MPICH2 User's Guide*

Version 1.1.0a1

Mathematics and Computer Science Division
Argonne National Laboratory

William Gropp
Ewing Lusk
David Ashton
Pavan Balaji
Darius Buntinas
Ralph Butler
Anthony Chan
David Goodell
Jayesh Krishna
Guillaume Mercier
Rob Ross
Rajeev Thakur
Brian Toonen

August 11, 2008

*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 DE-AC02-06CH11357.

Contents

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 (1.2.7p1 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. The *MPICH2 Installer's Guide* provides some information on changes between MPICH1 and MPICH2 to the process of configuring and installing MPICH. Changes to compiling, linking, and running MPI programs between MPICH1 and MPICH2 are described below.

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 managment 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 6.1 describes the MPD process management system in more detail. Other process managers are also available.

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 5.1 for standard arguments to `mpiexec` and various subsections of Section 5 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.

3 Quick Start

To use MPICH2, 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`.

If MPICH2 has not yet been installed, see the *MPICH2 Installer's Guide*.) We suggest that you put the `bin` subdirectory of that directory into 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.

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`, and that you have added that directory to your path, using

```
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.

4 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.

4.1 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.

4.2 Shared Libraries

Currently shared libraries are only tested on Linux and Mac OS X, 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.

4.3 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).

4.4 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.

5 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`.

5.1 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 exectuables 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 pg. 353 of [2].)

It is also possible to start a one process MPI job (with a `MPI_COMM_WORLD` whose size is equal to 1), without using `mpiexec`. This process will become an MPI process when it calls `MPI_Init`, and it may then call other MPI functions. Currently, MPICH2 does not fully support calling the dynamic process routines from MPI-2 (e.g., `MPI_Comm_spawn` or `MPI_Comm_accept`) from processes that are not started with `mpiexec`.

5.2 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.

5.3 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`.

5.3.1 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 5.1 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 6.1 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
```

5.3.2 Other Command-Line Arguments to `mpiexec` for MPD

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
```

In the case of MPD, we currently make an exception for the `PATH` environment variable, which is always passed through. This exception was added to make it unnecessary to explicitly pass this variable in the default case.

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 HOME,LD_LIBRARY_PATH -n 50 a.out
```

sets the HOME 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.

The redirection of `-stdin` through `mpiexec` to various MPI processes is intended primarily for interactive use. Because of the complexity of buffering large amounts of data at various processes that may not have read it yet, the redirection of large amounts of data to `mpiexec`'s `stdin` is discouraged, and may cause unexpected results. That is,

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

should not be used if `bigfile` is more than a few lines long. Have one of the processes open the file and read it instead. The functions in MPI-IO may be useful for this purpose.

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.

5.3.3 Environment Variables Affecting `mpiexec` for MPD

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_PORT_RANGE` If this environment variable is defined then the MPD system will restrict its usage of ports for connecting its various processes to ports in this range. If this variable is not assigned, but `MPICH_PORT_RANGE` *is* assigned, then it will use the range specified by `MPICH_PORT_RANGE` for its ports. Otherwise, it will use whatever paorts are assigned to it by the system. Port ranges are given as a pair of integers separated by a colon.

`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 – 1.2.7p1) 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.

5.4 Extensions for SMPD Process Management Environment

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).

5.4.1 `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 + ... + 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.

5.5 Extensions for the gforker Process Management Environment

`gforker` is a process management system for starting processes on a single machine, so called because the MPI processes are simply `fork`ed from the `mpiexec` process. This process manager supports programs that use `MPI_Comm_spawn` and the other dynamic process routines, but does not support the use of the dynamic process routines from programs that are not started with `mpiexec`. The `gforker` process manager is primiarily intended as a debugging aid as it simplifies development and testing of MPI programs on a single node or processor.

