropping support for any tool that uses Tcl/Tk.

In order to achieve maximum performance benefit, it is recommended to use latest Java 1, i.e. JDK-1.1.8, as well as latest Java 2, i.e. JDK-1.3.1 to build Jumpshots. So instead of using the --with-java=<JDK-1.1.8> option, --with-java1=<JDK-1.1.8> and --with=java2=<JDK-1.3.1> should be used instead.

A.2 Installation Instructions

As noted earlier, the MPE library can be installed as part of the MPICH configure or as an extension of an existing MPI implementation. Below are instructions and examples for typical installation of MPE on popular MPI implementations.

A.2.1 Configuring as part of the MPICH configure

The configure in the MPICH directory will try to determine the necessary information and pass it to the MPE configure. If no options are given, the MPE will automatically be configured by default. However, the user can provide extra configuration information to MPE through MPICH configure with the following options:

```
-mpe_opts=MPE_OPTS
```

where MPE_OPTS is one or more of the choices in section A.1.2. Multiple instances of -mpe_opts are allowed in MPICH configure to specify different options for the MPE. The following is a configure option which specifies a non-default location of JDK installation.

```
-mpe_opts=--with-java=/sandbox/chan/java/1.1.8
```

A.2.2 Configuring as part of an existing MPI implementation

The following are some examples for configuring MPE for an existing MPI implementation.

- **For SGI MPI**, do the following for default application binary interface (ABI), -n32:

  ```
  setenv MAKE gmake
  ```

```
${MPE_SRC_DIR}/configure --with-mpilibs=-lmpi \
                         --with-java=/usr/java-1.1.6/usr/java
make
make install PREFIX=${MPE_INSTALL_DIR}
```

for the 64 bit ABI, add options `--with-cflags=-64` and `--with-fflags=-64` to the configure options.

- **For IBM MPI**, do

```
setenv MPI_CC mpcc
setenv MPI_F77 mpxlf
${MPE_SRC_DIR}/configure --with-java=/homes/chan/pkgs/java/J1.1.8
make
make install PREFIX=${MPE_INSTALL_DIR}
```

- **For HP-UX's MPI**, do

```
${MPE_SRC_DIR}/configure --with-mpicc=mpicc \
                         --with-mpif77=mpif77 \
                         --with-flib_path_leader="-Wl,-L"
make
make install PREFIX=${MPE_INSTALL_DIR}
```

MPE's Fortran support on HP-UX's MPI is not working yet. So to get MPI Fortran code to generate logfile, you could use HP-UX's '`libfmpi.a`' if it is there. Here is the configure options:

```
${MPE_SRC_DIR}/configure --with-mpicc=mpicc \
                         --with-mpif77=mpif77 \
                         --with-flib_path_leader="-Wl,-L" \
                         --with-f2cmpilibs=-lfmpi
make
make install PREFIX=${MPE_INSTALL_DIR}
```

- **For an existing** `MPICH`, do

```
setenv MPI_CC ${MPICH_INSTALL_DIR}/mpicc
setenv MPI_F77 ${MPICH_INSTALL_DIR}/mpif77
${MPE_SRC_DIR}/configure  --with-f2cmpilibs=-lfmpich \
                          --with-mpelibname=newMPE \
                          --with-java1=/sandbox/jdk117_v3 \
                          --with-java2=/sandbox/jdk1.3.1
make
make install PREFIX=${MPE_INSTALL_DIR}
```

It is important to use the configure option –with-mpelibname to specify a different MPE library name than the default "mpe" when configuring MPE for older MPICH. Without this option, the linkage tests in MPE would most likely use the old MPE libraries in the MPICH instead of the newly built MPE libraries in resolving the MPE symbols. Also the option --with-f2cmpilibs forces MPE to use the Fortran to C MPI wrapper library in previous version of MPICH.

- **For LAM**, do

```
setenv MPI_CC ${LAM_INSTALL_DIR}/bin/mpicc
setenv MPI_f77 ${LAM_INSTALL_DIR}/bin/mpif77
${MPE_SRC_DIR}/configure --with-mpilibs="-L${LAM_INSTALL_DIR}/lib -lpmpi" \
                         --with-java1=/sandbox/jdk117_v3 \
                         --with-java2=/sandbox/jdk1.3.1

make
make install PREFIX=${MPE_INSTALL_DIR}
```

