

# **MPICH2 User's Guide\***

Version 1.0.3

Mathematics and Computer Science Division  
Argonne National Laboratory

William Gropp  
Ewing Lusk  
David Ashton  
Darius Buntinas  
Ralph Butler  
Anthony Chan  
Rob Ross  
Rajeev Thakur  
Brian Toonen

November 23, 2005

---

\*This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, SciDAC Program, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Migrating to MPICH2 from MPICH1</b>	<b>1</b>
2.1	Default Runtime Environment . . . . .	1
2.2	Starting Parallel Jobs . . . . .	2
2.3	Command-Line Arguments in Fortran . . . . .	2
2.4	Configure Options . . . . .	2
<b>3</b>	<b>Setting Paths</b>	<b>3</b>
<b>4</b>	<b>Quick Start</b>	<b>3</b>
<b>5</b>	<b>Compiling and Linking</b>	<b>4</b>
5.1	Specifying Compilers . . . . .	4
5.2	Shared Libraries . . . . .	4
5.3	Special Issues for C++ . . . . .	5
5.4	Special Issues for Fortran . . . . .	5
<b>6</b>	<b>Running Programs with mpiexec</b>	<b>5</b>
6.1	Standard mpiexec . . . . .	6
6.2	Extensions for All Process Management Environments . . . . .	6
6.3	Extensions for the MPD Process Management Environment . . . . .	7
6.3.1	Basic mpiexec arguments for MPD . . . . .	7
6.3.2	Other Command-Line Arguments to mpiexec . . . . .	8
6.3.3	Environment Variables Affecting mpiexec . . . . .	12
6.4	Extensions for SMPD . . . . .	12
6.4.1	mpiexec arguments for SMPD . . . . .	12

6.5	Extensions for gforker . . . . .	15
6.5.1	mpiexec arguments for gforker . . . . .	15
<b>7</b>	<b>Managing the Process Management Environment</b>	<b>15</b>
7.1	MPD . . . . .	15
<b>8</b>	<b>Debugging</b>	<b>16</b>
8.1	gdb via mpiexec . . . . .	16
8.2	TotalView . . . . .	20
<b>9</b>	<b>MPICH2 under Windows</b>	<b>20</b>
9.1	Directories . . . . .	20
9.2	Compiling . . . . .	20
9.3	Running . . . . .	21
<b>A</b>	<b>General Information</b>	<b>21</b>
<b>B</b>	<b>Building MPICH2</b>	<b>21</b>
<b>C</b>	<b>Compiling MPI Programs</b>	<b>21</b>
<b>D</b>	<b>Running MPI Programs</b>	<b>22</b>
D.1	Q: What is MPICH2? . . . . .	22
D.2	Q: What does MPICH stand for? . . . . .	22
D.3	Q: What is the difference between the MPD and SMPD process managers? . . . . .	22
D.4	Q: When I use the g95 Fortran compiler on a 64-bit platform, some of the tests fail . . . . .	23
D.5	C++ and SEEK_SET . . . . .	23

D.6	When building the ssm or sshm channel, I get the error "mpidu_process_locks.h:234:2: error: #error *** No atomic memory operation specified to implement busy locks ***" . . . . .	24
D.7	Q: How do I pass environment variables to the processes of my parallel program . . . . .	24
D.8	Q: How do I pass environment variables to the processes of my parallel program when using the mpd process manager? .	24
D.9	Q: What determines the hosts on which my MPI processes run?	25
D.10	Q: On Windows, I get an error when I attempt to call MPI_Comm_spawn.	26

## 1 Introduction

This manual assumes that MPICH2 has already been installed. For instructions on how to install MPICH2, see the *MPICH2 Installer's Guide*, or the **README** in the top-level MPICH2 directory. This manual explains how to compile, link, and run MPI applications, and use certain tools that come with MPICH2. This is a preliminary version and some sections are not complete yet. However, there should be enough here to get you started with MPICH2.

## 2 Migrating to MPICH2 from MPICH1

If you have been using MPICH 1.2.x (1.2.6 is the latest version), you will find a number of things about MPICH2 that are different (and hopefully better in every case.) Your MPI application programs need not change, of course, but a number of things about how you run them will be different.

MPICH2 is an all-new implementation of the MPI Standard, designed to implement all of the MPI-2 additions to MPI (dynamic process management, one-sided operations, parallel I/O, and other extensions) and to apply the lessons learned in implementing MPICH1 to make MPICH2 more robust, efficient, and convenient to use.

### 2.1 Default Runtime Environment

In MPICH1, the default configuration used the now-old **p4** portable programming environment. Processes were started via remote shell commands (**rsh** or **ssh**) and the information necessary for processes to find and connect with one another over sockets was collected and then distributed at startup time in a non-scalable fashion. Furthermore, the entanglement of process management functionality with the communication mechanism led to confusing behavior of the system when things went wrong.

MPICH2 provides a separation of process management and communication. The default runtime environment consists of a set of daemons, called **mpd**'s, that establish communication among the machines to be used before application process startup, thus providing a clearer picture of what is wrong when communication cannot be established and providing a fast

and scalable startup mechanism when parallel jobs are started. Section 7.1 describes the MPD process management system in more detail.

