ANL/MCS-TM-ANL-96/6 Rev D

User's Guide for mpich, a Portable Implementation of MPI Version 1.2.2

by

William Gropp and Ewing Lusk

MATHEMATICS AND
COMPUTER SCIENCE
DIVISION

Contents

Abstract 1

1 Introduction 2

2 Linking and running programs 3
- 2.1 Scripts to Compile and Link Applications . . . 3
  - 2.1.1 Using Shared Libraries . . . 4
  - 2.1.2 Fortran 90 and the MPI module . . . 6
- 2.2 Compiling and Linking without the Scripts . . . 6
- 2.3 Running with mpirun . . . 6

3 Special features of different systems 7
- 3.1 MPMD Programs . . . 7
- 3.2 Workstation clusters and the `ch_p4` device . . . 7
  - 3.2.1 Checking your machines list . . . 8
  - 3.2.2 Using the Secure Shell . . . 9
  - 3.2.3 Using the Secure Server . . . 9
  - 3.2.4 SMP Clusters . . . 9
  - 3.2.5 Heterogeneous networks and the ch_p4 device . . . 10
  - 3.2.6 The P4 Procgroup File . . . 11
  - 3.2.7 Tuning P4 Performance . . . 12
  - 3.2.8 Using special interconnects . . . 13
  - 3.2.9 Using Shared Libraries with the `ch_p4` device . . . 13
- 3.3 Fast Startup with the Multipurpose Daemon and the `ch_p4mpd` Device . . . 14
  - 3.3.1 Goals . . . 14
  - 3.3.2 Introduction . . . 14
  - 3.3.3 Examples . . . 16
  - 3.3.4 How the Daemons Work . . . 18
  - 3.3.5 Running MPICH Jobs under MPD . . . 19
- 3.4 Debugging MPI Programs . . . 20
  - 3.4.1 The `printf` Approach. . . . 20
  - 3.4.2 Using a Commercial Debugger. . . . 20
  - 3.4.3 Using `mpigdb`. . . . 20
- 3.5 Symmetric Multiprocessors (SMPs) and the `ch_shmem` device . . . 21
- 3.6 Computational Grids: the `globus2` device . . . 22
- 3.7 MPPs . . . 22

4 Sample MPI programs 23

5 The MPE library of useful extensions 23
- 5.1 Logfile Creation . . . 24
- 5.2 Logfile Format . . . 24
- 5.3 Parallel X Graphics . . . 24
- 5.4 Other `MPE` Routines . . . 24
- 5.5 Profiling Libraries . . . 25
  - 5.5.1 Accumulation of Time Spent in MPI routines . . . 25
  - 5.5.2 Automatic Logging . . . 26

5.5.3 Customized Logging . . . 26
5.5.4 Real-Time Animation . . . 27
5.6 Logfile Viewers . . . 27
5.6.1 Upshot and Nupshot . . . 27
5.6.2 Jumpshot-2 and Jumpshot-3 . . . 27
5.7 Automatic generation of profiling libraries . . . 28
5.8 Tools for Profiling Library Management . . . 29

6 Debugging MPI programs with built-in tools 31
6.1 Error handlers . . . 31
6.2 Command-line arguments for `mpirun` . . . 31
6.3 MPI arguments for the application program . . . 31
6.4 Debugging with the `ch_p4` Device . . . 32
6.4.1 p4 Debugging . . . 32
6.4.2 Setting the Working Directory for the `p4` Device . . . 33
6.5 Command-line arguments for the application program . . . 33
6.6 Starting jobs with a debugger . . . 33
6.7 Starting the debugger when an error occurs . . . 34
6.8 Attaching a debugger to a running program . . . 34
6.9 Signals . . . 34
6.10 Related tools . . . 35

7 Debugging MPI programs with TotalView 35
7.1 Preparing `mpich` for TotalView debugging . . . 35
7.2 Starting an `mpich` program under TotalView control . . . 35
7.3 Attaching to a running program . . . 36
7.4 Debugging with TotalView . . . 36

8 Other MPI Documentation 36

9 In Case of Trouble 37
9.1 Problems compiling or linking Fortran programs . . . 38
9.1.1 General . . . 38
9.2 Problems Linking C Programs . . . 39
9.2.1 General . . . 39
9.2.2 Sun Solaris . . . 40
9.2.3 HPUX . . . 40
9.2.4 LINUX . . . 40
9.3 Problems starting programs . . . 41
9.3.1 General . . . 41
9.3.2 Workstation Networks . . . 42
9.3.3 IBM RS6000 . . . 49
9.3.4 IBM SP . . . 49
9.4 Programs fail at startup . . . 51
9.4.1 General . . . 51
9.4.2 Workstation Networks . . . 51
9.5 Programs fail after starting . . . 52
9.5.1 General . . . 52

9.5.2 HPUX . . . 54
9.5.3 ch_shmem device . . . 54
9.5.4 LINUX . . . 54
9.5.5 Workstation Networks . . . 54
9.6 Trouble with Input and Output . . . 55
9.6.1 General . . . 55
9.6.2 IBM SP . . . 55
9.6.3 Workstation Networks . . . 56
9.7 Upshot and Nupshot . . . 56
9.7.1 General . . . 56
9.7.2 HP-UX . . . 57

A Automatic generation of profiling libraries **58**
A.1 Writing wrapper definitions . . . 58

B Options for `mpirun` **62**

C `mpirun` and Globus **65**
C.1 Using `mpirun` To Construct An RSL Script For You . . . 66
C.1.1 Using `mpirun` By Supplying Your Own RSL Script . . . 67

Acknowledgments **68**

D Deprecated Features **69**
D.1 More detailed control over compiling and linking . . . 69

This *User's Guide* corresponds to Version 1.2.2 of `mpich`.

Abstract

MPI (Message-Passing Interface) is a standard specification for message-passing libraries. `mpich` is a portable implementation of the full MPI specification for a wide variety of parallel and distributed computing environments. This paper describes how to build and run MPI programs using the `mpich` implementation of MPI.

This document describes how to use `mpich` [9], the portable implementation of the MPI Message-Passing Standard. Details on acquiring and installing the `mpich` implementation are presented in a separate *Installation Guide* for `mpich` [6]. Version 1.2.2 of `mpich` is primarily a bug fix and increased portability release, particularly for LINUX-based clusters.

New and improved in 1.2.2:

- A greatly improved `ch_p4mpd` device.
- Improved support for assorted Fortran 77 and Fortran 90 compilers, including compile-time evaluation of Fortran constants used in the `mpich` implementation.
- An improved `globus2` device, providing better performance.
- A new `bproc` mode for the `ch_p4` device supports Scyld Beowulfs.
- Many TCP performance improvements for the `ch_p4` and `ch_p4mpd` devices, as well as
- Many bug fixes and code improvements. See `www.mcs.anl.gov/mpi/mpich/r1_2_2changes.html` for a complete list of changes.

Features that were new in 1.2.1 included:

- Improved support for assorted Fortran and Fortran 90 compilers. In particular, a single version of `mpich` can now be built to use several different Fortran compilers; see the installation manual (in doc/install.ps.gz) for details.
- Using a C compiler for MPI programs that use `mpich` that is different from the one that `mpich` was built with is also easier now; see the installation manual.
- Known problems and bugs with this release are documented in the file '`mpich/KnownBugs`'.
- There is an FAQ at `http://www.mcs.anl.gov/mpi/mpich/faq.html` . See this if you get "permission denied", "connection reset by peer", or "poll: protocol failure in circuit setup" when trying to run `mpich`.
- There is a paper on jumpshot available at `ftp://ftp.mcs.anl.gov/pub/mpi/jumpshot.ps.gz` . A paper on MPD is available at `ftp://ftp.mcs.anl.gov/pub/mpd.ps.gz`.

1 Introduction

`Mpich` is a freely available implementation of the MPI standard that runs on a wide variety of systems. The details of the `mpich` implementation are described in [9]; related papers include [7] and [8]. This document assumes that `mpich` has already been installed; if not, you should first read *Installation Guide to* `mpich`, *a Portable Implementation of MPI* [6]. For concreteness, this document assumes that the `mpich` implementation is installed into '`/usr/local/mpich`' and that you have added '`/usr/local/mpich/bin`' to your path. If `mpich` is installed somewhere else, you should make the appropriate changes. If `mpich` has been built for several different architectures and/or communiation mechanisms (called *devices* in `mpich`), you must choose the directories appropriately; check with whoever installed `mpich` at your site.

Major Features of `mpich`:

- Full MPI 1.2 compliance, *including* cancel of sends.
- The MPI-2 standard C++ bindings are available for the MPI-1 functions.
- Both Fortran 77 and Fortran 90 bindings, including both '`mpif.h`' and an MPI module.
- A Windows NT version is available as open source. The installation and use for this version is different; this manual covers only the Unix version of `mpich`.
- Supports a wide variety of environments, including clusters of SMPs and massively parallel computers.
- Follows many (but not yet all) of GNU-recommended build and install targets, including VPATH.
- Parts of MPI-2 are also supported:
  - Most of MPI-IO is supported through the ROMIO implementation (See '`romio/README`' for details).
  - Support for `MPI_INIT_THREAD` (but only for `MPI_THREAD_SINGLE`).
  - Miscellaneous new `MPI_Info` and `MPI_Datatype` routines.
- `Mpich` also includes components of a parallel programming environment, including
  - Tracing and logfile tools based on the MPI profiling interface, including a scalable logfile format (SLOG).
  - Parallel performance visualization tools (`upshot` and `jumpshot`).
  - Extensive correctness and performance tests.

This document assumes that `mpich` has already been built and installed according to the instructions in the *Installation Guide to* `mpich`.

2 Linking and running programs

`Mpich` provides tools that simplify creating MPI executables. Because `mpich` programs may require special libraries and compile options, you should use the commands that `mpich` provides for compiling and linking programs. However, for special needs, Section 2.2 shows how to determine what libraries and options are needed when not using the `mpich` scripts.

2.1 Scripts to Compile and Link Applications

The `mpich` implementation provides four commands for compiling and linking C (`mpicc`), C++ (`mpiCC`), Fortran 77 (`mpif77`), and Fortran 90 (`mpif90`) programs.

In addition, the following special options are supported:

-mpilog Build version that generates MPE log files.

-mpitrace Build version that generates traces.

-mpianim Build version that generates real-time animation.

-show Show the commands that would be used without actually running them.

Use these commands just like the usual C, Fortran 77, C++, or Fortran compilers. For example,

```
mpicc -c foo.c
mpif77 -c foo.f
mpiCC -c foo.C
mpif90 -c foo.f
```

and

```
mpicc -o foo foo.o
mpif77 -o foo foo.o
mpiCC -o foo foo.o
mpif90 -o foo foo.o
```

Commands for the linker may include additional libraries. For example, to use routines from the C math library library, use

```
mpicc -o foo foo.o -lm
```

Combining compilation and linking in a single command, as shown here,

```
mpicc -o foo foo.c
mpif77 -o foo foo.f
mpiCC -o foo foo.C
mpif90 -o foo foo.f
```

may also be used.

Note that while the suffixes `.c` for C programs and `.f` for Fortran-77 programs are standard, there is no consensus for the suffixes for C++ and Fortran-90 programs. The ones shown here are accepted by many but not all systems. `Mpich` tries to determine the accepted suffixes, but may not always be able to.

You can override the choice of compiler by specifying the environment variable `MPICH_CC`, `MPICH_F77`, `MPICH_CCC`, or `MPICH_F90`. However, be warned that this will work only if the alternate compiler is compatible with the default one (by compatible, we mean that is uses the same sizes for all datatypes and layouts, and generates object code that can be used with the `mpich` libraries). If you wish to override the linker, use the environment variables `MPICH_CLINKER`, `MPICH_F77LINKER`, `MPICH_CCLINKER`, or `MPICH_F90LINKER`.

2.1.1 Using Shared Libraries

Shared libraries can help reduce the size of an executable. This is particularly valuable on clusters of workstations, where the executable must normally be copied over a network to each machine that is to execute the parallel program. However, there are some practical problems in using shared libraries; this section discusses some of them and how to solve most of those problems. Currently, shared libraries are not supported from C++.

In order to build shared libraries for `mpich`, you must have configured and built `mpich` with the `--enable-sharedlib` option. Because each Unix system and in fact each compiler uses a different and often incompatible set of options for creating shared objects and libraries, `mpich` may not be able to determine the correct options. Currently, `mpich` understands Solaris, GNU `gcc` (on most platforms, including LINUX and Solaris), and IRIX. Information on building shared libraries on other platforms should be sent to `mpi-bugs@mcs.anl.gov`.

Once the shared libraries are built, you must tell the `mpich` compilation and linking commands to use shared libraries (the reason that shared libraries are not the default will become clear below). You can do this either with the command line option `-shlib` or by setting the environment variable `MPICH_USE_SHLIB` to `yes`. For example,

```
mpicc -o cpi -shlib cpi.c
```

or

```
setenv MPICH_USE_SHLIB yes
mpicc -o cpi cpi.c
```

Using the environment variable `MPICH_USE_SHLIB` allows you to control whether shared libraries are used without changing the compilation commands; this can be very useful for projects that use makefiles.

Running a program built with shared libraries can be tricky. Some (most?) systems *do not remember* where the shared library was found when the executable was linked! Instead, they depend on finding the shared library in either a default location (such as '`/lib`') or in a

directory specified by an environment variable such as `LD_LIBRARY_PATH` or by a command line argument such as `-R` or `-rpath` (more on this below). The `mpich` configure tests for this and will report whether an executable built with shared libraries remembers the location of the libraries. It also attemps to use a compiler command line argument to force the executable to remember the location of the shared library.

If you need to set an environment variable to indicate where the `mpich` shared libraries are, you need to ensure that both the process that you run `mpirun` from and any processes that `mpirun` starts gets the enviroment variable. The easiest way to do this is to set the environment variable within your '`.cshrc`' (for csh or tcsh users) or '`.profile`' (for sh and ksh users) file.

However, setting the environment variable within your startup scripts can cause problems if you use several different systems. For example, you may have a single '`.cshrc`' file that you use with both an SGI (IRIX) and Solaris system. You do not want to set the `LD_LIBRARY_PATH` to point the SGI at the Solaris version of the `mpich` shared libraries[1]. Instead, you would like to set the environment variable before running `mpirun`:

```
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:/usr/local/mpich/lib/shared
mpirun -np 4 cpi
```

Unfortunately, this won't always work. Depending on the method that `mpirun` and `mpich` use to start the processes, the environment variable may not be sent to the new process. This will cause the program to fail with a message like

```
ld.so.1: /home/me/cpi: fatal: libpmpich.so.1.0: open failed: No such
file or directory
Killed
```

There are various solutions to this problem; each of them depends on the particular `mpich` device (e.g., `ch_p4`) that you are using, and those solutions are discussed in the appropriate section within Section 3.

An alternative to using `LD_LIBRARY_PATH` and the secure server is to add an option to the link command that provides the path to use in searching for shared libraries. Unfortunately, the option that you would like is "append this directory to the search path" (such as you get with `-L`). Instead, many compilers provide only "replace the search path with this path."[2] For example, some compilers allow `-Rpath:path:...:path` to specify a replacement path. Thus, if both `mpich` and the user provide library search paths with `-R`, one of the search paths will be lost. Eventually, `mpicc` and friends can check for `-R` options and create a unified version, but they currently do not do this. You can, however, provide a complete search path yourself if your compiler supports an option such as `-R`.

The preceeding may sound like a lot of effort to go to, and in some ways it is. For large clusters, however, the effort will be worth it: programs will start faster and more reliably, because there is less network and file system traffic.