Using MPE with LAM for fortran MPI program is not working until recently, i.e. MPE in MPICH-1.2.1 or later. Configure options listed above enable MPE's internal Fortran to C MPI library.[2] To use LAM's Fortran to C MPI library in LAM 6.3.3 or later ('liblamf77mpi.a'), do

```
setenv MPI_CC ${LAM_INSTALL_DIR}/bin/mpicc
setenv MPI_f77 ${LAM_INSTALL_DIR}/bin/mpif77
${MPE_SRC_DIR}/configure --with-mpilibs="-L${LAM_INSTALL_DIR}/lib -lpmpi" \
                         --with-f2cmpilibs=-llamf77mpi \
                         --with-java1=/sandbox/jdk117_v3 \
                         --with-java2=/sandbox/jdk1.3.1
make
make install PREFIX=${MPE_INSTALL_DIR}
```

A.3 Install/Uninstall Scripts

A mpeinstall script is created during configuration. If configuring with MPICH, then the mpiinstall script will invoke the mpeinstall script. However, mpeinstall can also be used by itself. This is only optional and is of use only if you wish to install the MPE library in a public place so that others may use it. Final install directory will consist of an 'include', 'lib', 'bin', 'sbin' and 'share' subdirectories. Examples and various logfile viewers will be installed under 'share'. The 'sbin' in installed directory contains an MPE uninstall script mpeuninstall.

[2] If you have MPE in MPICH-1.2.1, you need to download a patch from MPICH's FTP server to fix MPE before building MPE with LAM. The URL is ftp://ftp.mcs.anl.gov//pub/mpi/patch/1.2.1/5524

B Installing Java for Jumpshots

B.1 Java requirements

MPE includes a '`viewers`' subdirectory, which is an independent package, includes 2 versions of Jumpshots, `Jumpshot-2` and `Jumpshot-3`. Both need to be built with Java Development Kit (`JDK`), i.e. Java distribution that includes a Java compiler, `javac`. Theoretically, Jumpshots can be distributed with precompiled byte code instead of source code that needs to be compiled. The main reason is that there exists JDK which isn't compatible with Swing and one of the easiest ways to detect this problem is to compile and link the code to see if things are fine. We are still planning to distribute byte code in later version.

`Jumpshot-2` and `Jumpshot-3` are both developed based on Sun's JDK. `Jumpshot-2` is developed based on Java1, JDK-1.1 and Swing-1.0.3. So it can only be built with JDK-1.1, not JDK-1.2 or later. On the other hand, `Jumpshot-3` is developed based on JDK-1.1 and Swing-1.1.1, so it can be built by both Java1 and Java2, i.e. JDK-1.1 to JDK-1.4. It is recommended that user should use the latest Java1, i.e. JDK-1.1.8, to build `Jumpshot-2`, and use the latest Java2, i.e. JDK-1.4.x to build `Jumpshot-3` for optimal performance when configuring MPE or `MPICH`. `Jumpshot-2` understands both configure options: `--with-java` and `--with-java1` options. `Jumpshot-3` understands `--with-java` and `--with-java2` options. If only one version of JDK is installed, like JDK-1.1.8, `--with-java` option should be used. If both JDK-1.1.8 and JDK-1.4.x are installed on the machine, viewers package can be configured by defining `--with-java1=<JDK-1.1.8>` and `--with-java2=<JDK-1.4.0>`.

We will list the status of the JDK distributions on some of the popular UNIX platforms that we have tested and what is needed to build the JDK properly for Jumpshots.

- **Linux**: There are many choices of JDKs for Linux running on Intel x86 processors. `Blackdown.org` has released many different versions of JDK for Linux on this platform, including both Java1 and Java2. You can download them by locating the closest FTP mirror site of `http://www.blackdown.org`, one of the popular download sites in US is `ftp://metalab.unc.edu/pub/linux/devel/lang/java/blackdown.org/`. Pick the JDK distribution that has the correct '`libc`' for your platform.

  Sun also distributes JDK-1.2 and JDK-1.3 for Linux. Here are the URLs for download.

  `http://www.javasoft.com/products/jdk/1.2/download-linux.html`
  `http://java.sun.com/j2se/1.3/download-linux.html.`
  `http://java.sun.com/j2se/1.4/download.html.`

  As soon as the package is unpacked, it should be ready to compile Jumpshots.