## 2.2 Starting Parallel Jobs

MPICH1 provided the `mpirun` command to start MPICH1 jobs. The MPI-2 Forum recommended a standard, portable command, called `mpiexec`, for this purpose. MPICH2 implements `mpiexec` and all of its standard arguments, together with some extensions. See Section 6.1 for standard arguments to `mpiexec` and various subsections of Section 6 for extensions particular to various process management systems.

MPICH2 also provides an `mpirun` command for simple backward compatibility, but MPICH2's `mpirun` does not provide all the options of `mpiexec` or all of the options of MPICH1's `mpirun`.

## 2.3 Command-Line Arguments in Fortran

MPICH1 (more precisely) MPICH1's `mpirun`) required access to command line arguments in all application programs, including Fortran ones, and MPICH1's `configure` devoted some effort to finding the libraries that contained the right versions of `iargc` and `getarg` and including those libraries with which the `mpif77` script linked MPI programs. Since MPICH2 does not require access to command line arguments to applications, these functions are optional, and `configure` does nothing special with them. If you need them in your applications, you will have to ensure that they are available in the Fortran environment you are using.

## 2.4 Configure Options

The arguments to `configure` are different in MPICH1 and MPICH2; the `Installer's Guide` discusses `configure`. In particular, the newer `configure` in MPICH2 does not support the `-cc=<compiler-name>` (or `-fc`, `-c++`, or `-f90`) options. Instead, many of the items that could be specified in the command line to configure in MPICH1 must now be set by defining an environment variable. E.g., while MPICH1 allowed

```
./configure -cc=pgcc
```

MPICH2 requires

```
setenv CC pgcc
```

(or `export CC=pgcc` for `ksh` or `CC=pgcc ; export CC` for strict `sh`) before `./configure`. Basically, every option to the MPICH-1 `configure` that does not start with `--enable` or `--with` is not available as a `configure` option in MPICH2. Instead, environment variables must be used. This is consistent (and required) for use of version 2 GNU `autoconf`.

### 3 Setting Paths

You will have to know the directory where MPICH2 has been installed. (Either you installed it there yourself, or your systems administrator has installed it. One place to look in this case might be `/usr/local`.) We suggest that you put the `bin` subdirectory of that directory in your path. This will give you access to assorted MPICH2 commands to compile, link, and run your programs conveniently. Other commands in this directory manage parts of the run-time environment and execute tools.

One of the first commands you might run is `mpich2version` to find out the exact version and configuration of MPICH2 you are working with. Some of the material in this manual depends on just what version of MPICH2 you are using and how it was configured at installation time.

### 4 Quick Start

You should now be able to run an MPI program. Let us assume that the directory where MPICH2 has been installed is `/home/you/mpich2-installed`, so that in the section above you did

```
setenv PATH /home/you/mpich2-installed/bin:$PATH
```

for `tcsh` and `csh`, or

```
export PATH=/home/you/mpich2-installed/bin:$PATH
```

for `bash` or `sh`. Then to run an MPI program, albeit only on one machine, you can do:

```
mpd &  
cd /home/you/mpich2-installed/examples  
mpiexec -n 3 cpi  
mpdallexit
```

Details for these commands are provided below, but if you can successfully execute them here, then you have a correctly installed MPICH2 and have run an MPI program.

## 5 Compiling and Linking

A convenient way to compile and link your program is by using scripts that use the same compiler that MPICH2 was built with. These are `mpicc`, `mpicxx`, `mpif77`, and `mpif90`, for C, C++, Fortran 77, and Fortran 90 programs, respectively. If any of these commands are missing, it means that MPICH2 was configured without support for that particular language.

### 5.1 Specifying Compilers

You need not use the same compiler that MPICH2 was built with, but not all compilers are compatible. You can also specify the compiler for building MPICH2 itself, as reported by `mpich2version` just by using the compiling and linking commands from the previous section. The environment variables `MPICH_CC`, `MPICH_CXX`, `MPICH_F77`, and `MPICH_F90` may be used to specify alternate C, C++, Fortran 77, and Fortran 90 compilers, respectively.

### 5.2 Shared Libraries

Currently shared libraries are only tested on Linux, and there are restrictions. See the *Installer's Guide* for how to build MPICH2 as a shared library. If shared libraries have been built, you will get them automatically when you link your program with any of the MPICH2 compilation scripts.

### 5.3 Special Issues for C++

Some users may get error messages such as

```
SEEK_SET is #defined but must not be for the C++ binding of MPI
```

The problem is that both `stdio.h` and the MPI C++ interface use `SEEK_SET`, `SEEK_CUR`, and `SEEK_END`. This is really a bug in the MPI-2 standard. You can try adding

```
#undef SEEK_SET
#undef SEEK_END
#undef SEEK_CUR
```

before `mpi.h` is included, or add the definition

```
-DMPICH_IGNORE_CXX_SEEK
```

to the command line (this will cause the MPI versions of `SEEK_SET` etc. to be skipped).