[1] You can make '`.cshrc`' check for the kind of system that you are running on and pick the paths appropriately. This isn't as flexible as the approach of setting the environment variable from the running shell.

[2] Even though the linker may provide the "append to search path" form.

2.1.2 Fortran 90 and the MPI module

When `mpich` was configured, the installation process normally looks for a Fortran 90 compiler, and if it finds one, builds two different versions of an MPI module. One module includes only the MPI routines that do not take "choice" arguments; the other includes all MPI routines. A choice argument is an argument that can take any datatype; typically, these are the buffers in MPI communication routines such as `MPI_Send` and `MPI_Recv`. The two different modules can be accessed with the `-nochoice` and `-choice` option to `mpif90` respectively. The choice version of the module supports a limited set of datatypes (numeric scalars and numeric one- and two-dimensional arrays). This is an experimental feature; please send mail to `mpi-bugs@mcs.anl.gov` if you have any trouble. Neither of these modules offer full "extended Fortran support" as defined in the MPI-2 standard.

2.2 Compiling and Linking without the Scripts

In some cases, it is not possible to use the scripts supplied by `mpich` for compiling and linking programs. For example, another tool may have its own compilation scripts. In this case, you can use `-compile_info` and `-link_info` to have the `mpich` compilation scripts indicate the compiler flags and linking libraries that are required for correct operation of the `mpich` routines. For example, when using the `ch_shmem` device on Solaris systems, the library `thread` (`-lthread`) must be linked with the application. If the `thread` library is not provided, the application will still link, but essential routines will be replaced with dummy versions contained within the Solaris C library, causing the application to fail.

For example, to determine the flags used to compile and link C programs, you can use these commands, whose output for the `ch_p4` device on a Linux workstation is shown.

```
% mpicc -compile_info
cc -DUSE_STDARG -DHAVE_STDLIB_H=1 -DHAVE_STRING_H=1 -DHAVE_UNISTD_H=1
-DHAVE_STDARG_H=1 -DUSE_STDARG=1 -DMALLOC_RET_VOID=1
-I/usr/local/mpich/include -c

% mpicc -link_info
cc -DUSE_STDARG -DHAVE_STDLIB_H=1 -DHAVE_STRING_H=1 -DHAVE_UNISTD_H=1
-DHAVE_STDARG_H=1 -DUSE_STDARG=1 -DMALLOC_RET_VOID=1
-L/usr/local/mpich/lib -lmpich
```

2.3 Running with mpirun

To run an MPI program, use the `mpirun` command, which is located in '`/usr/local/mpich/bin`'. For almost all systems, you can use the command

```
mpirun -np 4 a.out
```

to run the program '`a.out`' on four processors. The command `mpirun -help` gives you a complete list of options, which may also be found in Appendix B.

On exit, `mpirun` returns the status of one of the processes, usually the process with rank zero in `MPI_COMM_WORLD`.

3 Special features of different systems

MPI makes it relatively easy to write portable parallel programs. However, one thing that MPI does not standardize is the environment within which the parallel program is running. There are three basic types of parallel environments: parallel computers, clusters of workstations, and integrated distributed environments, which we will call "computational grids," that include parallel computers and workstations and that may span multiple sites. Naturally, a parallel computer (usually) provides an integrated, relatively easy way of running parallel programs. Clusters of workstations and grid environments, on the other hand, usually have no standard way of running a parallel program and will require some additional setup. The `mpich` implementation is designed to hide these differences behind the `mpirun` script; however, if you need special features or options or if you are having problems running your programs, you will need to understand the differences between these systems. In the following, we describe the special features that apply for workstation clusters, grids (as supported by the `globus2` device), and certain parallel computers.

When coupling multiple multicomputers, the `globus2` device, described in Section 3.6, may be a better choice than the `ch_p4` device.

3.1 MPMD Programs

It is possible to run a parallel program with different executables with several of the device including the `ch_p4`, `ch_mpl`, and `globus2` devices. This style of parallel programming is often called MPMD for "multiple program multiple data". In many cases, it is easy to convert a MPMD program into a single program that uses the rank of the process to invoke a different routine; doing so makes it easier to start parallel programs and often to debug them. If converting a MPMD program to a SPMD (single program multiple data, not to be confused with single *instruction* multiple data, or SIMD) is not feasible, then you can run MPMD programs using `mpich`. However, you will not be able to use `mpirun` to start the programs; instead, you will need to follow the instructions for each device. For the `globus2` device, see Section 3.6. For the `ch_p4` device, see Section 3.2.6 and look for a discussion of procgroup files. For the `ch_mpl` device, you will need to check the POE documentation for your site for details on running MPMD programs.

3.2 Workstation clusters and the `ch_p4` device

Most massively parallel processors (MPPs) provide a way to start a program on a requested number of processors; `mpirun` makes use of the appropriate command whenever possible. In contrast, workstation clusters require that each process in a parallel job be started individually, though programs to help start these processes exist (see Section 3.2.3 below). Because workstation clusters are not already organized as an MPP, additional information is required to make use of them. `Mpich` should be installed with a list of participating

workstations in the file '`machines.<arch>`' in the directory '`/usr/local/mpich/share`'. This file is used by `mpirun` to choose processors to run on (using heterogeneous clusters is discussed in Section 3.2.6). The rest of this section discusses some of the details of this process, and how you can check for problems.

3.2.1 Checking your machines list

Use the script '`tstmachines`' in '`/usr/local/mpich/sbin`' to ensure that you can use all of the machines that you have listed. This script performs an `rsh` and a short directory listing; this tests that you both have access to the node and that a program in the current directory is visible on the remote node. If there are any problems, they will be listed. These problems *must* be fixed before proceeding.

The only argument to `tstmachines` is the name of the architecture; this is the same name as the extension on the machines file. For example,

```
/usr/local/mpich/sbin/tstmachines sun4
```

tests that a program in the current directory can be executed by all of the machines in the `sun4` machines list. This program is silent if all is well; if you want to see what it is doing, use the `-v` (for verbose) argument:

```
/usr/local/mpich/sbin/tstmachines -v sun4
```

The output from this command might look like

```
Trying true on host1.uoffoo.edu ...
Trying true on host2.uoffoo.edu ...
Trying ls on host1.uoffoo.edu ...
Trying ls on host2.uoffoo.edu ...
Trying user program on host1.uoffoo.edu ...
Trying user program on host2.uoffoo.edu ...
```

If `tstmachines` finds a problem, it will suggest possible reasons and solutions. In brief, there are three tests:

1. Can processes be started on remote machines? `tstmachines` attempts to run the shell command `true` on each machine in the '`machines`' files by using the remote shell command. Note that the `ch_p4` devices does *not* require a remote shell command and can use alternative methods (see Section 3.2.3 and 3.2.2).

2. Is current working directory available to all machines? This attempts to `ls` a file that `tstmachines` creates by running `ls` using the remote shell command. Note that `ch_p4` does *not* require that all processors have access to the same file system (see Section 3.2.6), but the `mpirun` command does require this.

3. Can user programs be run on remote systems? This checks that shared libraries and other components have been properly installed on all machines.

3.2.2 Using the Secure Shell

The *Installation Guide* explains how to set up your environment so that the `ch_p4` device on networks will use the secure shell `ssh` instead of `rsh`. This is useful on networks where for security reasons the use of `rsh` is discouraged or disallowed.

3.2.3 Using the Secure Server

Because each workstation in a cluster (usually) requires that a new user log into it, and because this process can be very time-consuming, `mpich` provides a program that may be used to speed this process. This is the *secure server*, and is located in '`serv_p4`' in the directory '`/usr/local/mpich/bin`'. The script '`chp4_servs`' in the same directory may be used to start '`serv_p4`' on those workstations that you can `rsh` programs on. You can also start the server by hand and allow it to run in the background; this is appropriate on machines that do not accept `rsh` connections but on which you have accounts.

Before you start this server, check to see if the secure server has been installed for general use; if so, the same server can be used by everyone. In this mode, root access is required to install the server. If the server has not been installed, then you can install it for your own use without needing any special privileges with

```
chp4_servs -port=1234
```

This starts the secure server on all of the machines listed in the file '`/usr/local/mpich/share/machines.<arch>`'.

The port number, provided with the option `-port=`, must be different from any other port in use on the workstations.

To make use of the secure server for the `ch_p4` device, add the following definitions to your environment:

```
setenv MPI_USEP4SSPORT yes
setenv MPI_P4SSPORT 1234
```

The value of `MPI_P4SSPORT` must be the port with which you started the secure server. When these environment variables are set, `mpirun` attempts to use the secure server to start programs that use the `ch_p4` device. (The command line argument `-p4ssport` to `mpirun` may be used instead of these environment variables; `mpirun -help` will give you more information.)

3.2.4 SMP Clusters

When using a cluster of symmetric multiprocessors (SMPs) (with the `ch_p4` device configured with `-comm=shared`), you can control the number of processes that communicate with shared memory on each SMP node. First, you need to modify the machines file (see Section 3.2) to indicate the number of processes that should be started on

each host. Normally this number should be no greater than the number of processors; on SMPs with large numbers of processors, the number should be one less than the number of processors in order to leave one processor for the operating system. The format is simple: each line of the machines file specifies a hostname, optionally followed by a colon (:) and the number of processes to allow. For example, the file containing the lines

```
mercury
venus
earth
mars:2
jupiter:15
```

specifies three single processor machines (`mercury`, `venus`, and `earth`), a 2 processor machine (`mars`), and a 15 processor machine (`jupiter`).

By default, `mpirun` will use at most the number of processors specified in the machines list for each node, upto 16 processes on each machine. By setting the environment variable `MPI_MAX_CLUSTER_SIZE` to a positive integer value, `mpirun` will use upto that many processes, sharing memory for communication, on a host. For example, if `MPI_MAX_CLUSTER_SIZE` had the value 4, then `mpirun -np 9` with the above machine file create one process on each of `mercury`, `venus`, and `earth`, 2 on `mars` (2 because the machines file specifies that `mars` may have 2 processes sharing memory) and 4 on `jupiter` (because `jupiter` may have 15 processes and only 4 more are needed). If 10 processes were needed, `mpirun` would start over from the beginning of the machines file, creating an additional process on `mercury`; the value of `MPI_MAX_CLUSTER_SIZE` prevents `mpirun` from starting a fifth process sharing memory on `jupiter`.

3.2.5 Heterogeneous networks and the ch_p4 device

A heterogeneous network of workstations is one in which the machines connected by the network have different architectures and/or operating systems. For example, a network may contain 3 Sun SPARC (sun4) workstations and 3 SGI IRIX workstations, all of which communicate via the TCP/IP protocol. The `mpirun` command may be told to use all of these by using multiple `-arch` and `-np` arguments. For example, to run a program on 3 sun4s and 2 SGI IRIX workstations, use

```
mpirun -arch sun4 -np 3 -arch IRIX -np 2 program.%a
```

The special program name `program.%a` allows you to specify the different executables for the program, since a Sun executable won't run on an SGI workstation and vice versa. The `%a` is replaced with the architecture name; in this example, `program.sun4` runs on the Suns and `program.IRIX` runs on the SGI IRIX workstations. You can also put the programs into different directories; for example,

```
mpirun -arch sun4 -np 3 -arch IRIX -np 2 /tmp/%a/program
```

It is important to specify the architecture with `-arch` *before* specifying the number of processors. Also, the *first* `arch` command must refer to the processor on which the job will

be started. Specifically, if `-nolocal` is not specified, then the first `-arch` must refer to the processor from which mpirun is running.

3.2.6 The P4 Procgroup File

For even more control over how jobs get started, we need to look at how `mpirun` starts a parallel program on a workstation cluster. Each time `mpirun` runs, it constructs and uses a new file of machine names for just that run, using the `machines` file as input. (The new file is called `PIyyyy`, where `yyyy` is the process identifier.) If you specify `-keep_pg` on your `mpirun` invocation, you can use this information to see where `mpirun` ran your last few jobs. You can construct this file yourself and specify it as an argument to `mpirun`. To do this for `ch_p4`, use

```
mpirun -p4pg pgfile myprog
```

where `pfile` is the name of the file. The file format is defined below.

This is necessary when you want closer control over the hosts you run on, or when `mpirun` cannot construct it automatically. Such is the case when

- You want to run different executables on different hosts (your program is not SPMD).
- You want to run on a network of shared-memory multiprocessors *and* need to specify the number of processes that will share memory on each machine.

The format of a `ch_p4` procgroup file is a set of lines of the form

```
<hostname>  <#procs>  <progname>  [<login>]
```

An example of such a file, where the command is being issued from host `sun1`, might be

```
sun1   0  /users/jones/myprog
sun2   1  /users/jones/myprog
sun3   1  /users/jones/myprog
hp1    1  /home/mbj/myprog    mbj
```

The above file specifies four processes, one on each of three suns and one on another workstation where the user's account name is different. Note the `0` in the first line. It is there to indicate that no *other* processes are to be started on host `sun1` than the one started by the user by his command.

You might want to run all the processes on your own machine, as a test. You can do this by repeating its name in the file:

```
sun1 0 /users/jones/myprog
sun1 1 /users/jones/myprog
sun1 1 /users/jones/myprog
```

This will run three processes on `sun1`, communicating via sockets.

To run on a shared-memory multiprocessor, with 10 processes, you would use a file like:

```
sgimp  9  /u/me/prog
```

Note that this is for 10 processes, one of them started by the user directly, and the other nine specified in this file. This requires that `mpich` was configured with the option `-comm=shared`; see the installation manual for more information.

If you are logged into host `gyrfalcon` and want to start a job with one process on `gyrfalcon` and three processes on `alaska`, where the `alaska` processes communicate through shared memory, you would use

```
local    0  /home/jbg/main
alaska   3  /afs/u/graphics
```

It is not possible to provide different command line argument to different MPI processes.

3.2.7 Tuning P4 Performance

There are several enviroment variables and command line options that can be used to tune the performance of the `ch_p4` device. Note that these environment variables must be defined for all processes that are created, not just the process that you are launching MPI programs from (i.e., setting these variables should be part of your '`.login`' or '`.cshrc`' startup files). The environment variables are:

`P4_SOCKBUFSIZE`. Specifies the socket buffer size in bytes. Increasing this value can improve performance on some system.

`P4_WINSHIFT`. This is another socket parameter that is supported on only a few platforms. We recommend leaving it alone.

`P4_GLOBMEMSIZE`. This is the amount of memory in bytes reserved for communication with shared memory (when `mpich` is configured with `-comm=shared`). Increase this if you get error messages about `p4_shmalloc` returning `NULL`.

TCP Tuning. The command line option `-p4sctrl` takes a parameter that specifies various socket options. These are in the form of `name=value`, separated by colons. With the exception of `bufsize`, users should normally not change the defaults for these. The names and their meanings are

bufsize Socket buffer size, in 1K (1024) bytes. E.g., `bufsize=32` requests 32K byte socket buffers. The default is 16.

winsize Winshift size. Available only on systems that define `TCP_WINSHFT` and is ignored otherwise.

netsendw Use `select` to wait for `write` to proceed. Values are `y` (default) or `n`.

netreadw Use `select` to wait for `read` to proceed. Values are `y` (default) or `n`.

writev Use `writev` to send the header (MPI envelope) and data in a single message. Values are `y` (default) or `n`.

readb Switch a socket to blocking mode while wait to read instead of busy-waiting or using `select`. Values are `y` or `n` (default).

stat Print out statistics on `write` and `read` operations. Useful only for experts!

