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** .................................................. 1

2. **Quick Start** .................................................. 1

   2.1 Prerequisites .................................................. 1

   2.2 From A Standing Start to Running an MPI Program .......... 2

   2.3 Compiler Optimization Levels ................................ 6

   2.4 Common Non-Default Configuration Options ................. 7

      2.4.1 The Most Important Configure Options ................... 7

      2.4.2 Using the Absoft Fortran compilers with MPICH2 ....... 8

   2.5 Shared Libraries ............................................. 9

   2.6 What to Tell the Users ...................................... 9

3. **Migrating from MPICH1** ...................................... 9

   3.1 Configure Options ......................................... 9

   3.2 Other Differences ......................................... 10

4. **Choosing the Communication Device** .......................... 11

5. **Installing and Managing Process Managers** ................ 11

   5.1 hydra ....................................................... 12

   5.2 mpd ........................................................... 12

      5.2.1 Configuring mpd ........................................ 12

      5.2.2 System Requirements ................................... 12

      5.2.3 Using mpd .............................................. 13

      5.2.4 Options for mpd ...................................... 14

      5.2.5 Running MPD on multi-homed systems .................. 14
1 Introduction

This manual describes how to obtain and install MPICH2, the MPI-2 implementation from Argonne National Laboratory. (Of course, if you are reading this, chances are good that you have already obtained it and found this document, among others, in its doc subdirectory.) This Guide will explain how to install MPICH so that you and others can use it to run MPI applications. Some particular features are different if you have system administration privileges (can become “root” on a Unix system), and these are explained here. It is not necessary to have such privileges to build and install MPICH2. In the event of problems, send mail to mpich-discuss@mcs.anl.gov. Once MPICH2 is installed, details on how to run MPI jobs are covered in the MPICH2 User’s Guide, found in this same doc subdirectory.

MPICH2 has many options. We will first go through a recommended, “standard” installation in a step-by-step fashion, and later describe alternative possibilities.

2 Quick Start

In this section we describe a “default” set of installation steps. It uses the default set of configuration options, which builds the sock communication device and the MPD process manager, for languages C, C++, Fortran-77, and Fortran-90 (if those compilers exist), with compilers chosen automatically from the user’s environment, without tracing and debugging options. It uses the VPATH feature of make, so that the build process can take place on a local disk for speed.

2.1 Prerequisites

For the default installation, you will need:

1. A copy of the distribution, mpich2.tar.gz.
3. A Fortran-77, Fortran-90, and/or C++ compiler if you wish to write MPI programs in any of these languages.
4. Any one of a number of Unix operating systems, such as IA32-Linux. MPICH2 is most extensively tested on Linux; there remain some difficulties on systems to which we do not currently have access. Our configure script attempts to adapt MPICH2 to new systems.

Configure will check for these prerequisites and try to work around deficiencies if possible. (If you don’t have Fortran, you will still be able to use MPICH2, just not with Fortran applications.)

2.2 From A Standing Start to Running an MPI Program

Here are the steps from obtaining MPICH2 through running your own parallel program on multiple machines.

1. Unpack the tar file.

   \texttt{tar xzf mpich2.tar.gz}

   If your tar doesn’t accept the \textit{z} option, use

   \texttt{gunzip -c mpich2.tar.gz | tar xf -}

   Let us assume that the directory where you do this is /\texttt{home/you/libraries}. It will now contain a subdirectory named \texttt{mpich2-1.3}.

2. Choose an installation directory (the default is /\texttt{usr/local/bin}):

   \texttt{mkdir /home/you/mpich2-install}

   It will be most convenient if this directory is shared by all of the machines where you intend to run processes. If not, you will have to duplicate it on the other machines after installation. Actually, if you leave out this step, the next step will create the directory for you.

3. Choose a build directory. Building will proceed \textit{much} faster if your build directory is on a file system local to the machine on which the configuration and compilation steps are executed. It is preferable that this also be separate from the source directory, so that the source directories remain clean and can be reused to build other copies on other machines.
mkdir /tmp/you/mpich2-1.3

4. Choose any configure options. See Section 2.4.1 for a description of the most important options to consider.

5. Configure MPICH2, specifying the installation directory, and running the configure script in the source directory:

   cd /tmp/you/mpich2-1.3
   /home/you/libraries/mpich2-1.3/configure \
      -prefix=/home/you/mpich2-install |& tee c.txt

where the \ means that this is really one line. (On sh and its derivatives, use 2>&1 | tee c.txt instead of |& tee c.txt). Other configure options are described below. Check the c.txt file to make sure everything went well. Problems should be self-explanatory, but if not, send c.txt to mpich-discuss@mcs.anl.gov. The file config.log is created by configure and contains a record of the tests that configure performed. It is normal for some tests recorded in config.log to fail.