### 5.4 Special Issues for Fortran

MPICH2 provides two kinds of support for Fortran programs. For Fortran 77 programmers, the file `mpif.h` provides the definitions of the MPI constants such as `MPI_COMM_WORLD`. Fortran 90 programmers should use the `MPI` module instead; this provides all of the definitions as well as interface definitions for many of the MPI functions. However, this `MPI` module does not provide full Fortran 90 support; in particular, interfaces for the routines, such as `MPI_Send`, that take “choice” arguments are not provided.

## 6 Running Programs with `mpiexec`

If you have been using the original `MPICH`, or any of a number of other MPI implementations, then you have probably been using `mpirun` as a way to start your MPI programs. The MPI-2 Standard describes `mpiexec` as a suggested way to run MPI programs. `MPICH2` implements the `mpiexec`

standard, and also provides some extensions. MPICH2 provides `mpirun` for backward compatibility with existing scripts, but it does not support the same or as many options as `mpiexec` or all of the options of MPICH1's `mpirun`.

## 6.1 Standard `mpiexec`

Here we describe the standard `mpiexec` arguments from the MPI-2 Standard [1]. The simplest form of a command to start an MPI job is

```
mpiexec -n 32 a.out
```

to start the executable `a.out` with 32 processes (providing an `MPI_COMM_WORLD` of size 32 inside the MPI application). Other options are supported, for specifying hosts to run on, search paths for executables, working directories, and even a more general way of specifying a number of processes. Multiple sets of processes can be run with different executables and different values for their arguments, with “:” separating the sets of processes, as in:

```
mpiexec -n 1 -host loginnode master : -n 32 -host smp slave
```

The `-configfile` argument allows one to specify a file containing the specifications for process sets on separate lines in the file. This makes it unnecessary to have long command lines for `mpiexec`. (See p. 353 of [2].)

It is also possible to start a one process MPI job (with size of `MPI_COMM_WORLD` equal to 1), without using `mpiexec`. This process will become an MPI process when it calls `MPI_Init`, and can then call other MPI functions, including `MPI_Comm_spawn`.

## 6.2 Extensions for All Process Management Environments

Some `mpiexec` arguments are specific to particular communication subsystems (“devices”) or process management environments (“process managers”). Our intention is to make all arguments as uniform as possible across devices and process managers. For the time being we will document these separately.

### 6.3 Extensions for the MPD Process Management Environment

MPICH2 provides a number of process management systems. The default is called MPD. MPD provides a number of extensions to the standard form of `mpiexec`.

#### 6.3.1 Basic `mpiexec` arguments for MPD

The default configuration of MPICH2 chooses the MPD process manager and the “simple” implementation of the Process Management Interface. MPD provides a version of `mpiexec` that supports both the standard arguments described in Section 6.1 and other arguments described in this section. MPD also provides a number of commands for querying the MPD process management environment and interacting with jobs it has started.

Before running `mpiexec`, the runtime environment must be established. In the case of MPD, the daemons must be running. See Section 7.1 for how to run and manage the MPD daemons.

We assume that the MPD ring is up and the installation’s `bin` directory is in your path; that is, you can do:

```
mpdtrace
```

and it will output a list of nodes on which you can run MPI programs. Now you are ready to run a program with `mpiexec`. Let us assume that you have compiled and linked the program `cpi` (in the `installdir/examples` directory and that this directory is in your `PATH`. Or that is your current working directory and `‘.’` (“dot”) is in your `PATH`. The simplest thing to do is

```
mpiexec -n 5 cpi
```

to run `cpi` on five nodes. The process management system (such as MPD) will choose machines to run them on, and `cpi` will tell you where each is running.

You can use `mpiexec` to run non-MPI programs as well. This is sometimes useful in making sure all the machines are up and ready for use. Useful examples include

```
mpiexec -n 10 hostname
```

and

```
mpiexec -n 10 printenv
```

### 6.3.2 Other Command-Line Arguments to mpiexec

The MPI-2 standard specifies the syntax and semantics of the arguments `-n`, `-path`, `-wdir`, `-host`, `-file`, `-configfile`, and `-soft`. All of these are currently implemented for MPD's `mpiexec`. Each of these is what we call a “local” option, since its scope is the processes in the set of processes described between colons, or on separate lines of the file specified by `-configfile`. We add some extensions that are local in this way and some that are “global” in the sense that they apply to all the processes being started by the invocation of `mpiexec`.

The MPI-2 Standard provides a way to pass different arguments to different application processes, but does not provide a way to pass environment variables. MPICH2 provides an extension that supports environment variables. The local parameter `-env` does this for one set of processes. That is,

```
mpiexec -n 1 -env FOO BAR a.out : -n 2 -env BAZZ FAZZ b.out
```

makes `BAR` the value of environment variable `FOO` on the first process, running the executable `a.out`, and gives the environment variable `BAZZ` the value `FAZZ` on the second two processes, running the executable `b.out`. To set an environment variable without giving it a value, use `''` as the value in the above command line.