For example, to use 64K socket buffers and turn off the use of `writev`, you could use

```
mpirun -np 2 mpptest -p4sctrl bufsize=64:writev=n
```

3.2.8 Using special interconnects

In some installations, certain hosts can be connected in multiple ways. For example, the "normal" Ethernet may be supplemented by a high-speed FDDI ring. Usually, alternate host names are used to identify the high-speed connection. All you need to do is put these alternate names in your '`machines.xxxx`' file. In this case, it is important not to use the form `local 0` but to use the name of the local host. For example, if hosts `host1` and `host2` have ATM connected to `host1-atm` and `host2-atm` respectively, the correct `ch_p4` procgroup file to connect them (running the program '`/home/me/a.out`') is

```
host1-atm 0 /home/me/a.out
host2-atm 1 /home/me/a.out
```

3.2.9 Using Shared Libraries with the `ch_p4` device

As described at the end of Section 2.1.1, it is sometime necessary to ensure that environment variables have been communicated to the remote machines before the program that makes use of shared libraries starts. The various remote shell commands (e.g., `rsh` and `ssh`) do not do this. Fortunately, the secure server (Section 3.2.3) does communicate the environment variables. This server is built and installed as part of the `ch_p4` device, and can be installed on all machines in the machines file for the current architecture (assuming that there is a working remote shell command) with

```
chp4_servs -port=1234
```

The secure server propagates *all* environment variables to the remote process, and ensures that the environment in which that process (containing your MPI program) contains all environment variables that start with `LD_` (just in case the system uses `LD_SEARCH_PATH` or some other name for finding shared libraries).

3.3 Fast Startup with the Multipurpose Daemon and the ch_p4mpd Device

This device is experimental, and its version of mpirun is a little different from that for the other devices. In this section we describe how the mpd system of daemons works and how to run MPI programs using it. To use this system, mpich must have been configured with the ch_p4mpd device, and the daemons must have been started on the machines where you will be running. This section describes how to do these things.

3.3.1 Goals

The goal of the multipurpose daemon (mpd and the associated ch_p4mpd device) is to make mpirun behave like a single program even as it starts multiple processes to execute an MPI job. We will refer to the *mpirun* process and the *MPI* processes. Such behavior includes

- fast, scalable startup of MPI (and even non-MPI) processes. For those accustomed to using the ch_p4 device on TCP networks, this will be the most immediately noticeable change. Job startup is now much faster.
- collection of stdout and stderr from the MPI processes to the stdout and stderr of the mpirun process.
- delivery of mpirun's stdin to the stdin of MPI process 0.
- delivery of signals from the mpirun process to the MPI processes. This means that it is easy to kill, suspend, and resume your parallel job just as if it were a single process, with cntl-C, cntl-Z, and bg and fg commands.
- delivery of command-line arguments to all MPI processes.
- copying of the PATH environment from the environment in which mpirun is executed to the environments in which the MPI processes are executed.
- use of an optional argument to provide other environment variables.
- use of a further optional argument to specify where the MPI processes will run (see below).

3.3.2 Introduction

The ch_p4 device relies by default on rsh for process startup on remote machines. The need for authentication at job startup time, combined with the sequential process by which contact information is collected from each remote machine and broadcast back to all machines, makes job startup unscalably slow, especially for large numbers of processes.

With version 1.2.0 of mpich, we introduced a new method of process startup based on daemons. This mechanism, which requires configuration with a new device, has not yet been widely enough tested to become the default for clusters, but we anticipate that it eventually will become so. In the current version of mpich, it has been significantly enhanced, and

will now be installed when `mpich` is installed with `make install`. On systems with `gdb`, it supports a simple parallel debugger we call `mpigdb`.

The basic idea is to establish, ahead of job-startup time, a network of daemons on the machines where MPI processes will run, and also on the machine on which `mpirun` will be executed. Then job startup commands (and other commands) will contact the local daemon and use the pre-existing daemons to start processes. Much of the initial synchronization done by the `ch_p4` device is eliminated, since the daemons can be used at run time to aid in establishing communication between processes.

To use the new startup mechanism, you must

- configure with the new device:

  ```
  configure --with-device=ch_p4mpd
  ```

- make as usual:

  ```
  make
  ```

- go to the '`mpich/mpid/mpd`' directory, where the daemons code is located and the daemons are built, or else put this directory in your `PATH`.

- start the daemons:

  The daemons can be started by hand on the remote machines using the port numbers advertised by the daemons as they come up:

  - On fire:

    ```
    fire% mpd &
    [2] 23792
    [fire_55681]: MPD starting
    fire%
    ```

  - On soot:

    ```
    soot% mpd -h fire -p 55681 &
    [1] 6629
    [soot_35836]: MPD starting
    soot%
    ```

  The `mpd`'s are identified by a host and port number.

  If the daemons do not advertise themselves, one can find the host and port by using the `mpdtrace` command:

  - On fire:

    ```
    fire% mpd &
    fire% mpdtrace
    mpdtrace: fire_55681:  lhs=fire_55681  rhs=fire_55681  rhs2=fire_55681
    fire%
    ```

  - On soot:

```
soot% mpd -h fire -p 55681 &
soot% mpdtrace
mpdtrace: fire_55681:  lhs=soot_33239  rhs=soot_33239  rhs2=fire_55681
mpdtrace: soot_33239:  lhs=fire_55681  rhs=fire_55681  rhs2=soot_33239
soot%
```

What `mpidtrace` is showing is the ring of mpd's, by hostname and port that can be used to introduce another mpd into the ring. The left and right neighbor of each mpd in the ring is shown as `lhs` and `rhs` respectively. `rhs2` shows the daemon two steps away to the right (which in this case is the daemon itself).

You can also use `mpd -b` to start the daemons as real daemons, disconnected from any terminal. This has advantages and disadvantages.

There is also a pair of scripts in the '`mpich/mpid/mpd`' directory that can help:

```
localmpds <number>
```

will start `<number>` mpds on the local machine. This is only really useful for testing. Usually you would do

```
mpd &
```

to start one mpd on the local machine. Then other mpd's can be started on remote machines via `rsh`, if that is available:

```
remotempds <hostfile>
```

where `<hostfile>` contains the names of the other machines to start the mpd's on. It is a simple list of hostnames only, unlike the format of the `MACHINES` files used by the `ch_p4` device, which can contain comments and other symbols.

See also the `startdaemons` script, which will be installed when `mpich` is installed.

- Finally, start jobs with the `mpirun` command as usual:

```
mpirun -np 4 a.out
```

You can kill the daemons with the `mpichstop` command.

3.3.3 Examples

Here are a few examples of usage of the `mpirun` that is built when the `mpich` is configured and built with the `ch_p4mpd` device.

- Run the `cpi` example

```
mpirun -np 16 cpi
```

- You can get line labels on `stdout` and `stderr` from your program by including the `-l` option. Output lines will be labeled by process rank.

- Run the `fpi` program, which prompts for a number of intervals to use.

  ```
  mpirun -np 32 fpi
  ```

  The streams `stdin`, `stdout`, and `stderr` will be mapped back to your `mpirun` process, even if the MPI process with rank 0 is executed on a remote machine.

- Use arguments and environment variables.

  ```
  mpirun -np 32 myprog arg1 arg2 -MPDENV- MPE_LOG_FORMAT=SLOG \
                GLOBMEMSIZE=16000000
  ```

  The argument `-MPDENV-` is a *fence*. All arguments after it are handled by `mpirun` rather than the application program.

- Specify where the first process is to run. By default, MPI processes are spawned by by consecutive `mpd`'s in the rung, starting with the one *after* the local one (the one running on the same machine as the `mpirun` process. Thus if you are logged into `dion` and there are `mpd`'s running `dion` and on `belmont1`, `belmont2`, ..., `belmont64`, and you type

  ```
  mpirun -np 32 cpi
  ```

  your processes will run on `belmont1`, `belmont2`, ..., `belmont32`. You can force your MPI processes to start elsewhere by giving `mpirun` optional location arguments. If you type

  ```
  mpirun -np 32 cpi -MPDLOC- belmont33 belmont34 ... belmont64
  ```

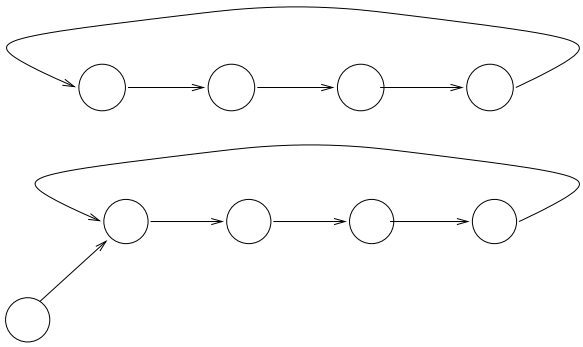
  then your job will run on `belmont33`, `belmont34`, ..., `belmont64`. In general, processes will only be run on machines in the list of machines after `-MPDLOC-`.

  This provides an extremely preliminary and crude way for `mpirun` to choose locations for MPI processes. In the long run we intend to use the `mpd` project as an environment for exploring the interfaces among job schedules, process managers, parallel application programs (particularly in the dynamic environment of MPI-2), and user commands.

- Find out what hosts your `mpd`'s are running on:

  ```
  mpirun -np 32 hostname | sort | uniq
  ```

  This will run 32 instances of `hostname` assuming '`/bin`' is in your path, regardless of how many `mpd`'s there are. The other processes will be wrapped around the ring of `mpd`'s.



3.3.4 How the Daemons Work

Once the daemons are started they are connected in a ring: A "console" process (`mpirun`, `mpdtrace`, `mpdallexit`, etc.) can connect to any mpd, which it does by using a Unix named socket set up in '`/tmp`' by the local mpd. If it is an `mpirun` process, it requests that a number of processes be started, starting at the machine given by `-MPDLOC-` as described above. The location defaults to the mpd `next` in the ring after the one contacted by the console. Then the following events take place.

- The mpd's fork that number of `manager` processes (the executable is called `mpdman` and is located in the '`mpich/mpid/mpd`' directory). The managers are forked consecutively by the mpd's around the ring, wrapping around if necessary.
- The managers form themselves into a ring, and fork the application processes, called *clients*.
- The console disconnects from the mpd and reconnects to the first manager. `stdin` from `mpirun` is delivered to the client of manager 0.
- The managers intercept standard I/O fro the clients, and deliver command-line arguments and the environment variables that were specified on the `mpirun` command. The sockets carrying `stdout` and `sdterr` form a tree with manager 0 at the root.

At this point the situation looks something like Figure 1. When the clients need to contact each other, they use the managers to find the appropriate process on the destination host. The `mpirun` process can be suspended, in which case it and the clients are suspended, but the mpd's and managers remain executing, so that they can unsuspend the clients when `mpirun` is unsuspended. Killing the `mpirun` process kills the clients and managers.

The same ring of mpd's can be used to run multiple jobs from multiple consoles at the same time. Under ordinary circumstances, there still needs to be a separate ring of mpd's for each user. For security purposes, each user needs to have a '`.mpdpasswd`' file in the user's home directory, readable only by the user, containing a password. This file is read when the `mpd` is started. Only `mpd`'s that know this password can enter a ring of existing `mpd`'s.

A new feature is the ability to configure the mpd system so that the daemons can be run as root. To do this, after configuring `mpich` you need to reconfigure in the '`mpid/mpd`'

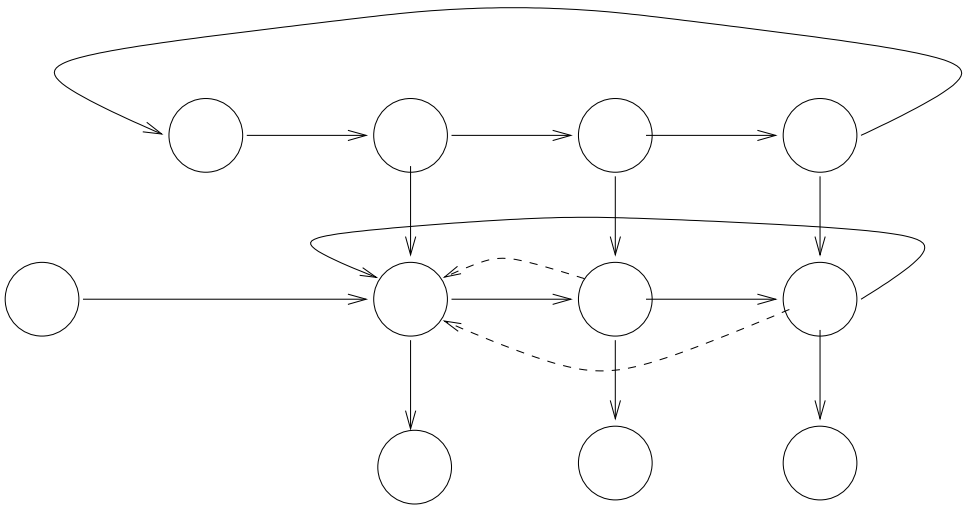


Figure 1: Mpds with console, managers, and clients

directory with `--enable-root` and remake. Then `mpirun` should be installed as a `setuid` program. Multiple users can use the same set of `mpd`'s, which are run as root, although their `mpirun`, managers, and clients will be run as the user who invoked `mpirun`.

3.3.5 Running MPICH Jobs under MPD

Because the MPD daemons are already in communication with one another before the job starts, job startup is much faster than with the `ch_p4` device. The `mpirun` command for the `ch_p4mpd` device has a number of special command-line arguments. If you type `mpirun` with no arguments, they are displayed:

```
% mpirun
Usage: mpirun <args> executable <args_to_executable>
Arguments are:
    -np num_processes_to_run  (required as first two args)
    [-s]  (close stdin; can run in bkgd w/o tty input problems)
    [-g group_size]  (start group_size processes per mpd)
    [-m machine_file]  (filename for allowed machines)
    [-l]  (line labels; unique id for each process' output
    [-1]  (do NOT start first process locally)
    [-y]  (run as Myrinet job)
```

The `-1` option allows you, for example, to run `mpirun` on a "login" or "development" node on your cluster but to start all the application processes on "computation" nodes.

The program `mpirun` runs in a separate (non-MPI) process that starts the MPI processes running the specified executable. It serves as a single-process representative of the parallel MPI processes in that signals sent to it, such as `^Z` and `^C` are conveyed by the MPD system to all the processes. The output streams `stdout` and `stderr` from the MPI processes are routed back to the `stdout` and `stderr` of `mpirun`. As in most MPI implementations,

`mpirun`'s `stdin` is routed to the `stdin` of the MPI process with rank 0.

3.4 Debugging MPI Programs

Debugging parallel programs is notoriously difficult. Parallel programs are subject not only to the usual kinds of bugs but also to new kinds having to do with timing and synchronization errors. Often, the program "hangs," for example when a process is waiting for a message to arrive that is never sent or is sent with the wrong tag. Parallel bugs often disappear precisely when you adds code to try to identify the bug, which is particularly frustrating. In this section we discuss three approaches to parallel debugging.

3.4.1 The `printf` Approach.

Just as in sequential debugging, you often wish to trace interesting events in the program by printing trace messages. Usually you wish to identify a message by the rank of the process emitting it. This can be done explicitly by putting the rank in the trace message. As noted above, using the "line labels" option (`-l`) with `mpirun` in the `ch_p4mpd` device in MPICH adds the rank automatically.

3.4.2 Using a Commercial Debugger.

The TotalView© debugger from Etnus, Ltd. [1] runs on a variety of platforms and interacts with many vendor implementations of MPI, including MPICH on Linux clusters. For the `ch_p4` device you invoke TotalView with

```
mpirun -tv <other arguments>
```

and with the `ch_p4mpd` device you use

```
totalview mpirun <other arguments>
```

That is, again `mpirun` represents the parallel job as a whole. TotalView has special commands to display the message queues of an MPI process. It is possible to attach TotalView to a collection of processes that are already running in parallel; it is also possible to attach to just one of those processes.