5.5.1 `mpiexec` arguments for gforker

In addition to the standard `mpiexec` command-line arguments, the `gforker` `mpiexec` supports the following options:

`-np <num>` A synonym for the standard `-n` argument

`-env <name> <value>` Set the environment variable `<name>` to `<value>` for the processes being run by `mpiexec`.

`-envnone` Pass no environment variables (other than ones specified with other `-env` or `-genv` arguments) to the processes being run by `mpiexec`. By default, all environment variables are provided to each MPI process (rationale: principle of least surprise for the user)

`-envlist <list>` Pass the listed environment variables (names separated by commas), with their current values, to the processes being run by `mpiexec`.

`-genv <name> <value>` The

-genv options have the same meaning as their corresponding `-env` version, except they apply to all executables, not just the current executable (in the case that the colon syntax is used to specify multiple execuables).

`-genvnone` Like `-envnone`, but for all executables

`-genvlist <list>` Like `-envlist`, but for all executables

`-usize <n>` Specify the value returned for the value of the attribute `MPI_UNIVERSE_SIZE`.

`-l` Label standard out and standard error (`stdout` and `stderr`) with the rank of the process

`-maxtime <n>` Set a timelimit of `<n>` seconds.

`-exitinfo` Provide more information on the reason each process exited if there is an abnormal exit

In addition to the commandline argments, the `gforker` `mpiexec` provides a number of environment variables that can be used to control the behavior of `mpiexec`:

`MPIEXEC_TIMEOUT` Maximum running time in seconds. `mpiexec` will terminate MPI programs that take longer than the value specified by `MPIEXEC_TIMEOUT`.

`MPIEXEC_UNIVERSE_SIZE` Set the universe size

`MPIEXEC_PORT_RANGE` Set the range of ports that `mpiexec` will use in communicating with the processes that it starts. The format of this is `<low>:<high>`. For example, to specify any port between 10000 and 10100, use `10000:10100`.

`MPICH_PORT_RANGE` Has the same meaning as `MPIEXEC_PORT_RANGE` and is used if `MPIEXEC_PORT_RANGE` is not set.

`MPIEXEC_PREFIX_DEFAULT` If this environment variable is set, output to standard output is prefixed by the rank in `MPI_COMM_WORLD` of the process and output to standard error is prefixed by the rank and the text `(err)`; both are followed by an angle bracket (`>`). If this variable is not set, there is no prefix.

`MPIEXEC_PREFIX_STDOUT` Set the prefix used for lines sent to standard output. A `%d` is replaced with the rank in `MPI_COMM_WORLD`; a `%w` is replaced with an indication of which `MPI_COMM_WORLD` in MPI jobs that involve multiple `MPI_COMM_WORLD`s (e.g., ones that use `MPI_Comm_spawn` or `MPI_Comm_connect`).

`MPIEXEC_PREFIX_STDERR` Like `MPIEXEC_PREFIX_STDOUT`, but for standard error.

`MPIEXEC_STDOUTBUF` Sets the buffering mode for standard output. Valid values are `NONE` (no buffering), `LINE` (buffering by lines), and `BLOCK` (buffering by blocks of characters; the size of the block is implementation defined). The default is `NONE`.

`MPIEXEC_STDERRBUF` Like `MPIEXEC_STDOUTBUF`, but for standard error.

5.6 Restrictions of the remshell Process Management Environment

The `remshell` "process manager" provides a very simple version of `mpiexec` that makes use of the secure shell command (`ssh`) to start processes on a collection of machines. As this is intended primarily as an illustration of how to build a version of `mpiexec` that works with other process managers, it does not implement all of the features of the other `mpiexec` programs described in this document. In particular, it ignores the command line options that control the environment variables given to the MPI programs. It does support the same output labeling features provided by the `gforker` version of `mpiexec`. However, this version of `mpiexec` can be used much like the `mpirun` for the `ch_p4` device in MPICH-1 to run programs on a collection of machines that allow remote shells. A file by the name of `machines` should contain the names of machines on which processes can be run, one machine name per line. There must be enough machines listed to satisfy the requested number of processes; you can list the same machine name multiple times if necessary.