6. Build MPICH2:

   make |& tee m.txt      (for csh and tcsh)
   OR
   make 2>&1 | tee m.txt   (for bash and sh)

This step should succeed if there were no problems with the preceding step. Check file m.txt. If there were problems, do a make clean and then run make again with VERBOSE=1

   make VERBOSE=1 |& tee m.txt    (for csh and tcsh)
   OR
   make VERBOSE=1 2>&1 | tee m.txt (for bash and sh)

and then send m.txt and c.txt to mpich-discuss@mcs.anl.gov.

7. Install the MPICH2 commands:

   make install |& tee mi.txt
This step collects all required executables and scripts in the bin subdirectory of the directory specified by the prefix argument to configure. (For users who want an install directory structure compliant to GNU coding standards (i.e., documentation files go to \(\text{datarootdir}/\text{doc}/\{\text{PACKAGE}\}\)), architecture independent read-only files go to \(\text{datadir}/\{\text{PACKAGE}\}\)), replace make install by

\[
\text{make install PACKAGE=mpich2-<version>}
\]

and corresponding installcheck step should be

\[
\text{make installcheck PACKAGE=mpich2-<version>}
\]

Setting PACKAGE in make install or make installcheck step is optional and unnecessary for typical MPI users.)

8. Add the bin subdirectory of the installation directory to your path:

\[
\text{setenv PATH /home/you/mpich2-install/bin:$PATH}
\]

for csh and tcsh, or

\[
\text{export PATH=/home/you/mpich2-install/bin:$PATH}
\]

for bash and sh. Check that everything is in order at this point by doing

\[
\text{which mpicc}
\]
\[
\text{which mpiexec}
\]

All should refer to the commands in the bin subdirectory of your install directory. It is at this point that you will need to duplicate this directory on your other machines if it is not in a shared file system such as NFS.

9. Check that you can reach these machines with ssh or rsh without entering a password. You can test by doing

\[
\text{ssh othermachine date}
\]
or

\texttt{rsh othermachine date}

If you cannot get this to work without entering a password, you will need to configure \texttt{ssh} or \texttt{rsh} so that this can be done.

10. Test the setup you just created:

\[
\texttt{mpiexec \texttt{-f machinefile -n <number> hostname}}
\]

The machinefile contains the list of hosts you want to run the executable on.

\[
\% \texttt{cat machinefile}
\]

\[
\begin{align*}
\text{host1} & \quad \# \text{Run 1 process on host1} \\
\text{host2:4} & \quad \# \text{Run 4 processes on host2} \\
\text{host3:2} & \quad \# \text{Run 2 processes on host3} \\
\text{host4:1} & \quad \# \text{Run 1 process on host4}
\end{align*}
\]

11. Now we will run an MPI job, using the \texttt{mpiexec} command as specified in the MPI-2 standard.

As part of the build process for MPICH2, a simple program to compute the value of $\pi$ by numerical integration is created in the \texttt{mpich2-1.3/examples} directory. If the current directory is the top level MPICH2 build directory, then you can run this program with

\[
\texttt{mpiexec \texttt{-n 5 -f machinefile ./examples/cpi}}
\]

The \texttt{cpi} example will tell you which hosts it is running on.

There are many options for \texttt{mpiexec}, by which multiple executables can be run, hosts can be specified (as long as they are in the mpd ring), separate command-line arguments and environment variables can be passed to different processes, and working directories and search paths for executables can be specified. Do

\[
\texttt{mpiexec \texttt{--help}}
\]

for details. A typical example is:
mpiexec -f machinefile -n 1 ./master : -n 19 ./slave

to ensure that the process with rank 0 runs on your workstation.

The arguments between `:'s in this syntax are called “argument sets,”
since they apply to a set of processes. More arguments are described in
the User’s Guide.