- **Linux/alpha**: Linux running on alpha processors has a limited choice of JDK. The only versions that have been tested to be able to compile Jumpshots is JDK-1.1.8 from `alphalinux.org`. Here are the URL:

  `ftp://ftp.alphalinux.org/pub/java/java-1.1/r2/jdk118\_RH60\_alpha\_bin\_21164\_v2.tgz`

  or

  `ftp://ftp.alphalinux.org/pub/java/java-1.1/r3/jdk118\_RH60\_alpha\_bin\_21164\_v3.tgz`

Since the distribution does not come with the classes file, you need to download the classes file separately. The URL is

`ftp://ftp.alphalinux.org/pub/java/java-1.1/r2/jdk118\_alpha\_classes\_v2.tgz`

Since Jumpshots need file '`classes.zip`', so after unzipped the file, be sure to do the following to generate the '`classes.zip`'.

```
cd jdk118/classes
zip -0 ../lib/classes.zip *
```

As opposed to JDK for Linux, '`libawt.so`' is dynamically linked instead of statically linked. So when running Jumpshot, it will complain about missing file '`libXm.so`' if your system doesn't have Motif installed. We installed Lesstif which seems to resolve this issue.

Bugs: The "nesting" of states in `Jumpshot-3` does not yet work.

- **Solaris**: We have tested JDKs as old as JDK-1.1.6 under Solaris, e.g. `Solaris_-JDK_1.1.6_03`. [3]

- **IRIX64**: We have tested JDK-1.1.6, JDK-1.1.7 and JDK-1.1.8 from SGI, they all seem to work fine with Jumpshots. JDK-1.1.8 seems to work best with Jumpshot on IRIX. You can download them at

  `http://www.sgi.com/developers/devtools/languages/java.html`.

- **AIX**: Only JDK-1.1.8 for AIX has been tested with Jumpshot. But newer JDK for AIX should work with Jumpshot. You can download it at

  `http://www.ibm.com/java/jdk/aix/index.html`

- **HP-UX**: HP distributes JDK for its HP-UX OS. None of JDK for HP-UX has been tested with Jumpshots because of lack of access to the platform. The URL for HP's JDK is:

  `http://www.unixsolutions.hp.com/products/java/index.html`

- **Windows and X11**: MPE has been ported (at least the logging part of it) to Windows, the Java requirement for running Jumpshot on Windows isn't an issue and the installation of Java on Windows is straight forward as well. However, many users use Windows desktops to access UNIX servers to run their MPI programs. They run Jumpshot remotely and display it back to their desktops through running X11 server locally. Let's assume SSH is used to connect to the UNIX servers. We notice that there are some incompatibilities between various X11 servers and different versions of JDKs. In particular, some versions of Exceed X11 servers when used with SecureCRT (a version of SSH) will crash when displaying `Jumpshot-2` compiled with JDK-1.1.7 from Blackdown. Using SuperX instead of Exceed causes `Jumpshot-2` to hang in this situation. We notice that using Cygwin's OpenSSH and XFree86 seems to avoid hangs and crashes. XFree86 has its strengths and weakness when compared to other

[3] Swing 1.1.1 requires at least JDK 1.1.7, so in principle `Jumpshot-3` may have problem with JDK-1.1.6 on some implementations.

commerical X servers. XFree86 seems to be slower when the desktop hardware is old and especially when Java 2 (JDK 1.3.1) is used. XFree86 performs remarkablely well when used with JDK 1.1 instead of JDK 1.3. The combination being tested is OpenSSH 3.0.1p1-4 and XFree86 4.1.

C Automatic generation of profiling libraries

The profiling wrapper generator (`wrappergen`) has been designed to complement the MPI profiling interface. It allows the user to write any number of 'meta' wrappers which can be applied to any number of MPI functions. Wrappers can be in separate files, and can nest properly, so that more than one layer of profiling may exist on indiividual functions.

`Wrappergen` needs three sources of input:

1. A list of functions for which to generate wrappers.
2. Declarations for the functions that are to be profiled. For speed and parsing simplicity, a special format has been used. See the file '`proto`'. The MPI-1 functions are in '`mpi_proto`'. The I/O functions from MPI-2 are in '`mpiio_proto`'.
3. Wrapper definitions.

The list of functions is simply a file of whitespace-separated function names. If omitted, any `forallfn` or `fnall` macros will expand for every function in the declaration file.

If no function declarations are provided, the ones in '`mpi_proto`' are used (this is set with the `PROTO_FILE` definition in the '`Makefile`').

The options to `wrappergen` are:

-w file Add file to the list of wrapper files to use.

-f file file contains a whitespace separated list of function names to profile.

-p file file contains the special function prototype declarations.

-o file Send output to file.