For more complex needs or for faster startup, we recommend the use of the `mpd` process manager.

5.7 Using MPICH2 with SLURM and PBS

MPICH2 can be used in both SLURM and PBS environments. If configured with SLURM, use the `s`run job launching utility provided by SLURM. For PBS, MPICH2 jobs can be launched in two ways: (i) using MPD or (ii) using the OSC mpiexec.

5.7.1 MPD in the PBS environment

PBS specifies the machines allocated to a particular job in the file $PBS_NODEFILE. But the format used by PBS is different from that of MPD. Specifically, PBS lists each node on a single line; if a node (n0) has two processors, it is listed twice. MPD on the other hand uses an identifier (ncpus) to describe how many processors a node has. So, if n0 has two processors, it is listed as `n0:2`.

One way to convert the node file to the MPD format is as follows:

```
sort $PBS_NODEFILE | uniq -C | awk '{ printf("%s:%s", $2, $1); }' >
mpd.nodes
```

Once the PBS node file is converted, MPD can be normally started within the PBS job script using mpdboot and torn down using mpdallexit.

```
mpdboot -f mpd.hosts -n [NUM_NODES_REQUESTED]
mpiexec -n [NUM_PROCESSES] ./my_test_program
mpdallexit
```

5.7.2 OSC mpiexec

Pete Wyckoff from the Ohio Supercomputer Center provides a alternate utility called OSC mpiexec to launch MPICH2 jobs on PBS systems without using MPD. More information about this can be found here: http://www.osc.edu/∼pw/mpiexec

6 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.

6.1 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`.

7 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.

7.1 `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. The `-gdb` option 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 current implementation has some minor limitations. For example, we do not support setting your own prompt. This is because we capture the `gdb` output and examine it before processing it, e.g. merging identical lines. Also, we set a breakpoint at the beginning of `main` to get all processes synchronized at the beginning. Thus, the user will have a duplicate, unusable breakpoint if he sets one at the very first executable line of `main`. Otherwise, to the extent possible,

we try to simply pass user input through to `gdb` and lets things progress normally.

The following script of a `-gdb` session gives an idea of how this works. Input keystrokes are sent to all processes unless specifially 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 = 3.14159265358979323846264 3;
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:
                      file /home/lusk/mpich2/examples/cpi.c, line 30.
```

```
0-9:  (gdb) r
0-9:  Continuing.
0:  Process 0 of 10 is on ksl2
1:  Process 1 of 10 is on ksl2
2:  Process 2 of 10 is on ksl2
3:  Process 3 of 10 is on ksl2
4:  Process 4 of 10 is on ksl2
5:  Process 5 of 10 is on ksl2
6:  Process 6 of 10 is on ksl2
7:  Process 7 of 10 is on ksl2
8:  Process 8 of 10 is on ksl2
9:  Process 9 of 10 is on ksl2
0-9:
0-9:  Breakpoint 2, main (argc=1, argv=0x60000fffffffb4b8)
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=0x60000fffffffb4b8)
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)
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 0.0000000008333325
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`.

7.2 TotalView

MPICH2 supports use of the TotalView debugger from Etnus, through the MPD process manager only. If MPICH2 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`.

When debugging with TotalView using the above startup sequence, MPICH2 jobs cannot be restarted without exiting TotalView. In MPICH2 version 1.0.6 or later, TotalView can be invoked on an MPICH2 job as follows:

```
totalview python -a `which mpiexec` -tvsu \
            <mpiexec args> <program> <program args>
```

and the MPICH2 job will be fully restartable within TotalView.

If you have MPICH2 version 1.0.6 or later and TotalView 8.1.0 or later, you can use a TotalView feature called indirect launch with MPICH2. Invoke TotalView as:

```
totalview <program> -a <program args>
```

Then select the Process/Startup Parameters command. Choose the Parallel tab in the resulting dialog box and choose MPICH2 as the parallel system.