If you have completed all of the above steps, you have successfully in-
stalled MPICH2 and run an MPI example.

2.3 Compiler Optimization Levels

MPICH2 can be configured with two sets of compiler flags: CFLAGS, CXXFLAGS,
FFLAGS, FCFLAGS (abbreviated as xFLAGS) and MPICH2LIB_CFLAGS, MPICH2LIB_CXXFLAGS,
MPICH2LIB_FFLAGS, MPICH2LIB_FCFLAGS (abbreviated as MPICH2LIB_xFLAGS)
for compilation; LDFLAGS and MPICH2LIB_LDFLAGS for linking. All these flags
can be set as part of configure command or through environment variables.
(CPPFLAGS stands for C preprocessor flags, which should NOT be set)

Both xFLAGS and MPICH2LIB_xFLAGS affect the compilation of the MPICH2
libraries. However, only xFLAGS is appended to MPI wrapper scripts, mpicc
and friends.

MPICH2 libraries are built with default compiler optimization, -O2,
which can be modified by --enable-fast configure option. For instance, --
disable-fast disables the default optimization option. --enable-fast=O<n> sets
default compiler optimization as -O<n>. For more details of --enable-fast, see
the output of "configure --help". Any other complicated optimization flags
for MPICH2 libraries have to be set through MPICH2LIB_xFLAGS. CFLAGS
and friends are empty by default.

For example, to build a ”production” MPICH2 environment with -O3
for all language bindings, one can simply do

./configure --enable-fast=O3

or

./configure --disable-fast MPICH2LIB_CFLAGS=-O3 \
This will cause the MPICH2 libraries to be built with -O3, and -O3 will not be included in the mpicc and other MPI wrapper script.

There are certain compiler flags that should not be used with MPICH2’s configure, e.g. gcc’s -Werror which would confuse configure and cause certain configure tests to fail to detect the correct system features. To use -Werror in building MPICH2 libraries, you can pass the compiler flags during the make step through the Makefile variable, \texttt{MPICH2\_MAKE\_CFLAGS} as follows:

\begin{verbatim}
make VERBOSE=1 MPICH2\_MAKE\_CFLAGS="-Wall -Werror"
\end{verbatim}

(assume CC is set to gcc). The content of \texttt{MPICH2\_MAKE\_CFLAGS} is appended to the \texttt{CFLAGS} in almost all Makefiles.

### 2.4 Common Non-Default Configuration Options

A list of configure options is found in Section \ref{section:config-options}. Here we comment on some of them.

#### 2.4.1 The Most Important Configure Options

- \texttt{-prefix} Set the installation directories for MPICH2.

- \texttt{-enable-debuginfo} Provide access to the message queues for debuggers such as Totalview.

- \texttt{-enable-g} Build MPICH2 with various debugging options. This is of interest primarily to MPICH2 developers. The options

  \begin{verbatim}
  --enable-g=dbg,mem,log
  \end{verbatim}

  are recommended in that case.

- \texttt{-enable-fast} Configure MPICH2 for fastest performance at the expense of error reporting and other program development aids. This is recommended only for getting the best performance out of proven production applications, and for benchmarking.
**enable-shared**  Build MPICH2 with shared libraries. MPICH2 will try to automatically detect the type of shared library support required. For cases where you want to manually control this, the following options can be used.

--enable-sharedlibs=gcc for standard gcc on Linux
--enable-sharedlibs=osx-gcc for Mac OS X or
--enable-sharedlibs=solaris-cc for cc on Solaris

**with-pm**  Select the process manager. The default is hydra; also useful are gforker and mpd. You can build with all three process managers by specifying

--with-pm=hydra:mpd:gforker

**without-mpe**  Configure MPICH2 without the MPE package of program development tools (including the Jumpshot performance viewer)

**with-java**  Set the location of Java installation. This option is necessary only if the default Java installation in your PATH does not contain a valid Java installation for Jumpshot, e.g.