The global parameter `-genv` can be used to pass the same environment variables to all processes. That is,

```
mpiexec -genv FOO BAR -n 2 a.out : -n 4 b.out
```

makes `BAR` the value of the environment variable `FOO` on all six processes. If `-genv` appears, it must appear in the first group. If both `-genv` and `-env` are used, the `-env`'s add to the environment specified or added to by the

`-genv` variables. If there is only one set of processes (no “:”), the `-genv` and `-env` are equivalent.

The local parameter `-envall` is an abbreviation for passing the entire environment in which `mpiexec` is executed. The global version of it is `-genvall`. This global version is implicitly present. To pass no environment variables, use `-envnone` and `-genvnone`. So, for example, to set *only* the environment variable `FOO` and no others, regardless of the current environment, you would use

```
mpiexec -genvnone -env FOO BAR -n 50 a.out
```

A list of environment variable names whose values are to be copied from the current environment can be given with the `-envlist` (respectively, `-genvlist`) parameter; for example,

```
mpiexec -genvnone -envlist PATH,LD_SEARCH_PATH -n 50 a.out
```

sets the `PATH` and `LD_LIBRARY_PATH` in the environment of the `a.out` processes to their values in the environment where `mpiexec` is being run. In this situation you can't have commas in the environment variable names, although of course they are permitted in values.

Some extension parameters have only global versions. They are

`-l` provides rank labels for lines of `stdout` and `stderr`. These are a bit obscure for processes that have been explicitly spawned, but are still useful.

`-usize` sets the “universe size” that is retrieved by the MPI attribute `MPI_UNIVERSE_SIZE` on `MPI_COMM_WORLD`.

`-bnr` is used when one wants to run executables that have been compiled and linked using the `ch_p4mpd` or `myrinet` device in `MPICH1`. The MPD process manager provides backward compatibility in this case.

`-machinefile` can be used to specify information about each of a set of machines. This information may include the number of processes to run on each host when executing user programs. For example, assume that a machinefile named `mf` contains:

```

# comment line
hosta
hostb:2
hostc   ifhn=hostc-gige
hostd:4  ifhn=hostd-gige

```

In addition to specifying hosts and number of processes to run on each, this machinefile indicates that processes running on `hostc` and `hostd` should use the `gige` interface on `hostc` and `hostd` respectively for MPI communications. (`ifhn` stands for “interface host name” and should be set to an alternate host name for the machine that is used to designate an alternate communication interface.) This interface information causes the MPI implementation to choose the alternate host name when making connections. When the alternate hostname specifies a particular interface, MPICH communication will then travel over that interface.

You might use this machinefile in the following way:

```
mpiexec -machinefile mf -n 7 p0
```

Process rank 0 is to run on `hosta`, ranks 1 and 2 on `hostb`, rank 3 on `hostc`, and ranks 4-6 on `hostd`. Note that the file specifies information for up to 8 ranks and we only used 7. That is OK. But, if we had used “-n 9”, an error would be raised. The file is not used as a pool of machines that are cycled through; the processes are mapped to the hosts in the order specified in the file.

A more complex command-line example might be:

```
mpiexec -l -machinefile mf -n 3 p1 : -n 2 p2 : -n 2 p3
```

Here, ranks 0-2 all run program `p1` and are executed placing rank 0 on `hosta` and ranks 1-2 on `hostb`. Similarly, ranks 3-4 run `p2` and are executed on `hostc` and `hostd`, respectively. Ranks 5-6 run on `hostd` and execute `p3`.

`-s` can be used to direct the `stdin` of `mpiexec` to specific processes in a parallel job. For example:

```
mpiexec -s all -n 5 a.out
```

directs the `stdin` of `mpiexec` to all five processes.

```
mpiexec -s 4 -n 5 a.out
```

directs it to just the process with rank 4, and

```
mpiexec -s 1,3 -n 5 a.out
```

sends it to processes 1 and 3, while

```
mpiexec -s 0-3 -n 5 a.out
```

sends `stdin` to processes 0,1,2, and 3.

The default, if `-s` is not specified, is to send `mpiexec`'s `stdin` to process 0 only.

`-kx` is used only for debugging. The `mpd` process manager encapsulates the command-line arguments, the contents of the `-machinefile` argument, `-configfile`, and in some cases the environment, into an XML file for delivery to the internals of the process manager. Under ordinary circumstances, this file, created in `/tmp`, is not seen by the user, and is deleted after use. In some cases it may be desirable to examine the contents of this file after it is used, in order to debug difficulties with the installation and functioning of `mpd`. In this case, the `-kx` argument causes `mpiexec` to keep the file, which you will be able to find in `/tmp` with a name like `smith_tempxml.1234`.

A “:” can optionally be used between global args and normal argument sets, e.g.:

```
mpiexec -l -n 1 -host host1 pgm1 : -n 4 -host host2 pgm2
```

is equivalent to:

```
mpiexec -l : -n 1 -host host1 pgm1 : -n 4 -host host2 pgm2
```

This option implies that the global arguments can occur on a separate line in the file specified by `-configfile` when it is used to replace a long command line.