Then set the number of tasks using the Tasks field and enter other needed mpiexec arguments into the Additional Starter Arguments field.

If you want to be able to attach to a running MPICH2 job using TotalView, you must use the `-tvsu` option to mpiexec when starting the job. Using this option will add a barrier inside `MPI_Init` and hence may affect startup performance slightly. It will have no effect on the running of the job once all tasks have returned from `MPI_Init`. In order to debug a running MPICH2 job, you must attach TotalView to the instance of Python that is running the mpiexec script. If you have just one task running on the node where you invoked mpiexec, and no other Python scripts running, there will be three instances of Python running on the node. One of these is the parent of the MPICH2 task on that node, and one is the parent of that Python process. Neither of those is the instance of Python you want to attach to—they are both running the MPD script. The third instance of Python has no children and is not the child of a Python process. That is the one that is running mpiexec and is the one you want to attach to.

8 MPE

MPICH2 comes with the same MPE (Multi-Processing Environment) tools that are included with MPICH1. These include several trace libraries for recording the execution of MPI programs and the Jumpshot and SLOG tools for performance visualization, and a MPI collective and datatype checking library. The MPE tools are built and installed by default and should be available without requiring any additional steps. The easiest way to use MPE profiling libraries is through the `-mpe=` switch provided by MPICH2's compiler wrappers, `mpicc`, `mpicxx`, `mpif77`, and `mpif90`.

8.1 MPI Logging

MPE provides automatic MPI logging. For instance, to view MPI communication pattern of a program, fpilog.f, one can simply link the source file as follows:

```
mpif90 -mpe=mpilog -o fpilog fpilog.f
```

The `-mpe=mpilog` option will link with appropriate MPE profiling libraries. Then running the program through `mpiexec` will result a logfile, `Unknown.clog2`, in the working directory. The final step is to convert and view the logfile through Jumpshot:

```
jumpshot Unknown.clog2
```

8.2 User-defined logging

In addition to using the predefined MPE logging to log 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 typical way to insert user-defined states is as follows:

- Get handles from MPE logging library: `MPE_Log_get_state_eventIDs()` has to be used to get unique event IDs (MPE logging handles). [1] This is important if you are writing a library that uses the MPE logging routines from the MPE system. Hardwiring the eventIDs is considered a bad idea since it may cause eventID confict and so the practice isn't supported.
- Set the logged state's characteristics: `MPE_Describe_state()` sets the name and color of the states.
- Log the events of the logged states: `MPE_Log_event()` are called twice to log the user-defined states.

[1] Older MPE libraries provide `MPE_Log_get_event_number()` which is still being supported but has been deprecated. Users are strongly urged to use `MPE_Log_get_state_eventIDs()` instead.

Below is a simple example that uses the 3 steps outlined above.

```
int eventID_begin, eventID_end;
...
MPE_Log_get_state_eventIDs( &eventID_begin, &eventID_end );
...
MPE_Describe_state( eventID_begin, eventID_end,
                    "Multiplication", "red" );
...
MyAmult( Matrix m, Vector v )
{
    /* Log the start event of the red "Multiplication" state */
    MPE_Log_event( eventID_begin, 0, NULL );
    ... Amult code, including MPI calls ...
    /* Log the end event of the red "Multiplication" state */
    MPE_Log_event( eventID_end, 0, NULL );
}
```

The logfile generated by this code will have the MPI routines nested within the routine MyAmult().

Besides user-defined states, MPE2 also provides support for user-defined events which can be defined through use of `MPE_Log_get_solo_eventID()` and `MPE_Describe_event()`. For more details, e.g. see cpilog.c.