--with-java=/opt/jdk1.6.0

### 2.4.2 Using the Absoft Fortran compilers with MPICH2

For best results, it is important to force the Absoft Fortran compilers to make all routine names monocase. In addition, if lower case is chosen (this will match common use by many programs), you must also tell the the Absoft compiles to append an underscore to global names in order to access routines such as getarg (getarg is not used by MPICH2 but is used in some of the tests and is often used in application programs). We recommend configuring MPICH2 with the following options

```bash
setenv F77 f77
setenv FFLAGS "-f -N15"
setenv FCFLAGS "-YALL_NAMES=LCS -YEXT_SFX=_"

./configure ....
```
2.5 Shared Libraries

Shared libraries are currently only supported for gcc (and gcc-like compilers) on Linux and Mac and for cc on Solaris. To have shared libraries created when MPICH2 is built, specify the following when MPICH2 is configured:

```
configure --enable-shared
```

For users who wish to manually control the linker parameters, this can be done using:

```
configure --enable-sharedlibs=gcc  (on Linux)
configure --enable-sharedlibs=osx-gcc (on Mac OS X)
configure --enable-sharedlibs=solaris-cc (on Solaris)
```

2.6 What to Tell the Users

Now that MPICH2 has been installed, the users have to be informed of how to use it. Part of this is covered in the User’s Guide. Other things users need to know are covered here.

3 Migrating from MPICH1

MPICH2 is an all-new rewrite of MPICH1. Although the basic steps for installation have remained the same (configure, make, make install), a number of things have changed. In this section we attempt to point out what you may be used to in MPICH1 that are now different in MPICH2.

3.1 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.

### 3.2 Other Differences

Other differences between MPICH1 and MPICH2 include the handling of process managers and the choice of communication device.

For example, the new process managers have a new format and slightly different semantics for the `-machinefile` option. Assume that you type this data into a file named `machfile`:

```
bp400:2
bp401:2
bp402:2
bp403:2
```

If you then run a parallel job with this machinefile, you would expect ranks 0 and 1 to run on bp400 because it says to run 2 processes there before going on to bp401. Ranks 2 and 3 would run on bp401, and rank 4 on bp402, e.g.:

```
mpiexec -l -machinefile machfile -n 5 hostname
```

produces:

```
0: bp400
1: bp400
2: bp401
3: bp401
4: bp402
```
4 Choosing the Communication Device

MPICH2 is designed to be build with many different communication devices, allowing an implementation to be tuned for different communication fabrics. A simple communication device, known as “ch3” (for the third version of the “channel” interface) is provided with MPICH2 and is the default choice.

The ch3 device itself supports a variety of communication methods. These are specified by providing the name of the method after a colon in the `--with-device` configure option. For example, `--with-device=ch3:sock` selects the (older) socket-base communication method. The supported methods include:

- **ch3:nemesis**: This method is our new, high performance method. It has been made the default communication channel starting the 1.1 release of MPICH2. It uses shared-memory to send messages between processes on the same node and the network for processes between nodes. Currently sockets and Myrinet-MX are supported networks. It supports `MPI_THREAD_MULTIPLE` and other levels of thread safety.

- **ch3:sock**: This method uses sockets for all communications between processes. It supports `MPI_THREAD_MULTIPLE` and other levels of thread safety.

Most installations should use the default ch3:nemesis method for best performance. For platforms that are not supported by nemesis, the ch3:sock method is suggested.

5 Installing and Managing Process Managers

MPICH2 has been designed to work with multiple process managers; that is, although you can start MPICH2 jobs with `mpiexec`, there are different mechanisms by which your processes are started. An interface (called PMI) isolates the MPICH2 library code from the process manager. Currently three process managers are distributed with MPICH2.

- **hydra**: This is the default process manager that natively uses the existing daemons on the system such as ssh, slurm, pbs.
mpd  This is the traditional process manager used in MPICH2. It consists of a ring of daemons.

smpd  This one can be used for both Linux and Windows. It is the only process manager that supports the Windows version of MPICH2.

gforker  This is a simple process manager that creates all processes on a single machine. It is useful for both debugging and on shared memory multiprocessors.

5.1 hydra

hydra  is the default process manager that launches processes using the native daemons present on the system such as ssh, slurm, pbs, etc. To configure with the hydra process manager, use

    configure --with-pm=hydra ... 

5.2 mpd

5.2.1 Configuring mpd

The mpd process manager can be explicitly chosen at configure time by adding

    --with-pm=mpd

to the configure arguments. This is not necessary, since mpd is the default.