3.4.3 Using `mpigdb`.

The `ch_p4mpd` device version of MPICH features a "parallel debugger" that consists simply of multiple copies of the `gdb` debugger, together with a mechanism for redirecting `stdin`. The `mpigdb` command is a version of `mpirun` that runs each user process under the control of `gdb` and also takes control of `stdin` for `gdb`. The '`z`' command allows you to direct terminal input to any specified process or to broadcast it to all processes. We demonstrate this by running the π example under this simple debugger.

```
donner% mpigdb -np 5 cpi                # default is stdin bcast
(mpigdb) b 29                           # set breakpoint for all
0-4: Breakpoint 1 at 0x8049e93: file cpi.c, line 29.
(mpigdb) r                              # run all
0-4: Starting program: /home/lusk/mpich/examples/basic/cpi
0: Breakpoint 1, main (argc=1, argv=0xbffffa84) at cpi.c:29
1-4: Breakpoint 1, main (argc=1, argv=0xbffffa74) at cpi.c:29
0-4: 29     n = 0;                      # all reach breakpoint
(mpigdb) n                              # single step all
0: 38               if (n==0) n=100; else n=0;
1-4: 42         MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
(mpigdb) z 0                            # limit stdin to rank 0
(mpigdb) n                              # single step process 0
0: 40               startwtime = MPI_Wtime();
(mpigdb) n                              # until  caught up
0: 42           MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
(mpigdb) z                              # go back to bcast stdin
(mpigdb) n                              # single step all
                ...                     # until interesting spot
(mpigdb) n
0-4: 52                 x = h * ((double)i - 0.5);
(mpigdb) p x                            # bcast print command
0: $1 = 0.0050000000000000001           # 0's value of x
1: $1 = 0.014999999999999999            # 1's value of x
2: $1 = 0.025000000000000001            # 2's value of x
3: $1 = 0.035000000000000003            # 3's value of x
4: $1 = 0.044999999999999998            # 4's value of x
(mpigdb) c                              # continue all
0: pi is approximately 3.141600986923, Error is 0.000008333333
0-4: Program exited normally.
(mpigdb) q                              # quit
donner%
```

If the debugging process hangs (no `mpigdb` prompt) because the current process is waiting for action by another process, `ctl-C` will bring up a menu that allows you to switch processes. The `mpigdb` is not nearly as advanced as TotalView, but it is often useful, and it is freely distributed with MPICH.

3.5 Symmetric Multiprocessors (SMPs) and the `ch_shmem` device

On many of the shared-memory implementations, `mpich` reserves some shared memory in which messages are transferred back and forth. By default, `mpich` reserves roughly four MBytes of shared memory. You can change this with the environment variable `MPI_GLOBMEMSIZE`. For example, to make it 8 MB, enter

```
setenv MPI_GLOBMEMSIZE 8388608
```

Large messages are transfered in fragments, so `MPI_GLOBMEMSIZE` does not limit the maximum message size but increasing it may improve performance. Also note that systems may limit the amount of shared memory available.

By default, `mpich` limits the number of processes for the `ch_shmem` device to 32, unless it determines at configure time that the machine has more processors. You can override this limit by setting the environment variable `PROCESSOR_COUNT` to the maximum number of processes that you will want to run, and then reconfigure and remake `mpich`.

3.6 Computational Grids: the `globus2` device

The `globus2` device[3] supports the execution of MPI programs on "computational grids" that may include parallel computers and workstations, and that may span multiple sites. In such grid environments, different sites may support different security mechanisms and different process creation mechanisms. The `globus2` device hides these low-level details from you, allowing you to start programs with `mpirun` as on MPPs and workstation clusters. The `globus2` device also provides other convenient features, such as remote access to files and executable staging. These features are provided by using services supplied by the Globus toolkit: see `http://www.globus.org` for details.

The `globus2` device requires that special servers be running on the computers where processes are to be created. In our discussion of how to use the `globus2` device, we assume that we are using the `globus2` device on a collection of machines on which various Globus servers are installed and running. Such a collection is often referred to as a *computational grid*, for example, NASA's Information Power Grid (IPG). If possible, we recommend that you use the `globus2` device in this environment. If you wish to use the `globus2` device in other situations, please send email to `developers@globus.org`. Details of how to run MPI programs using the `globus2` device on Globus-enabled computational grids are in Appendix C.

3.7 MPPs

Each MPP is slightly different, and even systems from the same vendor may have different ways for running jobs at different installations. The `mpirun` program attempts to adapt to this, but you may find that it does not handle your installation. One step that you can take is to use the `-show` or `-t` (for test) option to `mpirun`. This shows how `mpirun` would start your MPI program without actually doing so. Often, you can use this information, along with the instructions for starting programs at your site, to discover how to start the program. Please let us know (`mpi-bugs@mcs.anl.gov`) about any special needs.

IBM SP. Using `mpirun` with the IBM SP computers can be tricky, because there are so many different (and often mutually exclusive) ways of running programs on them. The `mpirun` distributed with `mpich` works on systems using the Argonne scheduler (sometimes called EASY) and with systems using the default resource manager values (i.e., those not requiring the user to choose an `RMPOOL`). If you have trouble running an `mpich` program,

[3] `globus2` replaces the `globus` device distributed in previous releases of `mpich`.

try following the rules at your installation for running an MPL or POE program (if using the `ch_mpl` device) or for running p4 (if using the `ch_p4` device).

4 Sample MPI programs

The `mpich` distribution contains a variety of sample programs, which are located in the `mpich` source tree. Most of these will work with any MPI implementation, not just `mpich`.

examples/basic contains a few short programs in Fortran, C, and C++ for testing the simplest features of MPI.

examples/test contains multiple test directories for the various parts of MPI. Enter "make testing" in this directory to run our suite of function tests.

examples/perftest Performance benchmarking programs. See the script `runmpptest` for information on how to run the benchmarks. These are relatively sophisticated.

mpe/contrib/mandel A Mandelbrot program that uses the MPE graphics package that comes with mpich. It should work with any other MPI implementation as well, but we have not tested it. This is a good demo program if you have a fast X server and not too many processes.

mpe/contrib/mastermind A program for solving the Mastermind puzzle in parallel. It can use graphics (`gmm`) or not (`mm`).

Additional examples from the book *Using MPI* [10] are available at `www.mcs.anl.gov/mpi/using`. Tutorial material on MPI can also be found at `www.mcs.anl.gov/mpi`.

5 The MPE library of useful extensions

A more up to date version of this material can be found in the *User's Guide for MPE* and in the file '`mpe/README`'.

It is anticipated that `mpich` will continue to accumulate extension routines of various kinds. We keep them in a library we call `mpe`, for MultiProcessing Environment.

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.

5.1 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 5.5.3.

5.2 Logfile Format

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 "external32"). 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.

5.3 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.

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.

5.4 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.

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.

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.)

5.5 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.

5.5.1 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.

5.5.2 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 5.6.1 and 5.6.2.

5.5.3 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[4] 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.

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

If the MPE logging library, '`liblmpe.a`', is not linked with the user program, `MPE_Init_log` and `MPE_Finish_log` must be called before and 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.

5.5.4 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 that represent message traffic as the program runs.

5.6 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`.

5.6.1 Upshot and Nupshot

One tool that we use is called `upshot`, which is a derivative of Upshot [13], written in Tcl/Tk. A screen dump of Upshot in use is shown in Figure 2. It shows parallel time lines with process states, like one of the paraGraph [12]. The view can be zoomed in or out, horizontally or vertically, centered on any point in the display chosen with the mouse. In Figure 2, 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. Because of this limitation, `Nupshot` is not built by default in current MPE.

5.6.2 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 3. For `Jumpshot-2`, see [17] 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

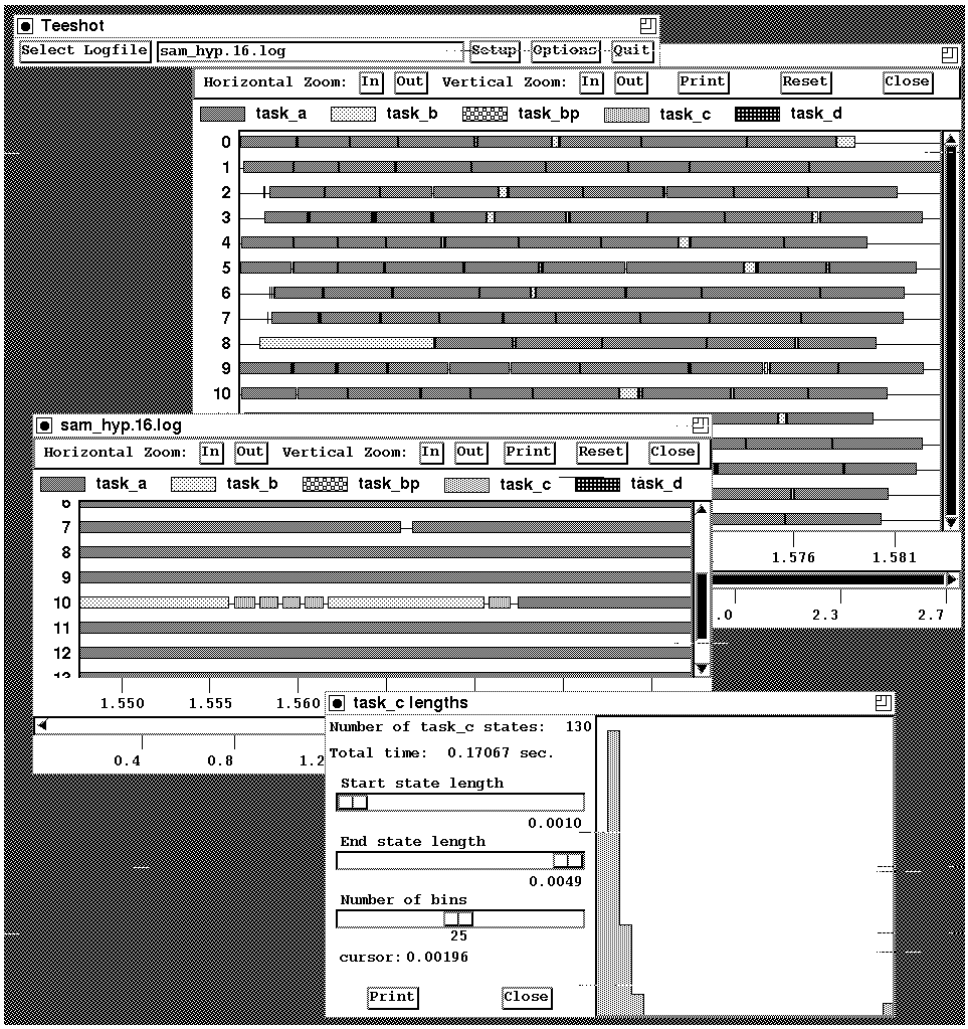


Figure 2: A screendump from `upshot`

this effort are SLOG which supports scalable logging of data and `Jumpshot-3` which reads SLOG.

5.7 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;
```

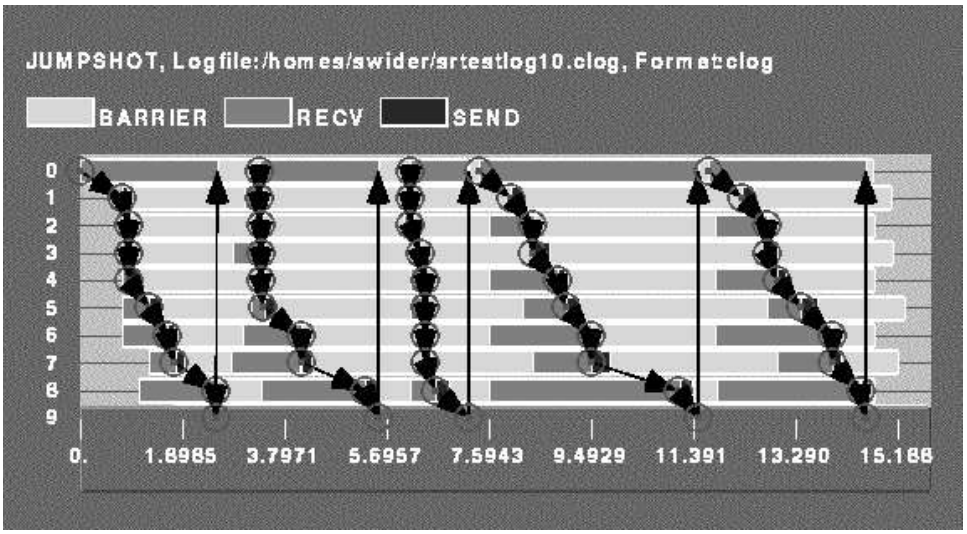


Figure 3: Jumpshot-1 Display

```
}
```

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 '`mpe/profiling/wrappergen`' or Appendix A.

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.

5.8 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 5.7). 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 that `-DMPE_GRAPHICS` be included in the flags to the C compiler.
- 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 (-lpmpich)> \
<standard mpi library (-lmpich)>
```

To simplify it, some sample makefile sections have been created in '`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 $(PROF_LIB) -lmpich
```

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
```

6 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.

6.1 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`).

6.2 Command-line arguments for `mpirun`

`mpirun` provides some help in starting programs with a debugger.

```
mpirun -dbg=<name of debugger> -np 2 program
```

starts `program` on two machines, with the local one running under the chosen debugger. There are 5 debugger scripts included with `mpich` which will be located in the '`mpich/bin`' directory after executing `make`. They are named `mpirun_dbg.%d` where `%d` can be replaced with `dbx`, `ddd`, `gdb`, `totalview`, and `xxgdb`. The appropriate script is invoked when the `-dbg` option is used with `mpirun`

6.3 MPI arguments for the application program

These are currently undocumented, and some require configure options to have been specified (like `-mpipktsize` and `-chmemdebug`). The `-mpiversion` option is useful for finding out how your installation of `mpich` was configured and exactly what version it is.

-mpedbg If an error occurs, start `xterm`s attached to the process that generated the error. Requires the `mpich` be configured with `-mpedbg` and works on only some workstations systems.

-mpiversion Print out the version and configuration arguements for the `mpich` implementation being used.

These arguments are provided to the program, not to `mpirun`. For example,

```
mpirun -np 2 a.out -mpiversion
```

6.4 Debugging with the `ch_p4` Device

When using the `ch_p4` device, a number of command-line arguments may be used to control the behavior of the program.

6.4.1 p4 Debugging

If your configuration of `mpich` used `-device=ch_p4`, then some of the `p4` debugging capabilities are available to you. The most useful of these are the command line arguments to the application program. Thus

```
mpirun -np 10 myprog -p4dbg 20 -p4rdbg 20
```

results in program tracing information at a level of 20 being written to `stdout` during execution. For more information about what is printed at what levels, see the `p4` Users' Guide [2].