### 6.3.3 Environment Variables Affecting `mpiexec`

A small number of environment variables affect the behavior of `mpiexec`.

`MPIEXEC_TIMEOUT` The value of this environment variable is the maximum number of seconds this job will be permitted to run. When time is up, the job is aborted.

`MPIEXEC_BNR` If this environment variable is defined (its value, if any, is currently insignificant), then MPD will act in backward-compatibility mode, supporting the BNR interface from the original MPICH (e.g. versions 1.2.0 – 1.2.6) instead of its native PMI interface, as a way for application processes to interact with the process management system.

`MPD_CON_EXT` Adds a string to the default Unix socket name used by `mpiexec` to find the local `mpd`. This allows one to run multiple `mpd` rings at the same time.

## 6.4 Extensions for SMPD

SMPD is an alternate process manager that runs on both Unix and Windows. It can launch jobs across both platforms if the binary formats match (big/little endianness and size of C types– `int`, `long`, `void*`, etc).

### 6.4.1 `mpiexec` arguments for SMPD

`mpiexec` for `smPd` accepts the standard MPI-2 `mpiexec` options. Execute

```
mpiexec
```

or

```
mpiexec -help2
```

to print the usage options. Typical usage:

```
mpiexec -n 10 myapp.exe
```

All options to `mpiexec`:

`-n x`

`-np x`

launch `x` processes

`-localonly x`

`-np x -localonly`

launch `x` processes on the local machine

`-machinefile filename`

use a file to list the names of machines to launch on

`-host hostname`

launch on the specified host.

`-hosts n host1 host2 ... hostn`

`-hosts n host1 m1 host2 m2 ... hostn mn`

launch on the specified hosts. In the second version the number of processes =  $m1 + m2 + \dots + mn$

`-dir drive:\my\working\directory`

`-wdir /my/working/directory`

launch processes with the specified working directory. (`-dir` and `-wdir` are equivalent)

`-env var val`

set environment variable before launching the processes

`-exitcodes`

print the process exit codes when each process exits.

`-noprompt`

prevent `mpiexec` from prompting for user credentials. Instead errors will be printed and `mpiexec` will exit.

`-localroot`

launch the root process directly from `mpiexec` if the host is local. (This allows the root process to create windows and be debugged.)

`-port port`

- `-p port`  
specify the port that `smpd` is listening on.
- `-phrase passphrase`  
specify the passphrase to authenticate connections to `smpd` with.
- `-smpdfile filename`  
specify the file where the `smpd` options are stored including the passphrase.  
(unix only option)
- `-path search_path`  
search path for executable, ; separated
- `-timeout seconds`  
timeout for the job.

Windows specific options:

- `-map drive:\\host\share`  
map a drive on all the nodes this mapping will be removed when the processes exit
- `-logon`  
prompt for user account and password
- `-pwdfile filename`  
read the account and password from the file specified.  
put the account on the first line and the password on the second
- `-nopopup_debug`  
disable the system popup dialog if the process crashes
- `-priority class[:level]`  
set the process startup priority class and optionally level.  
class = 0,1,2,3,4 = idle, below, normal, above, high  
level = 0,1,2,3,4,5 = idle, lowest, below, normal, above, highest  
the default is `-priority 2:3`
- `-register`  
encrypt a user name and password to the Windows registry.
- `-remove`  
delete the encrypted credentials from the Windows registry.

- `-validate [-host hostname]`  
validate the encrypted credentials for the current or specified host.
- `-delegate`  
use passwordless delegation to launch processes.
- `-impersonate`  
use passwordless authentication to launch processes.
- `-plaintext`  
don't encrypt the data on the wire.

## 6.5 Extensions for gforker

`gforker` is a process management system for starting processes on a single machine, so called because the MPI processes are simply `forked` from the `mpiexec` process.

### 6.5.1 `mpiexec` arguments for `gforker`

The argument `-maxtime` sets a maximum time in seconds for the job to run. `gforker` also supports all of the MPI-2 standard arguments for `mpiexec` as well as the extensions for environment variables described in Section 6.3.2.

## 7 Managing the Process Management Environment

Some of the process managers supply user commands that can be used to interact with the process manager and to control jobs. In this section we describe user commands that may be useful.

### 7.1 MPD

`mpd` starts an `mpd` daemon.

`mpdboot` starts a set of `mpd`'s on a list of machines.

`mpdtrace` lists all the MPD daemons that are running. The `-l` option lists full hostnames and the port where the `mpd` is listening.

`mpdlistjobs` lists the jobs that the mpd's are running. Jobs are identified by the name of the mpd where they were submitted and a number.

`mpdkilljob` kills a job specified by the name returned by `mpdlistjobs`

`mpdsigjob` delivers a signal to the named job. Signals are specified by name or number.

You can use keystrokes to provide signals in the usual way, where `mpiexec` stands in for the entire parallel application. That is, if `mpiexec` is being run in a Unix shell in the foreground, you can use `^C` (control-C) to send a `SIGINT` to the processes, or `^Z` (control-Z) to suspend all of them. A suspended job can be continued in the usual way.