8.3 MPI Checking

To validate all the MPI collective calls in a program by linking the source file as follows:

```
mpif90 -mpe=mpicheck -o wrong_reals wrong_reals.f
```

Running the program will result with the following output:

```
> mpiexec -n 4 wrong_reals
Starting MPI Collective and Datatype Checking!
 Process            3  of            4  is alive
Backtrace of the callstack at rank 3:
        At [0]: wrong_reals(CollChk_err_han+0xb9)[0x8055a09]
```

```
        At [1]: wrong_reals(CollChk_dtype_scatter+0xbf)[0x8057bff]
        At [2]: wrong_reals(CollChk_dtype_bcast+0x3d)[0x8057ccd]
        At [3]: wrong_reals(MPI_Bcast+0x6c)[0x80554bc]
        At [4]: wrong_reals(mpi_bcast_+0x35)[0x80529b5]
        At [5]: wrong_reals(MAIN__+0x17b)[0x805264f]
        At [6]: wrong_reals(main+0x27)[0x80dd187]
        At [7]: /lib/libc.so.6(__libc_start_main+0xdc)[0x9a34e4]
        At [8]: wrong_reals[0x8052451]
[cli_3]: aborting job:
Fatal error in MPI_Comm_call_errhandler:

Collective Checking: BCAST (Rank 3) --> Inconsistent datatype signatures
                                        detected between rank 3 and rank 0.
```

The error message here shows that the `MPI_Bcast` has been used with inconsistent datatype in the program wrong_reals.f.

8.4 MPE options

Other MPE profiling options that are available through MPICH2 compiler wrappers are

```
    -mpe=mpilog     : Automatic MPI and MPE user-defined states logging.
                      This links against -llmpe -lmpe.

    -mpe=mpitrace   : Trace MPI program with printf.
                      This links against -ltmpe.

    -mpe=mpianim    : Animate MPI program in real-time.
                      This links against -lampe -lmpe.

    -mpe=mpicheck   : Check MPI Program with the Collective & Datatype
                      Checking library.  This links against -lmpe_collchk.

    -mpe=graphics   : Use MPE graphics routines with X11 library.
                      This links against -lmpe <X11 libraries>.

    -mpe=log        : MPE user-defined states logging.
                      This links against -lmpe.
```

```
-mpe=nolog      : Nullify MPE user-defined states logging.
                  This links against -lmpe_null.

-mpe=help       : Print the help page.
```

For more details of how to use MPE profiling tools, see `mpich2/src/mpe2/README`.

9 Other Tools Provided with MPICH2

MPICH2 also includes a test suite for MPI-1 and MPI-2 functionality; this suite may be found in the `mpich2/test/mpi` source directory and can be run with the command `make testing`. This test suite should work with any MPI implementation, not just MPICH2.

10 MPICH2 under Windows

10.1 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.

10.2 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`.

10.3 Running

MPI jobs are run from a command prompt using `mpiexec.exe`. See Section 5.4 on `mpiexec` for `smpd` for a description of the options to `mpiexec`.

A Frequently Asked Questions

This is the content of the online FAQ, as of June 23, 2006.

A.1 General Information

A.1.1 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.

A.1.2 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.

A.1.3 Q: Can MPI be used to program multicore systems?

A: There are two common ways to use MPI with multicore processors or multiprocessor nodes:

Use one MPI process per core (here, a core is defined as a program counter and some set of arithmetic, logic, and load/store units).

Use one MPI process per node (here, a node is defined as a collection of cores that share a single address space). Use threads or compiler-provided parallelism to exploit the multiple cores. OpenMP may be used with MPI; the loop-level parallelism of OpenMP may be used with any implementation of MPI (you do not need an MPI that supports `MPI_THREAD_MULTIPLE` when threads are used only for computational tasks). This is sometimes called the hybrid programming model.

A.2 Building MPICH2

A.2.1 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.

A.2.2 Q: Do I have to configure/make/install MPICH2 each time for each compiler I use?