If one specifies `-p4norem` on the command line, `mpirun` will not actually start the processes. The master process prints a message suggesting how the user can do it. The point of this option is to enable the user to start the remote processes under his favorite debugger, for instance. The option only makes sense when processes are being started remotely, such as on a workstation network. Note that this is an argument to the *program*, not to `mpirun`. For example, to run `myprog` this way, use

```
mpirun -np 4 myprog -p4norem
```

For example, to run `cpi` with 2 processes, where the second process is run under the debugger, the session would look something like

```
mpirun -np 2 cpi -p4norem
waiting for process on host shakey.mcs.anl.gov:
/home/me/mpich/examples/basic/cpi sys2.foo.edu 38357 -p4amslave
```

on the first machine and

```
% gdb cpi
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i586-mandrake-linux"...
(gdb) run sys2.foo.edu 38357 -p4amslave
Starting program: /home/me/mpich/examples/basic/cpi sys2.foo.edu 38357 -p4amslave
```

6.4.2 Setting the Working Directory for the p4 Device

By default, the working directory for processes running remotely with `ch_p4` device is the same as that of the executable. To specify a different working directory, use `-p4wdir` as follows:

```
mpirun -np 4 myprog -p4wdir myrundir
```

6.5 Command-line arguments for the application program

Arguments on the command line that follow the application program name and are not directed to the `mpich` system (don't begin with `-mpi` or `-p4`) are passed through to all processes of the application program. For example, if you execute

```
mpirun -echo -np 4 myprog -mpiversion -p4dbg 10 x y z
```

then `-echo -np 4` is interpreted by `mpirun` (echo actions of `mpirun` and run four processes), `-mpiversion` is interpreted by `mpich` (each process prints configuration information), `-p4dbg 10` is interpreted by the p4 device if your version was configured with `-device=ch_p4` (sets p4 debugging level to 10), and `x y z` are passed through to the application program. In addition, `MPI_Init` strips out non-application arguments, so that after the call to `MPI_Init` in your C program, the argument vector `argv` contains only

```
myprog x y z
```

and your program can process its own command-line arguments in the normal way. Note that the argument vector for Fortran and Fortran 77 programs will contain the `mpich` commands because there is no standard mechanism defined by Fortran for accessing or modifying the command line.

It is not possible to provide different command-line arguments for the different processes.

6.6 Starting jobs with a debugger

The `-dbg=<name of debugger>` option to `mpirun` causes processes to be run under the control of the chosen debugger. For example, enter

```
mpirun -dbg=gdb or mpirun -dbg=gdb a.out
```

invokes the `mpirun_dbg.gdb` script located in the '`mpich/bin`' directory. This script captures the correct arguments, invokes the `gdb` debugger, and starts the first process under `gdb` where possible. There are 4 debugger scripts; `gdb`, `xxgdb`, `ddd`, and `totalview`. These may need to be edited depending on your system. There is another debugger script for `dbx`, but this one will always need to be edited as the debugger commands for `dbx` varies between versions. You can also use this option to call another debugger; for example, `-dbg=mydebug`. All you need to do is write a script file, '`mpirun_dbg.mydebug`', which follows the format of the included debugger scripts, and place it in the '`mpich/bin`' directory. More information on using the Totalview debugger with `mpich` can be found in Section 7.

6.7 Starting the debugger when an error occurs

It is often convenient to have a debugger start when a program detects an error. If `mpich` was configured with the option `--enable-mpedbg`, then adding the command-line option `-mpedbg` to the program will cause `mpich` to attempt to start a debugger (usually `dbx` or `gdb`) when an error that generates a signal (such as `SIGSEGV`) occurs. For example,

```
mpirun -np 4 a.out -mpedbg
```

If you are not sure if your `mpich` provides this service, you can use `-mpiversion` to see if `mpich` was built with the `--enable-mpedbg` option.

6.8 Attaching a debugger to a running program

On workstation clusters, you can often attach a debugger to a running process. For example, the debugger `dbx` often accepts a process id (pid) which you can get by using the `ps` command. The form is either

```
dbx a.out 1234
```

or

```
dbx -pid 1234 a.out
```

where `1234` is the process id. One can also attach the TotalView debugger to a running program (See Section 7.3 below).

6.9 Signals

In general, users should avoid using signals with MPI programs. The manual page for `MPI_Init` describes the signals that are used by the MPI implementation; these should not be changed by the user.

Because Unix does not chain signals, there is the possibility that several packages will attempt to use the same signal, causing the program to fail. For example, by default, the `ch_p4` device uses `SIGUSR1`; some thread packages also use `SIGUSR1`. If you have such a situation, see the *Installation Manual for* `mpich` for information on how to select a different signal for use by `mpich`.

In a few cases, you can change the signal *before* calling `MPI_Init`. In those cases, your signal handler will be called after the `mpich` implementation acts on the signal. For example, if you want to change the behavior of `SIGSEGV` to print a message, you can establish such a signal handler before calling `MPI_Init`. With devices such as the `ch_p4` device that handle `SIGSEGV`, this will cause your signal handler to be called after `mpich` processes it.

6.10 Related tools

The *Scalable Unix Tools* (SUT) is a collection for managing workstation networks as a MPP. These include programs for looking at all of the processes in a cluster and performing operations on them (such as attaching the debugger to every process you own that is running a particular program). This is not part of MPI but can be very useful in working with workstation clusters. An MPI version of these tools is available at `www.mcs.anl.gov/sut`; this implementation works well with the `ch_p4mpd` device of `mpich`.

7 Debugging MPI programs with TotalView

TotalView© is a powerful, commercial-grade, portable debugger for parallel and multi-threaded programs, available from Etnus (`http://www.etnus.com/`). TotalView understands multiple MPI implementations, including `mpich`. By "understand" is meant that if you have TotalView installed on your system, it is easy to start your `mpich` program under the control of TotalView, even if you are running on multiple machines, manage your processes both collectively and individually through TotalView's convenient GUI, and even examine internal `mpich` data structures to look at message queues [3]. The general operation model of TotalView will be familiar to users of command-line-based debuggers such as `gdb` or `dbx`.

7.1 Preparing `mpich` for TotalView debugging

See the *Installation Guide* for instructions on configuring `mpich` so that TotalView can display message queues.

7.2 Starting an `mpich` program under TotalView control

To start a parallel program under TotalView control, simply add '`-dbg=totalview`' to your `mpirun` arguments:

```
mpirun -dbg=totalview -np 4 cpi
```

TotalView will come up and you can start the program by typing 'G'. A window will come up asking whether you want to stop processes as they execute MPI_Init. You may find it more convenient to say "no" and instead to set your own breakpoint after MPI_Init (see Section 7.4). This way when the process stops it will be on a line in your program instead of somewhere inside MPI_Init.

7.3 Attaching to a running program

TotalView can attach to a running MPI program, which is particularly useful if you suspect that your code has deadlocked. To do this start TotalView with no arguments, and then press 'N' in the root window. This will bring up a list of the processes that you can attach to. When you dive through the initial mpich process in this window TotalView will also acquire all of the other mpich processes (even if they are not local). See the TotalView manual for more details of this process.

7.4 Debugging with TotalView

You can set breakpoints by clicking in the left margin on a line number. Most of the TotalView GUI is self-explanatory. You select things with the left mouse button, bring up an action menu with the middle button, and "dive" into functions, variables, structures, processes, etc., with the right button. Pressing cntl-? in any TotalView window brings up help relevant to that window. In the initial TotalView window it brings up general help. The full documentation (The TotalView *User's Guide*) is available from the Etnus web site.

You switch from viewing one process to the next with the arrow buttons at the top-right corner of the main window, or by explicitly selecting (left button) a process in the root window to re-focus an existing window onto that process, or by diving (right button) through a process in the root window to open a new window for the selected process. All the keyboard shortcuts for commands are listed in the menu that is attached to the middle button. The commands are mostly the familiar ones. The special one for MPI is the 'm' command, which displays message queues associated with the process.

Note also that if you use the MPI-2 function MPI_Comm_set_name on a communicator, TotalView will display this name whenever showing information about the communicator, making it easier to understand which communicator is which.

8 Other MPI Documentation

Information about MPI is available from a variety of sources. Some of these, particularly WWW pages, include pointers to other resources.

- The Standard itself:
  - As a Technical report [4]
  - As Postscript and HTML at www.mpi-forum.org

  - As a journal article in the Fall 1994 issue of the Journal of Supercomputing Applications [14]

- MPI Forum discussions
  - The MPI Forum email discussions and both current and earlier versions of the Standard are available from `www.netlib.org`. MPI-2 discussions are available at `www.mpi-forum.org`.

- Books:
  - *Using MPI: Portable Parallel Programming with the Message-Passing Interface, Second Edition*, by Gropp, Lusk, and Skjellum [10].
  - *Using MPI-2: Advanced Features of the Message-Passing Interface*, by Gropp, Lusk, and Thakur [11]
  - *MPI—The Complete Reference: Volume 1, The MPI Core*, by Snir, et al. [16].
  - *MPI—The Complete Reference: Volume 2, The MPI-2 Extensions*, by Gropp, et al. [5].
  - *Parallel Programming with MPI*, by Peter S. Pacheco. [15]

- Newsgroup:
  - `comp.parallel.mpi`

- Mailing lists:
  - `mpi-comments@mpi-forum.org`: The MPI Forum discussion list.
  - `mpi-impl@mcs.anl.gov`: The implementors' discussion list.
  - `mpi-bugs@mcs.anl.gov`is the address to report problems with `mpich` to.

- Implementations available from the web:
  - `mpich` is available from `http://www.mcs.anl.gov/mpi/mpich` or by anonymous `ftp` from `ftp.mcs.anl.gov` in the directory '`pub/mpi/mpich`', file '`mpich.tar.gz`'.
  - LAM is available from `http://www.lam-mpi.org`.

- Test code repository:
  - `ftp://ftp.mcs.anl.gov/pub/mpi/mpi-test`

9 In Case of Trouble

This section describes some commonly encountered problems and their solutions. It also describes machine-dependent considerations. Send any problem that you can not solve by checking this section to `mpi-bugs@mcs.anl.gov`.

Please include:

- The version of `mpich` (e.g., 1.2.2)
- The output of running your program with the `-mpiversion` argument (e.g., `mpirun -np 1 a.out -mpiversion`)
- The output of

  ```
  uname -a
  ```

  for your system. If you are on an SGI system, also

  ```
  hinv
  ```

- If the problem is with a script such as `configure` or `mpirun`, run the script with the `-echo` argument (e.g., `mpirun -echo -np 4 a.out`).
- If you are using the `ch_p4` device, also send the output of `sbin/tstmachines`.

Each section is organized in question and answer format, with questions that relate to more than one environment (workstation, operating system, etc.) first, followed by questions that are specific to a particular environment. Problems with workstation clusters are collected together as well.

9.1 Problems compiling or linking Fortran programs

9.1.1 General

1. **Q:** When linking the test program, the following message is generated:

   ```
   f77 -g -o secondf secondf.o -L/usr/local/mpich/lib/sun4/ch_p4 -lmpich
   invalid option -L/usr/local/mpich/lib/sun4/ch_p4
   ld: -lmpich: No such file or directory
   ```

   A: This `f77` program does not accept the `-L` command to set the library search path. Some systems provide a shell script for `f77` that is very limited in its abilities. To work around this, use the full library path instead of the `-L` option:

   ```
   f77 -g -o secondf secondf.o /usr/local/mpich/lib/sun4/ch_p4/libmpich.a
   ```

   As of the `mpich` 1.2.0 release, the `mpich` configure attempts to find the correct option for indicating library paths to the Fortran compiler. If you find that the `mpich` configure has made an error, please submit a bug report to `mpi-bugs@mcs.anl.gov`.

2. **Q:** When linking Fortran programs, I get undefined symbols such as

   ```
   f77  -c secondf.f
   secondf.f:
    MAIN main:
   f77  -o secondf secondf.o -L/home/mpich/lib/solaris/ch_shmem -lmpich
   ```

```
Undefined                       first referenced
 symbol                             in file
getdomainname
/home/mpich/lib/solaris/ch_shmem/libmpi .a(shmempriv.o)
ld: fatal: Symbol referencing errors. No output written to secondf
```

There is no problem with C programs.

A: This means that your C compiler is providing libraries for you that your Fortran compiler is not providing. Find the option for the C compiler and for the Fortran compilers that indicate which library files are being used (alternately, you may find an option such as `-dryrun` that shows what commands are being used by the compiler). Build a simple C and Fortran program and compare the libraries used (usually on the `ld` command line). Try the ones that are present for the C compiler and missing for the Fortran compiler.

3. **Q:** When trying to compile Fortran code with a Fortran 90 or Fortran 95 compiler, I get error messages like

   ```
   Error: foo.f, line 30: Inconsistent datatype for argument 1 in MPI_SEND
   ```

   A: The Fortran language requires that in two calls to the same subroutine, the types of arguments must be the same. That is, if you called `MPI_SEND` with a `REAL` buffer as the first argument and then called it with an `INTEGER` buffer as the first argument, a Fortran compiler can consider this an error. Few Fortran 77 compilers would complain about this; more Fortran 90 and Fortran 95 compilers check for this. There are two solutions. One is to use the MPI module (in the "choice" version: use the `-choicemod` option for `mpif90`); the other is to use an option to tell the Fortran 90 compiler to allow argument mismatches. For example, the argument `-mismatch` will cause the NAG Fortran compilers to allow mismatched arguments. Using the MPI module is the preferred approach.

   Fortran 77 users may sometimes see a similar message, particularly with later versions of `g77`. The option `-Wno-globals` will supress these warning messages.

9.2 Problems Linking C Programs

9.2.1 General

1. **Q:** When linking programs, I get messages about `__builtin_saveregs` being undefined.

   A: You may have a system on which C and Fortran compilers are incompatible (for example, using `gcc` and the vendor's Fortran compiler). If you do not plan to use Fortran, the easiest fix is to rebuild with the `-nof77` option to configure.

   You should also look into making your C compiler compatible with your Fortran compiler.

   The easiest but ugliest possibility is use `f2c` to convert Fortran to C, then use the C compiler to compile everything. If you take this route, remember that *every* Fortran routine has to be compiled using `f2c` and the C compiler.

Alternatively, you can use various options (check the man pages for your compilers) to see what libraries that add when they link. Add those libraries to the link line for the *other* compiler. If you find a workable set of libraries, edit the appropriate scripts (e.g., `mpicc`) to include the necessary libraries. `Mpich` attempts to find all the libaries that you need but is not always successful.

9.2.2 Sun Solaris

1. **Q:** When linking on Solaris, I get an error like this:

```
cc -g -o testtypes testtypes.o -L/usr/local/mpich/lib/solaris/ch_p4 -lmpich
       -lsocket -lnsl -lthread
ld: warning: symbol `_defaultstkcache' has differing sizes:
       (file /usr/lib/libthread.so value=0x20; file /usr/lib/libaio.so
       value=0x8);
       /usr/lib/libthread.so definition taken
```

A: This is a bug in Solaris 2.3 that is fixed in Solaris 2.4. There may be a patch for Solaris 2.3; contact Sun for more information.

9.2.3 HPUX

1. **Q:** When linking on HPUX, I get an error like this:

```
cc -o pgm pgm.o -L/usr/local/mpich/lib/hpux/ch_p4 -lmpich  -lm
/bin/ld: Unsatisfied symbols:
sigrelse (code)
sigset (code)
sighold (code)
*** Error code 1
```

A: You need to add the link option `-lV3`. The p4 device uses the System V signals on the HP; these are provided in the '`V3`' library.

9.2.4 LINUX

1. **Q:** When linking a Fortran program, I get

```
Linking:
foo.o(.data+0x0): undefined reference to `pmpi_wtime_'
```

A: This is a bug in the `pgf77` compiler (which is itself a workaround for a bug in the LINUX `ld` command). You can fix it by either adding -lpmpich to the link line or modifying the '`mpif.h`' to remove the `external pmpi_wtime, pmpi_wtick` statement.

The `mpich` configure attempts to determine if `pmpi_wtime` and `pmpi_wtick` can be declared in '`mpif.h`' and removes them if there is a problem. If this happens and you use `pmpi_wtime` or `pmpi_wtick` in your program, you will need to declare them as functions returning `double precision` values.

9.3 Problems starting programs

9.3.1 General

1. **Q:** When trying to start a program with

   ```
   mpirun -np 2 cpi
   ```

   either I get an error message or the program hangs.