Precise argument formats can be obtained by passing any MPD command the `--help` or `-h` argument. More details can be found in the `README` in the `mpich2` top-level directory or the `README` file in the MPD directory `mpich2/src/pm/mpd`.

## 8 Debugging

Debugging parallel programs is notoriously difficult. Here we describe a number of approaches, some of which depend on the exact version of `MPICH2` you are using.

### 8.1 gdb via mpiexec

If you are using the MPD process manager, you can use the `-gdb` argument to `mpiexec` to execute a program with each process running under the control of the `gdb` sequential debugger. `-gdb` helps control the multiple instances of `gdb` by sending `stdin` either to all processes or to a selected process and by labeling and merging output. The following script of a `-gdb` session gives an idea of how this works. Input keystrokes are sent to all processes unless specifically directed by the “z” command.

```
ksl2% mpiexec -gdb -n 10 cpi
0-9: (gdb) l
0-9: 5 double f(double);
0-9: 6
```

```
0-9: 7 double f(double a)
0-9: 8 {
0-9: 9     return (4.0 / (1.0 + a*a));
0-9: 10 }
0-9: 11
0-9: 12     int main(int argc, char *argv[])
0-9: 13     {
0-9: 14         int done = 0, n, myid, numprocs, i;
0-9: (gdb)
0-9: 15         double PI25DT = 3.141592653589793238462643;
0-9: 16         double mypi, pi, h, sum, x;
0-9: 17         double startwtime = 0.0, endwtime;
0-9: 18         int namelen;
0-9: 19         char processor_name[MPI_MAX_PROCESSOR_NAME];
0-9: 20
0-9: 21         MPI_Init(&argc, &argv);
0-9: 22         MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
0-9: 23         MPI_Comm_rank(MPI_COMM_WORLD, &myid);
0-9: 24         MPI_Get_processor_name(processor_name, &namelen);
0-9: (gdb)
0-9: 25
0-9: 26         fprintf(stdout, "Process %d of %d is on %s\n",
0-9: 27                 myid, numprocs, processor_name);
0-9: 28         fflush(stdout);
0-9: 29
0-9: 30         n = 10000;          /* default # of rectangles */
0-9: 31         if (myid == 0)
0-9: 32             startwtime = MPI_Wtime();
0-9: 33
0-9: 34         MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
0-9: (gdb) b 30
0-9: Breakpoint 2 at 0x4000000000002541:
0-9:         file /home/lusk/mpich2/examples/cpi.c, line 30.
0-9: (gdb) r
0-9: Continuing.
0: Process 0 of 10 is on ks12
1: Process 1 of 10 is on ks12
2: Process 2 of 10 is on ks12
3: Process 3 of 10 is on ks12
4: Process 4 of 10 is on ks12
5: Process 5 of 10 is on ks12
6: Process 6 of 10 is on ks12
7: Process 7 of 10 is on ks12
8: Process 8 of 10 is on ks12
9: Process 9 of 10 is on ks12
```

```

0-9:
0-9: Breakpoint 2, main (argc=1, argv=0x60000fffffb4b8)
0-9:   at /home/lusk/mpich2/examples/cpi.c:30
0-9: 30         n = 10000;          * default # of rectangles */
0-9: (gdb) n
0-9: 31         if (myid == 0)
0-9: (gdb) n
0: 32         startwtime = MPI_Wtime();
1-9: 34         MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
0-9: (gdb) z 0
0: (gdb) n
0: 34         MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
0: (gdb) z
0-9: (gdb) where
0-9: #0 main (argc=1, argv=0x60000fffffb4b8)
0-9:   at /home/lusk/mpich2/examples/cpi.c:34
0-9: (gdb) n
0-9: 36         h = 1.0 / (double) n;
0-9: (gdb)
0-9: 37         sum = 0.0;
0-9: (gdb)
0-9: 39         for (i = myid + 1; i <= n; i += numprocs)
0-9: (gdb)
0-9: 41             x = h * ((double)i - 0.5);
0-9: (gdb)
0-9: 42             sum += f(x);
0-9: (gdb)
0-9: 39         for (i = myid + 1; i <= n; i += numprocs)
0-9: (gdb)
0-9: 41             x = h * ((double)i - 0.5);
0-9: (gdb)
0-9: 42             sum += f(x);
0-9: (gdb)
0-9: 39         for (i = myid + 1; i <= n; i += numprocs)
0-9: (gdb)
0-9: 41             x = h * ((double)i - 0.5);
0-9: (gdb)
0-9: 42             sum += f(x);
0-9: (gdb)