No, in many cases you can build MPICH2 using one set of compilers and then use the libraries (and compilation scripts) with other compilers. However, this depends on the compilers producing *compatible* object files. Specifically, the compilers must

- Support the same basic datatypes with the same sizes. For example, the C compilers should use the same sizes for `long long` and `long double`.
- Map the names of routines in the source code to names in the object files in the object file in the same way. This can be a problem for Fortran and C++ compilers, though you can often force the Fortran compilers to use the same name mapping. More specifically, most Fortran compilers map names in the source code into all lower-case with one or two underscores appended to the name. To use the same MPICH2 library with all Fortran compilers, those compilers must make the same name mapping. There is one exception to this that is described below.
- Perform the same layout for C structures. The C langauge does not specify how structures are layed out in memory. For 100% compatibility, all compilers must follow the same rules. However, if you do not use any of the `MPI_MIN_LOC` or `MPI_MAX_LOC` datatypes, and you do not rely on the MPICH2 library to set the extent of a type created with `MPI_Type_struct` or `MPI_Type_create_struct`, you can often ignore this requirement.

- Require the same additional runtime libraries. Not all compilers will implement the same version of Unix, and some routines that MPICH2 uses may be present in only some of the run time libraries associated with specific compilers.

The above may seem like a stringent set of requirements, but in practice, many systems and compiler sets meet these needs, if for no other reason than that any software built with multiple libraries will have requirements similar to those of MPICH2 for compatibility.

If your compilers are completely compatible, down to the runtime libraries, you may use the compilation scripts (`mpicc` etc.) by either specifying the compiler on the command line, e.g.

```
mpicc -cc=icc -c foo.c
```

or with the environment variables `MPICH_CC` etc. (this example assume a c-shell syntax):

```
setenv MPICH_CC icc
mpicc -c foo.c
```

If the compiler is compatible *except* for the runtime libraries, then this same format works as long as a configuration file that describes the necessary runtime libraries is created and placed into the appropriate directory (the "`sysconfdir`" directory in configure terms). See the installation manual for more details.

In some cases, MPICH2 is able to build the Fortran interfaces in a way that supports multiple mappings of names from the Fortran source code to the object file. This is done by using the "multiple weak symbol" support in some environments. For example, when using `gcc` under Linux, this is the default.

A.2.3 Q: How do I configure to use the Absoft Fortran compilers?

A: You have several options. One is to use the Fortran 90 compiler for both F77 and F90. Another (if you do not need Fortran 90) is to use `--disable-f90` when configuring. The options with which we test MPICH2 and the Absoft compilers are the following:

```
setenv FFLAGS "-f -B108"
setenv F90FLAGS "-YALL_NAMES=LCS -B108"
setenv F77 f77
setenf F90 f90
```

A.2.4 Q: When I configure MPICH2, I get a message about FDZERO and the configure aborts

A: `FD_ZERO` is part of the support for the select calls (see "man select" or "man 2 select" on Linux and many other Unix systems) . What this means is that your system (probably a Mac) has a broken version of the select call and related data types. This is an OS bug; the only repair is to update the OS to get past this bug. This test was added specifically to detect this error; if there was an easy way to work around it, we would have included it (we don't just implement `FD_ZERO` ourselves because we don't know what else is broken in this implementation of select).

If this configure works with gcc but not with xlc, then the problem is with the include files that xlc is using; since this is an OS call (even if emulated), all compilers should be using consistent if not identical include files. In this case, you may need to update xlc.

A.2.5 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).

A.2.6 Q: When I run make, it fails immediately with many errors beginning with "sock.c:8:24: mpidu_sock.h: No such file or directory In file included from sock.c:9: ../../../../include/mpiimpl.h:91:21: mpidpre.h: No such file or directory In file included from sock.c:9: ../../../../include/mpiimpl.h:1150: error: syntax error before "MPID_VCRT" ../../../../include/mpiimpl.h:1150: warning: no semicolon at end of struct or union"

Check if you have set the envirnoment variable CPPFLAGS. If so, unset it and use CXXFLAGS instead. Then rerun configure and make.

A.2.7 Q: 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.