   A: On some systems such as IBM SPs, there are many mutually exclusive ways to run parallel programs; each site can pick the approach(es) that it allows. The script `mpirun` tries one of the more common methods, but may make the wrong choice. Use the `-v` or `-t` option to `mpirun` to see how it is trying to run the program, and then compare this with the site-specific instructions for using your system. You may need to adapt the code in `mpirun` to meet your needs. See also the next question.

2. **Q:** When trying to run a program with, e.g., `mpirun -np 4 cpi`, I get

   ```
   usage : mpirun [options] <executable> [<dstnodes>] [-- <args>]
   ```

   or

   ```
   mpirun [options] <schema>
   ```

   A: You have a command named `mpirun` in your path ahead of the `mpich` version. Execute the command

   ```
   which mpirun
   ```

   to see which command named `mpirun` was actually found. The fix is to either change the order of directories in your path to put the `mpich` version of `mpirun` first, or to define an alias for `mpirun` that uses an absolute path. For example, in the csh shell, you might do

   ```
   alias mpirun /usr/local/mpich/bin/mpirun
   ```

   to set `mpirun` to the `mpich` version.

3. **Q:** When trying to start a large number of processes on a workstation network, I get the message

   ```
   p4_error: latest msg from perror: Too many open files
   ```

   A: There is a limitation on the number of open file descriptors. On some systems you can increase this limit yourself; on others you must have help from your system administrator. You could experiment with the secure server, but it is not a complete solution. We are working now on a more scalable startup mechanism for the next release.

4. **Q:** When attempting to run `cpilog` I get the following message:

```
ld.so.1: cpilog: fatal: libX11.so.4: can't open file: errno 2
```

A: The X11 version that `configure` found isn't properly installed. This is a common problem with Sun/Solaris systems. One possibility is that your Solaris machines are running slightly different versions. You can try forcing static linking (-Bstatic on Solaris).

Alternately, consider adding these lines to your '`.login`' (assuming C shell):

```
setenv OPENWINHOME /usr/openwin
setenv LD_LIBRARY_PATH /opt/SUNWspro/lib:/usr/openwin/lib
```

(you may want to check with your system administrator first to make sure that the paths are correct for your system). Make sure that you add them *before* any line like

```
if ($?USER == 0 || $?prompt == 0) exit
```

5. **Q:** My program fails when it tries to write to a file.

A: If you opened the file *before* calling `MPI_INIT`, the behavior of MPI (not just the `mpich` implementation of MPI) is undefined. In the `ch_p4` device, only process zero (in `MPI_COMM_WORLD`) will have the file open; the other processes will not have opened the file. Move the operations that open files and interact with the outside world to after `MPI_INIT` (and before `MPI_FINALIZE`).

6. **Q:** Programs seem to take forever to start.

A: This can be caused by any of several problems. On systems with dynamically-linked executables, this can be caused by problems with the file system suddenly getting requests from many processors for the dynamically-linked parts of the executable (this has been measured as a problem with some DFS implementations). You can try statically linking your application.

On workstation networks, long startup times can be due to the time used to start remote processes; see the discussion on the secure server in Section 3.2.3 for the `ch_p4` device or consider using the `ch_p4mpd` device.

9.3.2 Workstation Networks

1. **Q:** When I use `mpirun`, I get the message `Permission denied`.

A: If you see something like this

```
% mpirun -np 2 cpi
Permission denied.
```

or

```
% mpirun -np 2 cpi
socket: protocol failure in circuit setup
```

when using the `ch_p4` device, it probably means that you do not have permission to use `rsh` to start processes. The script `tstmachines` can be used to test this. Try

```
tstmachines
```

If this fails, then you may need a '`.rhosts`' or '`/etc/hosts.equiv`' file (you may need to see your system administrator) or you may need to use the p4 server (see Section 3.2.3). Another possible problem is the choice of the remote shell program; some systems have several. Check with your systems administrator about which version of `rsh` or `remsh` you should be using. If you must use `ssh`, see the section on using `ssh` in the *Installation Manual*.

If your system policy allows a '`.rhosts`' file, do the following:

(a) Create a file '`.rhosts`' in your home directory

(b) Change the protection on it to user read/write only: `chmod og-rwx .rhosts`.

(c) Add one line to the '`.rhosts`' file for each processor that you want to use. The format is

```
host username
```

For example, if your username is `doe` and you want to user machines `a.our.org` and `b.our.org`, your '`.rhosts`' file should contain

```
a.our.org doe
b.our.org doe
```

Note the use of fully qualified host names (some systems require this).

On networks where the use of '`.rhosts`' files is not allowed, you should use the secure server to run on machines that are not trusted by the machine that you are initiating the job from.

Finally, you may need to use a non-standard `rsh` command within `mpich`. `mpich` must be reconfigured with `-rsh=command_name`, and perhaps also with `-rshnol` if the remote shell command does not support the `-l` argument. Systems using Kerberos and/or AFS may need this. See the section in the *Installation Guide* on using the secure shell `ssh`.

An alternate source of the "Permission denied." message is that you have used the `su` command to change your effective user id. On some systems the `ch_p4` device will not work in this situation. Log in normally and try again.

2. **Q:** When I use `mpirun`, I get the message `Try again`.

   A: If you see something like this

   ```
   % mpirun -np 2 cpi
   Try again.
   ```

   it means that you were unable to start a remote job with the remote shell command on some machine, even though you would normally be able to. This may mean that the destination machine is very busy, out of memory, or out of processes. The man page for `rshd` may give you more information.

   The only fix for this is to have your system administrator look into the machine that is generating this message.

3. **Q:** When running the `ch_p4` device, I get error messages of the form

```
stty: TCGETS: Operation not supported on socket
```

or

```
stty: tcgetattr: Permission denied
```

or

```
stty: Can't assign requested address
```

A: This means that one of your login startup scripts (i.e., '`.login`' and '`.cshrc`' or '`.profile`') contains an unguarded use of the `stty` or `tset` program. For C shell users, one typical fix is to check for the variables `TERM` or `PROMPT` to be initialized. For example,

```
if ($?TERM) then
    eval `tset -s -e^\? -k^U -Q -I $TERM`
endif
```

Another solution is to see if it is appropriate to add

```
if ($?USER == 0 || $?prompt == 0) exit
```

near the top of your '`.cshrc`' file (but *after* any code that sets up the runtime environment, such as library paths (e.g., `LD_LIBRARY_PATH`)).

4. **Q:** When running the `ch_p4` device and running either the `tstmachines` script to check the machines file or the `mpich` tests, I get messages about unexpected output or differences from the expected output. I also get extra output when I run programs. MPI programs do seem to work, however.

   A: This means that one your login startup scripts (i.e., '`.login`' and '`.cshrc`' or '`.profile`' or '`.bashrc`') contains an unguarded use of some program that generates output, such as `fortune` or even `echo`. For C shell users, one typical fix is to check for the variables `TERM` or `PROMPT` to be initialized. For example,

```
if ($?TERM) then
    fortune
endif
```

Another solution is to see if it is appropriate to add

```
if ($?USER == 0 || $?prompt == 0) exit
```

near the top of your '`.cshrc`' file (but *after* any code that sets up the runtime environment, such as library paths (e.g., `LD_LIBRARY_PATH`)).

5. **Q:** Occasionally, programs fail with the message

```
poll: protocol failure during circuit creation
```

A:

You may see this message if you attempt to run too many MPI programs in a short period of time. For example, in Linux and when using the `ch_p4` device (without the secure server or ssh), `mpich` may use `rsh` to start the MPI processes. Depending on the particular Linux distribution and verison, there may be a limit of as few as 40 processes per minute. When running the `mpich` test suite or starting short parallel jobs from a script, it is possible to exceed this limit.

To fix this, you can do one of the following:

(a) Wait a few seconds between running parallel jobs. You may need to wait up to a minute.

(b) Modify '`/etc/inetd.conf`' to allow more processes per minute for `rsh`. For example, change

```
shell stream tcp nowait root /etc/tcpd2 in.rshd
```

to

```
shell stream tcp nowait.200 root /etc/tcpd2 in.rshd
```

(c) Use the `ch_p4mpd` device or the secure server option of the `ch_p4` device instead. Neither of these relies on `inetd`.

6. **Q:** When using `mpirun` I get strange output like

```
arch: No such file or directory
```

A: This is usually a problem in your '`.cshrc`' file. Try the shell command

```
which hostname
```

If you see the same strange output, then your problem is in your '`.cshrc`' file. You may have some code in your '`.cshrc`' file that assumes that your shell is connected to a terminal.

7. **Q:** When I try to run my program, I get

```
p0_4652:  p4_error: open error on procgroup file (procgroup): 0
```

A: This indicates that the `mpirun` program did not create the expected input file needed to run the program. The most likely reason is that the `mpirun` command is trying to run a program built with device `ch_p4` as a shared memory (`ch_shmem`) or other device.

Try the following:

Run the program using mpirun and the `-t` argument:

```
mpirun -t -np 1 foo
```

This should show what mpirun would do (`-t` is for testing). Or you can use the `-echo` argument to see exactly what mpirun is doing:

```
mpirun -echo -np 1 foo
```

Depending on the choice made by the installer of `mpich`, you should select the device-specific version of mpirun over a "generic" version. We recommend that the installation prefix include the device name, for example, '`/usr/local/mpich/solaris/ch_p4`'.

8. **Q:** When trying to run a program I get this message:

```
icy%  mpirun -np 2 cpi -mpiversion
icy: icy: No such file or directory
```

A: Your problem is that '`/usr/lib/rsh`' is not the remote shell program. Try the following:

```
which rsh
ls /usr/*/rsh
```

You probably have '`/usr/lib`' in your path ahead of '`/usr/ucb`' or '`/usr/bin`'. This picks the 'restricted' shell instead of the 'remote' shell. The easiest fix is to just remove '`/usr/lib`' from your path (few people need it); alternately, you can move it to after the directory that contains the 'remote' shell rsh.

Another choice would be to add a link in a directory earlier in the search path to the remote shell. For example, I have '`/home/gropp/bin/solaris`' early in my search path; I could use

```
cd /home/gropp/bin/solaris
ln -s /usr/bin/rsh  rsh
```

there (assuming that '`/usr/bin/rsh`' is the remote shell).

9. **Q:** When trying to run a program I get this message:

```
trying normal rsh
```

A: You are using a version of the remote shell program that does not support the `-l` argument. Reconfigure `mpich` with `-rshnol` and rebuild `mpich`. You may suffer some loss of functionality if you try to run on systems where you have different user names. You might also try using `ssh`.

10. **Q:** When I run my program, I get messages like

```
| ld.so: warning: /usr/lib/libc.so.1.8 has older revision than expected 9
```

A: You are trying to run on another machine with an out-dated version of the basic C library. Unfortunately, some manufacturers do not make the shared libraries compatible between minor (or even maintenance) releases of their software. You need to have you system administrator bring the machines to the same software level.

One temporary fix that you can use is to add the link-time option to force static linking instead of dynamic linking for system libraries. For some Sun workstations, the option is `-Bstatic`.

11. **Q:** Programs never get started. Even `tstmachines` hangs.

    A: Check first that `rsh` works at all. For example, if you have workstations `w1` and `w2`, and you are running on `w1`, try

    ```
    rsh w2 true
    ```

    This should complete quickly. If it does not, try

    ```
    rsh w1 true
    ```

    (that is, use `rsh` to run `true` on the system that you are running on). If you get `Permission denied`, see the help on that. If you get

    ```
    krcmd: No ticket file (tf_util)
    rsh: warning, using standard rsh: can't provide Kerberos auth data.
    ```

    then your system has a faulty installation of `rsh`. Some FreeBSD systems have been observed with this problem. Have your system administrator correct the problem (often one of an inconsistent set of `rsh/rshd` programs).

12. **Q:** When running `ch_p4` device, I get error messages of the form

    ```
    more slaves than message queues
    ```

    A: This means that you are trying to run `mpich` in one mode when it was configured for another. In particular, you are specifying in your p4 procgroup file that several processes are to shared memory on a particular machine by either putting a number greater than 0 on the first line (where it signifies number of local processes besides the original one), or a number greater than 1 on any of the succeeding lines (where it indicates the total number of processes sharing memory on that machine). You should either change your procgroup file to specify only one process on line, or reconfigure `mpich` with

    ```
    configure --with-device=ch_p4 -comm=shared
    ```

    which will reconfigure the p4 device so that multiple processes can share memory on each host. The reason this is not the default is that with this configuration you will see busy waiting on each workstation, as the device goes back and forth between selecting on a socket and checking the internal shared-memory queue.

13. **Q:** My programs seem to hang in `MPI_Init`.

    A: There are a number of ways that this can happen:

    (a) One of the workstations you selected to run on is dead (try '`tstmachines`' if you are using the `ch_p4` device)

    (b) You linked with the FSU pthreads package; this has been reported to cause problems, particularly with the system `select` call that is part of Unix and is used by `mpich`.

    Another is if you use the library '`-ldxml`' (extended math library) on Compaq Alpha systems. This has been observed to cause `MPI_Init` to hang. No workaround is known at this time; contact Compaq for a fix if you need to use MPI and '`-ldxml`' together.

    The root of this problem is that the `ch_p4` device uses `SIG_USR1`, and so any library that also uses this signal can interfere with the operation of `mpich` if it is using `ch_p4`. You can rebuild `mpich` to use a different signal by using the configure argument `--with-device=ch_p4:-listener_sig=SIGNAL_NAME` and remaking `mpich`.

14. **Q:** My program (using device `ch_p4`) fails with

```
p0_2005:  p4_error: fork_p4: fork failed: -1
          p4_error: latest msg from perror: Error 0
```

    A: The executable size of your program may be too large. When a `ch_p4` or `ch_tcp` device program starts, it may create a copy of itself to handle certain communication tasks. Because of the way in which the code is organized, this (at least temporarily) is a full copy of your original program and occupies the same amount of space. Thus, if your program is over half as large as the maximum space available, you wil get this error. On SGI systems, you can use the command `size` to get the size of the executable and `swap -l` to get the available space. Note that `size` gives you the size in bytes and `swap -l` gives you the size in 512-byte blocks. Other systems may offer similar commands.

    A similar problem can happen on IBM SPs using the `ch_mpl` device; the cause is the same but it originates within the IBM MPL library.

15. **Q:** Sometimes, I get the error

```
Exec format error. Wrong Architecture.
```

    A: You are probably using NFS (Network File System). NFS can fail to keep files updated in a timely way; this problem can be caused by creating an executable on one machine and then attempting to use it from another. Usually, NFS catches up with the existence of the new file within a few minutes. You can also try using the `sync` command. `mpirun` in fact tries to run the `sync` command, but on many systems, `sync` is only advisory and will not guarentee that the file system has been made consistent.

16. **Q:** There seem to be two copies of my program running on each node. This doubles the memory requirement of my application. Is this normal?

A: Yes, this is normal. In the `ch_p4` implementation, the second process is used to dynamically establish connections to other processes. With Version 1.1.1 of `mpich`, this functionality can be placed in a separate thread on many architectures, and this second process will not be seen. To enable this, use the option `-p4_opts=-threaded_listener` on the `configure` command line for `mpich`.

17. **Q:** `MPI_Abort` sometimes doesn't work in the `ch_p4` device. Why?

    A: Currently (Version 1.2.2) a process detects that another process has aborted only when it tries to send or receive a message, and the aborting process is one that it has communicated with in the past. Thus it is porssible for a process busy with computation not to notice that one of its peers has issued an `MPI_Abort`, although for many common communication patterns this does not present a problem. This will be fixed in a future release.