```

```

0-9: 39          for (i = myid + 1; i <= n; i += numprocs)
0-9: (gdb)
0-9: 41          x = h * ((double)i - 0.5);
0-9: (gdb)
0-9: 42          sum += f(x);
0-9: (gdb)
0-9: 39          for (i = myid + 1; i <= n; i += numprocs)
0-9: (gdb)
0-9: 41          x = h * ((double)i - 0.5);
0-9: (gdb)
0-9: 42          sum += f(x);
0-9: (gdb) p sum
0: $1 = 19.999875951497799
1: $1 = 19.999867551672725
2: $1 = 19.999858751863549
3: $1 = 19.999849552071328
4: $1 = 19.999839952297158
5: $1 = 19.999829952542203
6: $1 = 19.999819552807658
7: $1 = 19.999808753094769
8: $1 = 19.999797553404832
9: $1 = 19.999785953739192
0-9: (gdb) c
0-9: Continuing.
0: pi is approximately 3.1415926544231256, Error is 0.000000008333325
1-9:
1-9: Program exited normally.
1-9: (gdb) 0: wall clock time = 44.909412
0:
0: Program exited normally.
0: (gdb) q
0-9: MPIGDB ENDING
ksl2%

```

You can attach to a running job with

```
mpiexec -gdba <jobid>
```

where <jobid> comes from `mpdlistjobs`.

## 8.2 TotalView

MPICH2 supports use of the TotalView debugger from Etnus. If `mpich` has been configured to enable debugging with TotalView (See the section on configuration of the `mpd` process manager in the *Installer's Guide*) then one can debug an MPI program started with `mpd` by adding `-tv` to the global `mpiexec` arguments, as in

```
mpiexec -tv -n 3 cpi
```

You will get a popup window from TotalView asking whether you want to start the job in a stopped state. If so, when the TotalView window appears, you may see assembly code in the source window. Click on `main` in the stack window (upper left) to see the source of the `main` function. TotalView will show that the program (all processes) are stopped in the call to `MPI_Init`.

## 9 MPICH2 under Windows

### 9.1 Directories

The default installation of MPICH2 is in `C:\Program Files\MPICH2`. Under the installation directory are three sub-directories: `include`, `bin`, and `lib`. The `include` and `lib` directories contain the header files and libraries necessary to compile MPI applications. The `bin` directory contains the process manager, `smpd.exe`, and the MPI job launcher, `mpiexec.exe`. The dlls that implement MPICH2 are copied to the Windows `system32` directory.

### 9.2 Compiling

The libraries in the `lib` directory were compiled with MS Visual C++ .NET 2003 and Intel Fortran 8.1. These compilers and any others that can link with the MS `.lib` files can be used to create user applications. `gcc` and `g77` for `cygwin` can be used with the `libmpich*.a` libraries.

For MS Developer Studio users: Create a project and add

```
C:\Program Files\MPICH2\include
```

to the include path and

```
C:\Program Files\MPICH2\lib
```

to the library path. Add `mpi.lib` and `cxx.lib` to the link command. Add `cxxd.lib` to the Debug target link instead of `cxx.lib`.

Intel Fortran 8 users should add `fmpich2.lib` to the link command.

Cygwin users should use `libmpich2.a` `libfmpich2g.a`.

### 9.3 Running

MPI jobs are run from a command prompt using `mpiexec.exe`. See Section 6.4 on `mpiexec` for `smpd` for a description of the options to `mpiexec`.

Frequently Asked Questions This is the content of the online FAQ, as of November 7, 2005.

## A General Information

- Q: What is MPICH2?
- Q: What does MPICH stand for?

## B Building MPICH2

- Q: What is the difference between the MPD and SMPD process managers?
- Q: When I use the g95 Fortran compiler on a 64-bit platform, some of the tests fail

## C Compiling MPI Programs

- C++ and `SEEK_SET`

- When building the ssm or sshm channel, I get the error "mpidu\_process\_locks.h:234:2: error: #error \*\*\* No atomic memory operation specified to implement busy locks \*\*\*"

## D Running MPI Programs

- Q: How do I pass environment variables to the processes of my parallel program
- Q: How do I pass environment variables to the processes of my parallel program when using the mpd process manager?
- Q: What determines the hosts on which my MPI processes run?
- Q: On Windows, I get an error when I attempt to call `MPI_Comm_spawn`.

### D.1 Q: What is MPICH2?

MPICH2 is a freely available, portable implementation of MPI, the Standard for message-passing libraries. It implements both MPI-1 and MPI-2.

### D.2 Q: What does MPICH stand for?

A: MPI stands for Message Passing Interface. The CH comes from Chameleon, the portability layer used in the original MPICH to provide portability to the existing message-passing systems.

### D.3 Q: What is the difference between the MPD and SMPD process managers?

MPD is the default process manager for MPICH2 on Unix platforms. It is written in Python. SMPD is the primary process manager for MPICH2 on Windows. It is also used for running on a combination of Windows and Linux machines. It is written in C.

#### D.4 Q: When I use the g95 Fortran compiler on a 64-bit platform, some of the tests fail

A: The g95 compiler incorrectly defines the default Fortran integer as a 64-bit integer while defining Fortran reals as 32-bit values (the Fortran standard requires that INTEGER and REAL be the same size). This was apparently done to allow a Fortran INTEGER to hold the value of a pointer, rather than requiring the programmer to select an INTEGER of a suitable KIND. To force the g95 compiler to correctly implement the Fortran standard, use the `-i4` flag. For example, set the environment variable `F90FLAGS` before configuring `MPICH2`:

```
setenv F90FLAGS "-i4"
```

G95 users should note that there (at this writing) are two distributions of g95 for 64-bit Linux platforms. One uses 32-bit integers and reals (and conforms to the Fortran standard) and one uses 32-bit integers and 64-bit reals. We recommend using the one that conforms to the standard (note that the standard specifies the *ratio* of sizes, not the absolute sizes, so a Fortran 95 compiler that used 64 bits for *both* INTEGER and REAL would also conform to the Fortran standard. However, such a compiler would need to use 128 bits for DOUBLE PRECISION quantities).