A.2.8 Q: When using the Intel Fortran 90 compiler (version 9), the make fails with errors in compiling statement that reference `MPI_ADDRESS_KIND`.

Check the output of the configure step. If configure claims that ifort is a cross compiler, the likely problem is that programs compiled and linked with ifort cannot be run because of a missing shared library. Try to compile and run the following program (named conftest.f90):

```
program conftest
integer, dimension(10) :: n
end
```

If this program fails to run, then the problem is that your installation of ifort either has an error or you need to add additional values to your environment variables (such as `LD_LIBRARY_PATH`). Check your installation documentation for the ifort compiler. See `http://softwareforums.intel.com/ISN/Community/en-US/search/SearchResults.aspx?q=libimf.so` for an example of problems of this kind that users are having with version 9 of ifort.

If you do not need Fortran 90, you can configure with `--disable-f90`.

A.2.9 Q: The build fails when I use parallel make

Parallel make (often invoked with `make -j4`) will cause several job steps in the build process to update the same library file (`libmpich.a`) concurrently. Unfortunately, neither the `ar` nor the `ranlib` programs correctly handle this case, and the result is a corrupted library. For now, the solution is to not use a parallel make when building MPICH2.

A.3 Windows version of MPICH2

A.3.1 I am having trouble installing and using the Windows version of MPICH2

See the tips for installing and running MPICH2 on Windows provided by a user, Brent Paul. Or see the MPICH2 Windows Development Guide.

A.4 Compiling MPI Programs

A.4.1 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).

A.4.2 C++ and Errors in Nullcomm::Clone

Some users, particularly with older C++ compilers, may see error messages of the form

```
"error C2555: 'MPI::Nullcomm::Clone' : overriding virtual function differs from
'MPI::Comm::Clone' only by return type or calling convention".
```

This is caused by the compiler not implementing part of the C++ standard. To work around this problem, add the definition

```
-DHAVE_NO_VARIABLE_RETURN_TYPE_SUPPORT
```

to the `CXXFLAGS` variable or add a

```
#define HAVE_NO_VARIABLE_RETURN_TYPE_SUPPORT 1
```

before including `mpi.h`.

A.5 Running MPI Programs

A.5.1 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. See the appropriate specific section.

A.5.2 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.

A.5.3 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.

A.5.4 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.

A.5.5 Q: My output does not appear until the program exits

A: Output to `stdout` and `stderr` may not be written from your process immediately after a `printf` or `fprintf` (or `PRINT` in Fortran) because, under Unix, such output is *buffered* unless the program believes that the output is to a terminal. When the program is run by `mpiexec`, the C standard I/O library (and normally the Fortran runtime library) will buffer the output. For C programmers, you can either use a call `fflush(stdout)` to force the output to be written or you can set no buffering by calling

```
#include <stdio.h>

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

on each file descriptor (`stdout` in this example) which you want to send the output immedately to your terminal or file.

There is no standard way to either change the buffering mode or to flush the output in Fortran. However, many Fortrans include an extension to provide this function. For example, in `g77`,

```
call flush()
```

can be used. The `xlf` compiler supports

```
call flush_(6)
```

where the argument is the Fortran logical unit number (here `6`, which is often the unit number associated with `PRINT`). With the G95 Fortran 95 compiler, set the environment variable `G95_UNBUFFERED_6` to cause output to unit 6 to be unbuffered.

A.5.6 Q: Fortran programs using stdio fail when using g95

A: By default, g95 does not flush output to stdout. This also appears to cause problems for standard input. If you are using the Fortran logical units 5 and 6 (or the `*` unit) for standard input and output, set the environment variable `G95_UNBUFFERED_6` to `yes`.

A.5.7 Q: How do I run MPI programs in the background when using the default MPD process manager?

A: To run MPI programs in the background when using MPD, you need to redirect stdin from `/dev/null`. For example,

```
mpiexec -n 4 a.out < /dev/null &
```

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.