9.3.3 IBM RS6000

1. **Q:** When trying to run on an IBM RS6000 with the `ch_p4` device, I got

```
% mpirun -np 2 cpi
Could not load program /home/me/mpich/examples/basic/cpi
Could not load library libC.a[shr.o]
Error was: No such file or directory
```

   A: This means that `mpich` was built with the `xlC` compiler but that some of the machines in your '`util/machines/machines.rs6000`' file do not have `xlC` installed. Either install `xlC` or rebuild `mpich` to use another compiler (either `xlc` or `gcc`; `gcc` has the advantage of never having any licensing restrictions).

9.3.4 IBM SP

1. **Q:** When starting my program on an IBM SP, I get this:

```
$ mpirun -np 2 hello
ERROR: 0031-124  Couldn't allocate nodes for parallel execution.  Exiting ...
ERROR: 0031-603  Resource Manager allocation for task: 0, node:
me1.myuniv
.edu, rc = JM_PARTIONCREATIONFAILURE
ERROR: 0031-635  Non-zero status -1 returned from pm_mgr_init
```

   A: This means that either `mpirun` is trying to start jobs on your SP in a way different than your installation supports or that there has been a failure in the IBM software that manages the parallel jobs (all of these error messages are from the IBM `poe` command that `mpirun` uses to start the MPI job). Contact your system administrator for help in fixing this situation. You system administrator can use

```
dsh -av "ps aux | egrep -i 'poe|pmd|jmd'"
```

from the control workstation to search for stray IBM POE jobs that can cause this behavior. The files `/tmp/jmd_err` on the individual nodes may also contain useful diagnostic information.

2. **Q:** When trying to run on an IBM SP, I get the message from `mpirun`:

```
ERROR: 0031-214  pmd: chdir </a/user/gamma/home/mpich/examples/basic>
ERROR: 0031-214  pmd: chdir </a/user/gamma/home/mpich/examples/basic>
```

   A: These are messages from tbe IBM system, not from `mpirun`. They may be caused by an incompatibility between POE, the automounter (especially the AMD automounter) and the shell, especially if you are using a shell other than `ksh`. There is no good solution; IBM often recommends changing your shell to `ksh`!

3. **Q:** When I tried to run my program on an IBM SP, I got

```
ERROR : Cannot locate message catalog (pepoe.cat) using current NLSPATH
INFO  : If NLSPATH is set correctly and catalog exists, check LANG or
LC_MESSAGES variables
(C) Opening of "pepoe.cat" message catalog failed
```

   (and other variations that mention NLSPATH and "message catalog").

   A: This is a problem in your system; contact your support staff. Have them look at (a) value of NLSPATH, (b) links from '`/usr/lib/nls/msg/prime`' to the appropriate language directory. The messages are not from `mpich`; they are from the IBM POE/MPL code that the `mpich` implementation is using.

4. **Q:** When trying to run on an IBM SP, I get this message:

```
ERROR: 0031-124  Less than 2 nodes available from pool 0
```

   A: This means that the IBM POE/MPL system could not allocate the requested nodes when you tried to run your program; most likely, someone else was using the system. You can try to use the environment variables `MP_RETRY` and `MP_RETRYCOUNT` to cause the job to wait until the nodes become available. Use `man poe` to get more information.

5. **Q:** When running on an IBM SP, my job generates the message

```
Message number 0031-254 not found in Message Catalog.
```

   and then dies.

   A: If your user name is eight characters long, you may be experiencing a bug in the IBM POE environment. The only fix at the time this was written was to use an account whose user name was seven characters or less. Ask your IBM representative about PMR 4017X (poe with userids of length eight fails) and the associated APAR IX56566.

9.4 Programs fail at startup

9.4.1 General

1. **Q:** With some systems, you might see

```
/lib/dld.sl: Bind-on-reference call failed
/lib/dld.sl: Invalid argument
```

(This example is from HP-UX), or

```
ld.so: libc.so.2: not found
```

(This example is from SunOS 4.1; similar things happen on other systems).

A: The problem here is that your program is using shared libraries, and the libraries are not available on some of the machines that you are running on. To fix this, relink your program without the shared libraries. To do this, add the appropriate command-line options to the link step. For example, for the HP system that produced the errors above, the fix is to use `-Wl,-Bimmediate` to the link step. For Solaris, the appropriate option is `-Bstatic`.

9.4.2 Workstation Networks

1. **Q:** I can run programs using a small number of processes, but once I ask for more than 4–8 processes, I do not get output from all of my processes, and the programs never finish.

A: We have seen this problem with installations using AFS. The remote shell program, `rsh`, supplied with some AFS systems limits the number of jobs that can use standard output. This seems to prevent some of the processes from exiting as well, causing the job to hang. There are four possible fixes:

(a) Use a different `rsh` command. You can probably do this by putting the directory containing the non-AFS version first in your `PATH`. This option may not be available to you, depending on your system. At one site, the non-AFS version was in '`/bin/rsh`'.

(b) Use the secure server (`serv_p4`). See the discussion in the Users Guide.

(c) Redirect all standard output to a file. The MPE routine `MPE_IO_Stdout_to_file` may be used to do this.

(d) Get a fixed `rsh` command. The likely source of the problem is an incorrect usage of the `select` system call in the `rsh` command. If the code is doing something like

```
int mask;
mask |= 1 << fd;
select( fd+1, &mask, ... );
```

instead of

```
fd_set mask;
FD_SET(fd,&mask);
select( fd+1, &mask, ... );
```

then the code is incorrect (the `select` call changed to allow more than 32 file descriptors many years ago, and the `rsh` program (or programmer!) hasn't changed with the times).

A fourth possiblity is to get an AFS version of `rsh` that fixes this bug. As we are not running AFS ourselves, we do not know whether such a fix is available.

2. **Q:** Not all processes start.

   A: This can happen when using the `ch_p4` device and a system that has extremely small limits on the number of remote shells you can have. Some systems using "Kerberos" (a network security package) allow only three or four remote shells; on these systems, the size of `MPI_COMM_WORLD` will be limited to the same number (plus one if you are using the local host).

   The only way around this is to try the secure server; this is documented in the `mpich` installation guide. Note that you will have to start the servers "by hand" since the `chp4_servs` script uses remote shell to start the servers.

9.5 Programs fail after starting

9.5.1 General

1. **Q:** I use `MPI_Allreduce`, and I get different answers depending on the number of processes I'm using.

   A: The MPI collective routines may make use of associativity to achieve better parallelism. For example, an

   ```
   MPI_Allreduce( &in, &out, MPI_DOUBLE, 1, ... );
   ```

   might compute

   $$((((((a+b)+c)+d)+e)+f)+g)+h)$$

   or it might compute

   $$((a+b)+(c+d))+((e+f)+(g+h)),$$

   where $a, b, \ldots$ are the values of `in` on each of eight processes. These expressions are equivalent for integers, reals, and other familar objects from mathematics but are *not* equivalent for datatypes, such as floating point, used in computers. The association that MPI uses will depend on the number of processes, thus, you may not get exactly the same result when you use different numbers of processes. Note that you are not getting a wrong result, just a different one (most programs assume the arithmetic operations are associative).

2. **Q:** My Fortran program fails with a BUS error.

   A: The C compiler that `mpich` was built with and the Fortran compiler that you are using have different alignment rules for things like `DOUBLE PRECISION`. For example, the GNU C compiler `gcc` may assume that all `double`s are aligned on eight-byte boundaries, but the Fortran language requires only that `DOUBLE PRECISION` align with `INTEGER`s, which may be four-byte aligned.

   There is no good fix. Consider rebuilding `mpich` with a C compiler that supports weaker data alignment rules. Some Fortran compilers will allow you to force eight-byte alignment for `DOUBLE PRECISION` (for example, `-dalign` or `-f` on some Sun Fortran compilers); note though that this may break some correct Fortran programs that exploit Fortran's storage association rules.

   Some versions of `gcc` may support `-munaligned-doubles`; `mpich` should be rebuilt with this option if you are using `gcc`, version 2.7 or later. `Mpich` attemps to detect and use this option where available.

3. **Q:** I'm using `fork` to create a new process, or I'm creating a new thread, and my code fails.

   A: The `mpich` implementation is not thread safe and does not support either fork or the creation of new processes. Note that the MPI specification is thread safe, but implementations are not required to be thread safe. At this writing, few implementations are thread-safe, primarily because this reduces the performance of the MPI implementation (you at least need to check to see if you need a thread lock, actually getting and releasing the lock is even more expensive).

   The `mpich` implementation supports the `MPI_Init_thread` call; with this call, new in MPI-2, you can find out what level of thread support the MPI implementation supports. As of version 1.2.0 of `mpich`, only `MPI_THREAD_SINGLE` is supported. We believe that version 1.2.0 and later support `MPI_THREAD_FUNNELED`, and some users have used `mpich` in this mode (particularly with OpenMP), but we have not rigourously tested `mpich` for this mode. Future versions of `mpich` will support `MPI_THREAD_MULTIPLE`.

   Q: C++ programs execute global destructors (or constructors) more times than expected. For example:

```
class Z {
public:
  Z()  { cerr << "*Z" << endl; }
  ~Z() { cerr << "+Z" << endl; }
};

Z z;

int main(int argc, char **argv) {
  MPI_Init(&argc, &argv);
  MPI_Finalize();
}
```

   when running with the `ch_p4` device on two processes executes the destructor twice for each process.

A: The number of processes running before `MPI_Init` or after `MPI_Finalize` is not defined by the MPI standard; you can not rely on any specific behavior. In the `ch_p4` case, a new process is `fork`ed to handle connection requests; it terminates with the end of the program.

You can use the threaded listener with the `ch_p4` device, or use the `ch_p4mpd` device instead. Note, however, that this code is not portable because it relies on behavior that the MPI standard does not specify.

9.5.2 HPUX

1. **Q:** My Fortran programs seem to fail with `SIGSEGV` when running on HP workstations.

   A: Try compiling and linking the Fortran programs with the option `+T`. This *may* be necessary to make the Fortran environment correctly handle interrupts used by `mpich` to create connections to other processes.

9.5.3 ch_shmem device

1. **Q:** My program sometimes hangs when using the `ch_shmem` device.

   A: Make sure that you are linking with *all* of the correct libraries. If you are not using `mpicc`, try using `mpicc` to link your application. The reason for this is that correct operation of the shared-memory version may depend on additional, system-provided libraries. For example, under Solaris, the thread library must used, otherwise, non-functional versions of the mutex routines critical to the correct functioning of the MPI implementation are taken from '`libc`' instead.

9.5.4 LINUX

1. **Q:** Processes fail with messages like

   ```
   p0_1835:  p4_error: Found a dead connection while looking for messages: 1
   ```

   A: What is happening is that the TCP implementation on this platform is deciding that the connection has "failed" when it really hasn't. The current `mpich` implementation assumes that the TCP implementation will not close connections and has no code to reanimate failed connections. Future versions of `mpich` will work around this problem.

   In addition, some users have found that the single processor Linux kernel is more stable than the SMP kernel.

9.5.5 Workstation Networks

1. **Q:** My job runs to completion but exits with the message

```
Timeout in waiting for processes to exit.  This may be due to a defective
rsh program (Some versions of Kerberos rsh have been observed to have this
problem).
This is not a problem with P4 or \mpich\ but a problem with the operating
environment.  For many applications, this problem will only slow down
process termination.
```

What does this mean?

A:

If anything causes the rundown in `MPI_Finalize` to take more than about 5 minutes, it becomes suspicious of the `rsh` implementation. The `rsh` used with some Kerberos installations assumed that `sizeof(FD_SET) == sizeof(int)`. This meant that the `rsh` program assumed that the largest FD value was 31. When a program uses `fork` to create processes that launch `rsh`, while maintaining the stdin, stdout, and stderr to the forked process, this assumption is no longer true, since the FD that `rsh` creates for the socket may be > 31 if there are enough processes running. When using such a broken implementation of `rsh`, the symptom is that jobs never terminate because the `rsh` jobs are waiting (with select) for the socket to close.

The `ch_p4mpd` device eliminates this problem.

9.6 Trouble with Input and Output

9.6.1 General

1. **Q:** I want output from `printf` to appear immediately.

   A: This is really a feature of your C and/or Fortran runtime system. For C, consider

   ```
   setbuf( stdout, (char *)0 );
   ```

   or

   ```
   setvbuf( stdout, NULL, _IONBF, 0 );
   ```

9.6.2 IBM SP

1. **Q:** I have code that prompts the user and then reads from standard input. On IBM SPx systems, the prompt does not appear until *after* the user answers the prompt!

   A: This is a feature of the IBM POE system. There is a POE routine, `mpc_flush(1)`, that you can use to flush the output. Read the man page on this routine; it is synchronizing over the entire job and cannot be used unless all processes in `MPI_COMM_WORLD` call it. Alternately, you can always end output with the newline character (`\n`); this will cause the output to be flushed but will also put the user's input on the next line.

9.6.3 Workstation Networks

1. **Q:** I want standard output (`stdout`) from each process to go to a different file.

   A: `mpich` has no built-in way to do this. In fact, it prides itself on gathering the stdouts for you. You can do one of the following:

   (a) Use Unix built-in commands for redirecting `stdout` from inside your program (`dup2`, etc.). The MPE routine `MPE_IO_Stdout_to_file`, in '`mpe/src/mpe_io.c`', shows one way to do this. Note that in Fortran, the approach of using `dup2` will work only if the Fortran `PRINT` writes to `stdout`. This is common but by no means universal.

   (b) Write explicitly to files instead of to `stdout` (use `fprintf` instead of `printf`, etc.). You can create the file name from the process's rank. This is the most portable way.

9.7 Upshot and Nupshot

The `upshot` and `nupshot` programs require specific versions of the `tcl` and `tk` languages. This section describes only problems that may occur once these tools have been successfully built.

9.7.1 General

1. **Q:** When I try to run `upshot` or `nupshot`, I get

   ```
   No display name and no $DISPLAY environment variables
   ```

   A: Your problem is with your X environment. Upshot is an X program. If your workstation name is '`foobar.kscg.gov.tw`', then before running any X program, you need to do

   ```
   setenv DISPLAY foobar.kscg.gov.tw:0
   ```

   If you are running on some other system and displaying on `foobar`, you might need to do

   ```
   xhost +othermachine
   ```

   on `foobar`; this gives `othermachine` permission to write on `foobar`'s display.

   If you do not have an X display (you are logged in from a Windows machine without an X capability) then you cannot use `upshot`.

2. **Q:** When trying to run `upshot`, I get

   ```
   upshot: Command not found.
   ```

   A: First, check that `upshot` is in your path. You can use the command

```
which upshot
```

to do this.

If it is in your path, the problem may be that the name of the `wish` interpreter (used by `upshot`) is too long for your Unix system. Look at the first line of the '`upshot`' file. It should be something like

```
#! /usr/local/bin/wish -f
```

If it is something like

```
#! /usr/local/tcl7.4-tk4.2/bin/wish -f
```

this may be too long of a name (some Unix systems restrict this first line to a mere 32 characters). To fix this, you'll need to put a link to '`wish`' somewhere where the name will be short enough. Alternately, you can start `upshot` with

```
/usr/local/tcl7.4-tk4.2/bin/wish -f /usr/local/mpich/bin/upshot
```

9.7.2 HP-UX

1. **Q:** When trying to run `upshot` under HP-UX, I get error messages like

   ```
   set: Variable name must begin with a letter.
   ```

   or

   ```
   upshot: syntax error at line 35: `(' unexpected
   ```

   A: Your version of HP-UX limits the shell names to very short strings. `Upshot` is a program that is executed by the `wish` shell, and for some reason HP-UX is both refusing to execute in this shell and then trying to execute the `upshot` program using your current shell (e.g., '`sh`' or '`csh`'), instead of issuing a sensible error message about the command name being too long. There are two possible fixes:

   (a) Add a link with a much shorter name, for example

   ```
   ln -s /usr/local/tk3.6/bin/wish /usr/local/bin/wish
   ```

   Then edit the `upshot` script to use this shorter name instead. This may require root access, depending on where you put the link.