#### D.5 C++ and SEEK\_SET

Some users may get error messages such as

```
SEEK_SET is #defined but must not be for the C++ binding of MPI
```

The problem is that both `stdio.h` and the MPI C++ interface use `SEEK_SET`, `SEEK_CUR`, and `SEEK_END`. This is really a bug in the MPI-2 standard. You can try adding

```
#undef SEEK_SET
#undef SEEK_END
#undef SEEK_CUR
```

before `mpi.h` is included, or add the definition

```
-DMPICH_IGNORE_CXX_SEEK
```

to the command line (this will cause the MPI versions of `SEEK_SET` etc. to be skipped).

**D.6 When building the ssm or sshm channel, I get the error "mpidu\_process\_locks.h:234:2: error: #error \*\*\* No atomic memory operation specified to implement busy locks \*\*\*"**

The ssm and sshm channels do not work on all platforms because they use special interprocess locks (often assembly) that may not work with some compilers or machine architectures. They work on Linux with gcc, Intel, and Pathscale compilers on various Intel architectures. They also work in Windows and Solaris environments.

**D.7 Q: How do I pass environment variables to the processes of my parallel program**

A: The specific method depends on the process manager and version of `mpiexec` that you are using.

**D.8 Q: How do I pass environment variables to the processes of my parallel program when using the mpd process manager?**

A: By default, all the environment variables in the shell where `mpiexec` is run are passed to all processes of the application program. (The one exception is `LD_LIBRARY_PATH` when the mpd's are being run as root.) This default can be overridden in many ways, and individual environment variables can be passed to specific processes using arguments to `mpiexec`. A synopsis of the possible arguments can be listed by typing

```
mpiexec -help
```

and further details are available in the Users Guide.

### D.9 Q: What determines the hosts on which my MPI processes run?

A: Where processes run, whether by default or by specifying them yourself, depends on the process manager being used.

If you are using the `gforker` process manager, then all MPI processes run on the same host where you are running `mpiexec`.

If you are using the `mpd` process manager, which is the default, then many options are available. If you are using `mpd`, then before you run `mpiexec`, you will have started, or will have had started for you, a ring of processes called `mpd`'s (multi-purpose daemons), each running on its own host. It is likely, but not necessary, that each `mpd` will be running on a separate host. You can find out what this ring of hosts consists of by running the program `mpdtrace`. One of the `mpd`'s will be running on the "local" machine, the one where you will run `mpiexec`. The default placement of MPI processes, if one runs

```
mpiexec -n 10 a.out
```

is to start the first MPI process (rank 0) on the local machine and then to distribute the rest around the `mpd` ring one at a time. If there are more processes than `mpd`'s, then wraparound occurs. If there are more `mpd`'s than MPI processes, then some `mpd`'s will not run MPI processes. Thus any number of processes can be run on a ring of any size. While one is doing development, it is handy to run only one `mpd`, on the local machine. Then all the MPI processes will run locally as well.

The first modification to this default behavior is the `-1` option to `mpiexec` (not a great argument name). If `-1` is specified, as in

```
mpiexec -1 -n 10 a.out
```

then the first application process will be started by the first `mpd` in the ring *after* the local host. (If there is only one `mpd` in the ring, then this will be on the local host.) This option is for use when a cluster of compute nodes has a "head node" where commands like `mpiexec` are run but not application processes.

If an `mpd` is started with the `--ncpus` option, then when it is its turn to start a process, it will start several application processes rather than just

one before handing off the task of starting more processes to the next `mpd` in the ring. For example, if the `mpd` is started with

```
mpd --ncpus=4
```

then it will start as many as four application processes, with consecutive ranks, when it is its turn to start processes. This option is for use in clusters of SMP's, when the user would like consecutive ranks to appear on the same machine. (In the default case, the same number of processes might well run on the machine, but their ranks would be different.)

(A feature of the `-ncpus=n` argument is that it has the above effect only until all of the `mpd`'s have started *n* processes at a time once; afterwards each `mpd` starts one process at a time. This is in order to balance the number of processes per machine to the extent possible.)

Other ways to control the placement of processes are by direct use of arguments to `mpiexec`. See the Users Guide.

**D.10 Q: On Windows, I get an error when I attempt to call `MPI_Comm_spawn`.**

A: On Windows, you need to start the program with `mpiexec` for any of the MPI-2 dynamic process functions to work.

## References

- [1] Message Passing Interface Forum. MPI2: A Message Passing Interface standard. *International Journal of High Performance Computing Applications*, 12(1-2):1-299, 1998.
- [2] 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.