5.2.2 System Requirements

mpd consists of a number of components written in Python. The configure script should automatically find a version of python in your PATH that has all the features needed to run mpd. If for some reason you need to pick a specific version of Python for mpd to use, you can do so by adding

    --with-python=<fullpathname of python interpreter>
to your configure arguments. If your system doesn’t have Python, you can get the latest version from http://www.python.org. Most Linux distributions include a moderately current version of Python. MPD requires release 2.2 or later.

The mpd process manager supports the use of the TotalView parallel debugger from Etnus. If totalview is in your PATH when MPICH2 is configured, then an interface module will be automatically compiled, linked, and installed so that you can use TotalView to debug MPICH jobs (See the User’s Guide under “Debugging”. You can also explicitly enable or disable this capability with --enable-totalview or --disable-totalview as arguments to configure.

5.2.3 Using mpd

In Section 2.2 you installed the mpd ring. Several commands can be used to use, test, and manage this ring. You can find out about them by running mpdhelp, whose output looks like this:

The following mpd commands are available. For usage of any specific one, invoke it with the single argument --help.

mpd start an mpd daemon
mpdtrace show all mpd’s in ring
mpdboot start a ring of daemons all at once
mpdstringtest test how long it takes for a message to circle the ring
mpdallexit take down all daemons in ring
mpdcleanup repair local Unix socket if ring crashed badly
mpdlistjobs list processes of jobs (-a or --all: all jobs for all users)
mpdkilljob kill all processes of a single job
mpdsigjob deliver a specific signal to the application processes of a job
mpiexec start a parallel job

Each command can be invoked with the --help argument, which prints usage information for the command without running it.

So for example, to see a complete list of the possible arguments for mpdboot, you would run

mpdboot --help
5.2.4 Options for mpd

`-help` causes mpd to print a list and description of all options

In addition to the cmd-line options, mpd will also check for presence of the environment variable `MPICH_PORT_RANGE` (note MPICH instead of MPD) and use only the ports in that range for listening sockets. The range is separated by a colon, e.g., 2000:8000.

5.2.5 Running MPD on multi-homed systems

If you plan to use one or more multi-homed systems, it is of course useful if the default hostname is associated with the interface that mpd will need to use for communications. If not however, you can cause mpd to use a specific interface by using the `--ifhn` (interface-hostname) option, e.g.:

```
n1% mpd --ifhn=192.168.1.1 &
```

If you then run mpiexec on n1 connecting to that mpd, the mpiexec will use the same ifhn for communications with remote processes that connect back to it. mpiexec will also accept a -ifhn option (mpiexec -help) in the unlikely event that you wish it to use a separate interface from the mpd.

mpdboot can also designate the ifhn to be used by both the local and remote mpds which it starts, e.g.:

```
n1% mpdboot --totalnum=3 --ifhn=192.168.1.1
```

where mpd.hosts contains:

```
n2 ifhn=192.168.1.2
n3 ifhn=192.168.1.3
```

will start one mpd locally, one on n2 and one on n3. Each will use the respectively designated ifhn.
5.2.6 Running MPD as Root

MPD can run as root to support multiple users simultaneously. To do this, it easiest to simply do the “make install” in the mpd sub-directory as root. This will cause the mpdroot program to be installed in the bin directory with setuid-root permissions. Individual users then have the option of starting and using their own mpd rings, or they can run with a ring started by root. To use root’s ring, they must use an option named MPD_USE_ROOT_MPD. This option may either be set as an environment variable or they can set it in their own .mpd.conf file, e.g.:

MPD_USE_ROOT_MPD=1

When root starts the mpds in the ring, the procedure is the same as for a regular user except that root’s configuration file is in /etc/mpd.conf (note that there is no leading dot in the file name).

5.2.7 Running MPD on SMP’s

Typically one starts one mpd on each host. When a job is started with mpiexec without any particular host specification, the processes are started on the ring of hosts one at a time, in round-robin fashion until all the processes have been started. Thus, if you start a four-process job on a ring of two machines, hosta and hostb, then you will find ranks 0 and 2 on hosta and ranks 1 and 3 on hostb. This might not be what you want, especially if the machines are SMP’s and you would like to have consecutive ranks on the same machine as much as possible. If you tell the mpd how many cpus it has to work with by using the --ncpus argument, as in

mpd --ncpus=2

then the number of processes started the first time the startup message circles the ring will be determined by this argument. That is, in the above four-process example, ranks 0 and 1 will be on hosta and ranks 2 and 3 will be on hostb. This effect only occurs the first time around the ring. That is if you start a six-process job on this ring (two mpd’s, each started with --ncpus=2) you will get processes 0, 1, and 4 on hosta and 2, 3, and 5 on hostb. This is for load-balancing purposes. (It is assumed that you do not want 0, 1, 4, and 5 on hosta and only 2 and 3 on hostb; if that is what
you do want, you can control process placement explicitly on the \texttt{mpiexec} command, as described in the \textit{User’s Guide}.