   (b) Create a regular shell program containing the lines

   ```
   #! /bin/sh
   /usr/local/tk3.6/bin/wish -f /usr/local/mpi/bin/upshot
   ```

   (with the appropriate names for both the '`wish`' and '`upshot`' executables).

Also, file a bug report with HP. At the very least, the error message here is wrong; also, there is no reason to restrict general shell choices (as opposed to login shells).

Appendices

A 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`'.

A.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`'.

B Options for `mpirun`

The options for `mpirun` as shown by `mpirun -help`, are (note that not all options are supported by all devices). Depending on the specific device, the output of `mpirun -help` may differ; the following is for the `globus2` device.

```
mpirun [mpirun_options...] <progname> [options...]

  mpirun_options:
    -arch <architecture>
            specify the architecture (must have matching machines.<arch>
            file in /usr/local/mpich/bin/machines) if using the execer
    -h      This help
    -machine <machine name>
            use startup procedure for <machine name>
mpirun [mpirun_options...] <progname> [options...]
```

```
mpirun_options:
  -arch <architecture>
          specify the architecture (must have matching machines.<arch>
          file in /usr/local/mpich/bin/machines) if using the execer
  -h      This help
  -machine <machine name>
          use startup procedure for <machine name>
          Currently supported:
            paragon
            p4
            sp1
            ibmspx
            anlspx
            sgi_mp
            ipsc860
            inteldelta
            cray_t3d
            execer
            smp
            symm_ptx
  -machinefile <machine-file name>
          Take the list of possible machines to run on from the
          file <machine-file name>. This is a list of all available
          machines; use -np <np> to request a specific number of machines.
  -np <np>
          specify the number of processors to run on
  -nodes <nodes>
          specify the number of nodes to run on (for SMP systems,
          currently only ch_mpl device supports this)
  -nolocal
          don't run on the local machine (only works for ch_p4 jobs)
  -all-cpus, -allcpus
          Use all available CPUs on all the nodes.
  -all-local
          Run all processes on the master node.
  -exclude <list>
          Exclude nodes in a colon delimited list.
  -map <list>
          Use the colon delimited list to specify which rank
          runs on which nodes.
  -stdin filename
          Use filename as the standard input for the program.  This
          is needed for programs that must be run as batch jobs, such
          as some IBM SP systems and Intel Paragons using NQS (see
          -paragontype below).
          use
                -stdin /dev/null
          if there is no input and you intend to run the program in the
          background.  An alternate is to redirect standard input from
          /dev/null, as in
                mpirun -np 4 a.out < /dev/null
  -t      Testing - do not actually run, just print what would be
          executed
```

```
-v      Verbose - throw in some comments
-dbg    The option '-dbg' may be used to select a debugger.  For example,
        -dbg=gdb invokes the mpirun_dbg.gdb script located in the
        'mpich/bin' directory.  This script captures the correct arguments,
        invokes the gdb debugger, and starts the first process under gdb
        where possible.  There are 4 debugger scripts; gdb, xxgdb, ddd,
        totalview.  These may need to be edited depending on your system.
        There is another debugger script for dbx, but this one will always
        need to be edited as the debugger commands for dbx varies between
        versions.  You can also use this option to call another debugger;
        for example, -dbg=mydebug.  All you need to do is write a script
        file, 'mpirun_dbg.mydebug', which follows the format of the
        included debugger scripts, and place it in the mpich/bin directory.
-ksq    Keep the send queue. This is useful if you expect later
        to attach totalview to the running (or deadlocked) job, and
        want to see the send queues. (Normally they are not maintained
        in a way which is visible to the debugger).

Options for the globus2 device:

With the exception of -h, these are the only mpirun options supported
by the globus device.

-machinefile <machine-file name>
        Take the list of possible machines to run on from the
        file <machine-file name>
-np <np>
        specify the number of processors to run on
-dumprsl - display the RSL string that would have been
          used to submit the job.  using this option
          does not run the job.
-globusrsl <globus-rsl-file name>
        <globus-rsl-file name> must contain a Globus RSL
        string.  When using this option all other mpirun options
        are ignored.


Special Options for Batch Environments:

-mvhome Move the executable to the home directory.  This
        is needed when all file systems are not cross-mounted
        Currently only used by anlspx
-mvback files
        Move the indicated files back to the current directory.
        Needed only when using -mvhome; has no effect otherwise.
-maxtime min
        Maximum job run time in minutes.  Currently used only
        by anlspx.  Default value is $max_time minutes.
-nopoll Do not use a polling-mode communication.
```

```
        Available only on IBM SPs.

Special Options for IBM SP2:

-cac name
        CAC for ANL scheduler.  Currently used only by anlspx.
        If not provided will choose some valid CAC.
```

On exit, `mpirun` returns a status of zero unless `mpirun` detected a problem, in which case it returns a non-zero status.

When using the `ch_p4` device, multiple architectures may be handled by giving multiple `-arch` and `-np` arguments. For example, to run a program on 2 sun4s and 3 rs6000s, with the local machine being a sun4, use

```
mpirun -arch sun4 -np 2 -arch rs6000 -np 3 program
```

This assumes that program will run on both architectures. If different executables are needed, the string '%a' will be replaced with the arch name. For example, if the programs are `program.sun4` and `program.rs6000`, then the command is

```
mpirun -arch sun4 -np 2 -arch rs6000 -np 3 program.%a
```

If instead the executables are in different directories; for example, '`/tmp/me/sun4`' and '`/tmp/me/rs6000`', then the command is

```
mpirun -arch sun4 -np 2 -arch rs6000 -np 3 /tmp/me/%a/program
```

It is important to specify the architecture with `-arch` *before* specifying the number of processors. Also, the *first* `arch` command must refer to the processor on which the job will be started. Specifically, if `-nolocal` is *not* specified, then the first `-arch` must refer to the processor from which mpirun is running.

C mpirun and Globus

In the section we describe how to run MPI programs using the `mpich globus2` device in a Globus-enabled distributed computing environment. It is assumed that (a) Globus has been installed and the appropriate Globus daemons are running on the machines you wish to launch your MPI application, (b) you have already acquired your Globus ID and security credentials, (c) your Globus ID has been registered on all machines, and (d) you have a valid (unexpired) Globus proxy. See `http://www.globus.org` for information on those topics.

Every `mpirun` command under the `globus2` device submits a Globus *Resource Specification Language Script*, or simply *RSL script*, to a Globus-enabled grid of computers. Each RSL script is composed of one or more RSL *subjobs*, typically one subjob for each machine

in the computation. You may supply your own RSL script[5] explicitly to `mpirun` (using the `-globusrsl <rslfilename>` option) in which case you would not specify any other options to `mpirun`, or you may have `mpirun` construct an RSL script for you based on the arguments you pass to `mpirun` and the contents of your *machines* file (discussed below). In either case it is important to remember *communication between nodes in different subjobs is always facilitated over TCP/IP and the more efficient vendor-supplied MPI is used only among nodes within the same subjob.*

C.1 Using `mpirun` To Construct An RSL Script For You

You would use this method if you wanted to launch a *single* executable file, which implies a set of one or more binary-compatible machines that all share the same filesystem (i.e., they can all access the executable file).

Using `mpirun` to construct an RSL script for you requires a *machines* file. The `mpirun` command determines which machines file to use as follows:

1. If a `-machinefile <machinefilename>` argument is specified on the `mpirun` command, it uses that; otherwise,
2. it looks for a file '`machines`' in the directory in which you typed `mpirun`; and finally,
3. it looks for '`/usr/local/mpich/bin/machines`' where '`/usr/local/mpich`' is the `mpich` installation directory.

If it cannot find a machines file from any of those places then `mpirun` fails.

The machines file is used to list the computers upon which you wish to run your application. Computers are listed by naming the Globus "service" on that machine. For most applications the default service can be used, which requires specifying only the fully qualified domain name. Consult your local Globus administrator or the Globus web site `www.globus.org` for more information regarding special Globus services. For example, consider the following pair of binary-compatible machines, {`m1,m2`}`.utech.edu`, that have access to the same filesystem. Here is what a machines file that uses default Globus services might look like.

```
"m1.utech.edu" 10
"m2.utech.edu" 5
```

The number appearing at the end of each line is optional (default=1). It specifies the maximum number of nodes that can be created in a single RSL subjob on each machine. `mpirun` uses the `-np` specification by "wrapping around" the machines file. For example, using the machines file above `mpirun -np 8` creates an RSL with a single subjob with 8 nodes on `m1.utech.edu`, while `mpirun -np 12` creates two subjobs where the first subjob has 10 nodes on `m1.utech.edu` and the second has 2 nodes on `m2.utech.edu`, and finally `mpirun -np 17` creates three subjobs with 10 nodes on `m1.utech.edu` followed by 5 nodes

[5] See `www.globus.org` for the syntax and semantics of the Globus Resource Specification Language.

on `m2.utech.edu` ending with the third a final subjob having two nodes on `m1.utech.edu` again. Note that inter-subjob messaging is *always* communicated over TCP, even if the two separate subjobs are the same machine.

C.1.1 Using `mpirun` By Supplying Your Own RSL Script

You would use `mpirun` supplying your own RSL script if you were submitting to a set of machines that could not run or access the same executable file (e.g., machines that are not binary compatible and/or do not share a file system). In this situation, we must currently use something called a *Resource Specification Language* (RSL) request to specify the executable filename for each machine. This technique is very flexible, but rather complex; work is currently underway to simplify the manner in which these issues are addressed.

The easiest way to learn how to write your own RSL request is to study the one generated for you by mpirun. Consider the example where we wanted to run an application on a cluster of workstations. Recall our `machines` file looked like this:

```
"m1.utech.edu" 10
"m2.utech.edu" 5
```

To view the RSL request generated in this situation, *without* actually launching the program, we type the following `mpirun` command:

```
% mpirun -dumprsl -np 12 myapp 123 456
```

which produces the following output:

```
+
( &(resourceManagerContact="m1.utech.edu")
   (count=10)
   (jobtype=mpi)
   (label="subjob 0")
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 0))
   (arguments=" 123 456")
   (directory=/homes/karonis/MPI/mpich.yukon/mpich/lib/IRIX64/globus)
   (executable=/homes/karonis/MPI/mpich.yukon/mpich/lib/IRIX64/globus/myapp)
)
( &(resourceManagerContact="m2.utech.edu")
   (count=2)
   (jobtype=mpi)
   (label="subjob 1")
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 1))
   (arguments=" 123 456")
   (directory=/homes/karonis/MPI/mpich.yukon/mpich/lib/IRIX64/globus)
   (executable=/homes/karonis/MPI/mpich.yukon/mpich/lib/IRIX64/globus/myapp)
)
```

Note that `(jobtype=mpi)` may appear only in those subjobs whose machines have vendor-supplied implementations of MPI. Additional environment variables may be added as in the example below:

```
+
( &(resourceManagerContact="m1.utech.edu")
   (count=10)
   (jobtype=mpi)
   (label="subjob 0")
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 0)
                (MY_ENV 246))
   (arguments=" 123 456")
   (directory=/homes/karonis/MPI/mpich.yukon/mpich/lib/IRIX64/globus)
   (executable=/homes/karonis/MPI/mpich.yukon/mpich/lib/IRIX64/globus/myapp)
)
( &(resourceManagerContact="m2.utech.edu")
   (count=2)
   (jobtype=mpi)
   (label="subjob 1")
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 1))
   (arguments=" 123 456")
   (directory=/homes/karonis/MPI/mpich.yukon/mpich/lib/IRIX64/globus)
   (executable=/homes/karonis/MPI/mpich.yukon/mpich/lib/IRIX64/globus/myapp)
)
```

After editing your own RSL file you may submit that directly to `mpirun` as follows:

```
% mpirun -globusrsl <myrslrequestfile>
```

Note that when supplying your own RSL it should be the *only* argument you specify to `mpirun`.

RSL is a flexible language capable of doing much more than has been presented here. For example, it can be used to stage executables and to set environment variables on remote computers before starting execution. A full description of the language can be found at `http://www.globus.org`.

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 `mpich`, particularly Patrick Bridges and Edward Karrels. Particular thanks goes to Nathan Doss and Anthony Skjellum for valuable help in the implementation and development of `mpich`. Debbie Swider, who worked with the `mpich` group for several years, helped support and enrich the `mpich` implementation. David Ashton has provided the Windows NT implementation of `mpich`, supported by a grant from the Microsoft coroporation. The Globus2 device was implemented by Nick Karonis of Northern Illinois University and Brian Toonen of Argonne National Laboratory. The C++ bindings were implemented by Andrew Lumsdaine and Jeff Squyres of the University of Notre Dame. The ROMIO MPI-2 parallel I/O subsystem was implemented by Rajeev Thakur of Argonne.

D Deprecated Features

During the development of `mpich`, various features were developed for the installation and use of `mpich`. Some of these have been superceeded by newer features that are described above. This section archives the documentation on the deprecated features.

D.1 More detailed control over compiling and linking

For more control over the process of compiling and linking programs for `mpich`, you should use a '`Makefile`'. Rather than modify your '`Makefile`' for each system, you can use a makefile template and use the command '`mpireconfig`' to convert the makefile template into a valid '`Makefile`'. To do this, start with the file '`Makefile.in`' in '`/usr/local/mpich/examples`'. Modify this '`Makefile.in`' for your program and then enter

```
mpireconfig Makefile
```

(not `mpireconfig Makefile.in`). This creates a '`Makefile`' from '`Makefile.in`'. Then enter:

```
make
```

References

[1] TotalView Multiprocess Debugger/Analyzer, 2000. http://www.etnus.com/Products/TotalView.

[2] Ralph Butler and Ewing Lusk. User's guide to the `p4` parallel programming system. Technical Report ANL-92/17, Argonne National Laboratory, Argonne, IL, 1992.

[3] James Cownie and William Gropp. A standard interface for debugger access to message queue information in MPI. In Jack Dongarra, Emilio Luque, and Tomàs Margalef, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1697 of *Lecture Notes in Computer Science*, pages 51–58. Springer Verlag, 1999. 6th European PVM/MPI Users' Group Meeting, Barcelona, Spain, September 1999.

[4] Message Passing Interface Forum. MPI: A message-passing interface standard. Computer Science Dept. Technical Report CS-94-230, University of Tennessee, Knoxville, TN, 1994.

[5] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI—The Complete Reference: Volume 2, The MPI-2 Extensions*. MIT Press, Cambridge, MA, 1998.

[6] William Gropp and Ewing Lusk. Installation guide for `mpich`, a portable implementation of MPI. Technical Report ANL-96/5, Argonne National Laboratory, 1996.

[7] William Gropp and Ewing Lusk. A high-performance MPI implementation on a shared-memory vector supercomputer. *Parallel Computing*, 22(11):1513–1526, January 1997.

[8] William Gropp and Ewing Lusk. Sowing MPICH: A case study in the dissemination of a portable environment for parallel scientific computing. *IJSA*, 11(2):103–114, Summer 1997.

[9] 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.

[10] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface,* 2nd edition. MIT Press, Cambridge, MA, 1999.

[11] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

[12] 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.

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

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

[15] Peter S. Pacheco. *Parallel Programming with MPI.* Morgan Kaufman, 1997.

[16] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI—The Complete Reference: Volume 1, The MPI Core,* 2nd edition. MIT Press, Cambridge, MA, 1998.

[17] 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.