For example, to time each of the I/O routines, use

```
cd mpe/profiling/lib
../wrappergen/wrappergen -p ../wrappergen/mpiio_proto \
        -w time_wrappers.w > time_io.c
```

The resulting code need only a version of `MPI_Finalize` to output the time values. That can be written either by adding `MPI_Finalize` and `MPI_Init` to '`mpiio_proto`' or through a fairly simple edit of the version produced when using '`mpi_proto`' instead of '`mpiio_proto`'.

C.1 Writing wrapper definitions

Wrapper definitions themselves consist of C code with special macros. Each macro is surrounded by the {{ }} escape sequence. The following macros are recognized by `wrappergen`:

`{{fileno}}`

An integral index representing which wrapper file the macro came from. This is useful when declaring file-global variables to prevent name collisions. It is suggested that all identifiers declared outside functions end with `_{{fileno}}`. For example:

```
static double overhead_time_{{fileno}};
```

might expand to:

```
static double overhead_time_0;
```

(end of example).

```
{{forallfn <function name escape> <function A> <function B> ... }}
   ...
{{endforallfn}}
```

The code between `{{forallfn}}` and `{{endforallfn}}` is copied once for every function profiled, except for the functions listed, replacing the escape string specified by `<function name escape>` with the name of each function. For example:

```
{{forallfn fn_name}}static int {{fn_name}}_ncalls_{{fileno}};
{{endforallfn}}
```

might expand to:

```
static int MPI_Send_ncalls_1;
static int MPI_Recv_ncalls_1;
static int MPI_Bcast_ncalls_1;
```

(end of example)

```
{{foreachfn <function name escape> <function A> <function B> ... }}
   ...
{{endforeachfn}}
```

`{{foreachfn}}` is the same as `{{forallfn}}` except that wrappers are written only the functions named explicitly. For example:

```
{{forallfn fn_name mpi_send mpi_recv}}
   static int {{fn_name}}_ncalls_{{fileno}};
{{endforallfn}}
```

might expand to:

```
static int MPI_Send_ncalls_2;
static int MPI_Recv_ncalls_2;
```

(end of example)

```
{{fnall <function name escape> <function A> <function B> ... }}
  ...
  {{callfn}}
  ...
{{endfnall}}
```

`{{fnall}}` defines a wrapper to be used on all functions except the functions named. `Wrappergen` will expand into a full function definition in traditional C format. The `{{callfn}}` macro tells `wrappergen` where to insert the call to the function that is being profiled. There must be exactly one instance of the `{{callfn}}` macro in each wrapper definition. The macro specified by `<function name escape>` will be replaced by the name of each function.

Within a wrapper definition, extra macros are recognized.

`{{vardecl <type> <arg> <arg> ... }}`

Use `vardecl` to declare variables within a wrapper definition. If nested macros request variables through vardecl with the same names, `wrappergen` will create unique names by adding consecutive integers to the end of the requested name (var, var1, var2, ...) until a unique name is created. It is unwise to declare variables manually in a wrapper definition, as variable names may clash with other wrappers, and the variable declarations may occur later in the code than statements from other wrappers, which is illegal in classical and ANSI C.

`{{<varname>}}`

If a variable is declared through `vardecl`, the requested name for that variable (which may be different from the uniquified form that will appear in the final code) becomes a temporary macro that will expand to the uniquified form. For example,

```
{{vardecl int i d}}
```

may expand to:

```
int i, d3;
```

(end of example)

`{{<argname>}}`

Suggested but not neccessary, a macro consisting of the name of one of the arguments to the function being profiled will be expanded to the name of the corresponding argument. This macro option serves little purpose other than asserting that the function being profilied does indeed have an argument with the given name.

`{{<argnum>}}`

Arguments to the function being profiled may also be referenced by number, starting with 0 and increasing.

`{{returnVal}}`

`ReturnVal` expands to the variable that is used to hold the return value of the function being profiled.

`{{callfn}}`

`callfn` expands to the call of the function being profiled. With nested wrapper definitions, this also represents the point at which to insert the code for any inner nested functions. The nesting order is determined by the order in which the wrappers are encountered by `wrappergen`. For example, if the two files '`prof1.w`' and '`prof2.w`' each contain two wrappers for `MPI_Send`, the profiling code produced when using both files will be of the form:

```
int MPI_Send( args...)
arg declarations...
{
   /*pre-callfn code from wrapper 1 from prof1.w */
   /*pre-callfn code from wrapper 2 from prof1.w */
   /*pre-callfn code from wrapper 1 from prof2.w */
   /*pre-callfn code from wrapper 2 from prof2.w */

   returnVal = MPI_Send( args... );

   /*post-callfn code from wrapper 2 from prof2.w */
   /*post-callfn code from wrapper 1 from prof2.w */
   /*post-callfn code from wrapper 2 from prof1.w */
   /*post-callfn code from wrapper 1 from prof1.w */

   return returnVal;
}
```

```
{{fn <function name escape>  <function A> <function B> ... }}
  ...
  {{callfn}}
  ...
{{endfnall}}
```

`fn` is identical to `fnall` except that it only generates wrappers for functions named explicitly. For example:

```
{{fn this_fn MPI_Send}}
  {{vardecl int i}}
  {{callfn}}
  printf( "Call to {{this_fn}}.\n" );
  printf( "{{i}} was not used.\n" );
  printf( "The first argument to {{this_fn}} is {{0}}\n" );
{{endfn}}
```

will expand to:

```
int  MPI_Send( buf, count, datatype, dest, tag, comm )
void * buf;
int count;
MPI_Datatype datatype;
int dest;
int tag;
MPI_Comm comm;
{
  int  returnVal;
  int i;
  returnVal = PMPI_Send( buf, count, datatype, dest, tag, comm );
  printf( "Call to MPI_Send.\n" );
  printf( "i was not used.\n" );
  printf( "The first argument to MPI_Send is buf\n" );
  return returnVal;
}
```

`{{fn_num}}`

This is a number, starting from zero. It is incremented every time it is used.

A sample wrapper file is in '`sample.w`' and the corresponding output file is in '`sample.out`'.

D Manual Pages

The MPE routines can be divided into six classes:

Logging routines

MPE_Describe_event
MPE_Describe_state
MPE_Finish_log
MPE_Init_log
MPE_Log_event
MPE_Log_get_event_number
MPE_Log_receive
MPE_Log_send

MPE_Start_log
MPE_Stop_log

X window graphics

MPE_Add_RGB_color
MPE_CaptureFile
MPE_Close_graphics
MPE_Draw_circle
MPE_Draw_line
MPE_Draw_logic
MPE_Draw_point
MPE_Draw_points
MPE_Draw_string
MPE_Fill_circle
MPE_Fill_rectangle
MPE_Get_mouse_press
MPE_Iget_mouse_press
MPE_Line_thickness
MPE_Make_color_array
MPE_Num_colors
MPE_Open_graphics
MPE_Update

Sequential Section

MPE_Seq_begin
MPE_Seq_end

Shared counter

MPE_Counter_create
MPE_Counter_free
MPE_Counter_nxtval

Tag management

MPE_GetTags
MPE_ReturnTags
MPE_TagsEnd

Miscellaneous

MPE_Comm_global_rank
MPE_Decomp1d
MPE_Errors_call_debugger
MPE_IO_Stdout_to_file
MPE_Print_datatype_pack_action
MPE_Print_datatype_unpack_action
MPE_Ptime
MPE_Signals_call_debugger
MPE_Wtime

Acknowledgments

The work described in this report has benefited from conversations with and use by a large number of people. We also thank those that have helped in the implementation of MPE, particularly Edward Karrels.

Debbie Swider had maintained MPE in MPICH-1.2.0 and before. Omer Zaki did the first implementation of Jumpshot, `Jumpshot-2`. Abhi Shandilya did the original `clog2slog` converter. Anthony Chan implemented `SLOG-API`, extended `Jumpshot-2` to `Jumpshot-3` to support SLOG, and integrated `SLOG-API`, `clog2slog`, `Jumpshot-2` and `Jumpshot-3` into MPE.

References

[1] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI Message-Passing Interface standard. *Parallel Computing*, 22(6):789–828, 1996.

[2] M. T. Heath. Recent developments and case studies in performance visualization using ParaGraph. In G. Haring and G. Kotsis, editors, *Performance Measurement and Visualization of Parallel Systems*, pages 175–200. Elsevier Science Publishers, 1993.

[3] Virginia Herrarte and Ewing Lusk. Studying parallel program behavior with `upshot`. Technical Report ANL–91/15, Argonne National Laboratory, 1991.

[4] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *International Journal of Supercomputer Applications*, 8(3/4):165–414, 1994.

[5] C. Eric Wu, Anthony Bolmarcich, Marc Snir, David Wootton, Farid Parpia, Anthony Chan, Ewing L. Lusk, and William Gropp. From trace generation to visualization: A performance framework for distributed parallel systems.

[6] Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. Toward scalable performance visualization with Jumpshot. *High Performance Computing Applications*, 13(2):277–288, Fall 1999.