### 5.2.8 Security Issues in MPD

Since the \texttt{mpd} process manager allows the remote execution of processes, it needs to incorporate security features that prevent inappropriate or malicious process creation by someone without permission to do so. In this section we describe how \texttt{mpd} deals with a number of security issues.

The first issue is the starting of the \texttt{mpd} processes themselves. This is done either by the user logging into each machine starting each \texttt{mpd} process “by hand,” or else by using the \texttt{mpdboot} program. \texttt{mpdboot} uses \texttt{ssh} by default, although the less secure \texttt{rsh} can be used if the user chooses.

The next issue occurs when a single new \texttt{mpd} joins a ring of existing \texttt{mpd}’s. This occurs during \texttt{mpdboot}, or can occur later if the user wishes to expand an existing ring. In this situation a “challenge-response” protocol is followed. A user must have the same \texttt{secretword} set in his \texttt{.mpd.conf} file on each machine. This file must be readable only by the user starting the \texttt{mpd}; otherwise the \texttt{mpd} will refuse to read it. When \texttt{mpd} A wishes to join an existing ring by connecting to \texttt{mpd} B, which is already in the ring, at the port B is listening on, the following sequence of events occurs:

1. B temporarily accepts the connection.
2. B seeds the random number generator with the high-resolution part of the time of day, generates a random number, and sends it to A.
3. A concatenates the random number with the value of \texttt{secretword} in \texttt{.mpd.conf}, encrypts it with \texttt{md5}, and sends the encrypted result to B.
4. Meanwhile, B encrypts the (same) random number concatenated with its value of \texttt{secretword}.
5. When B receives the encrypted value from A, it compares the encrypted value with its own.
6. If the values match, A is allowed to join the ring; otherwise the connection is closed.
Note that the secret word is never sent over the connection in the clear, and
the use of the time of day means that there will be no repeating pattern of
challenges to be observed.

The third issue concerns how mpiexec connects to the local mpd. This is
done through a Unix socket in /tmp rather than through an INET socket.
Thus security is preserved through the security of the file system.

Once connections are established, the messages that are sent over them
are not encrypted. However, no security information is sent over these con-
nections.

5.3 SMPD

5.3.1 Configuration

You may add the following configure options, --with-pm=smpd, to build
and install the smpd process manager. The process manager, smpd, will be
installed to the bin sub-directory of the installation directory of your choice
specified by the --prefix option.

smpd process managers run on each node as stand-alone daemons and
need to be running on all nodes that will participate in MPI jobs. smpd
process managers are not connected to each other and rely on a known port
to communicate with each other. Note: If you want multiple users to use
the same nodes they must each configure their smpds to use a unique port
per user.

smpd uses a configuration file to store settings. The default location is
~/smpd. This file must not be readable by anyone other than the owner and
contains at least one required option - the access passphrase. This is stored
in the configuration file as phrase=<phrase>. Access to running smpds is
authenticated using this passphrase and it must not be your user password.

5.3.2 Usage and administration

Users will start the smpd daemons before launching mpi jobs. To get an
smpd running on a node, execute

    smpd -s
Executing this for the first time will prompt the user to create a `~/.smpd` configuration file and passphrase if one does not already exist.

Then users can use `mpiexec` to launch MPI jobs.

All options to `smpd`:

```
smmd -s
```

Start the smpd service/daemon for the current user. You can add `-p <port>` to specify the port to listen on. All smpds must use the same port and if you don’t use the default then you will have to add `-p <port>` to mpiexec or add the `port=<port>` to the `.smpd` configuration file.

```
smmd -r
```

Start the smpd service/daemon in root/multi-user mode. This is not yet implemented.

```
smmd -shutdown [host]
```

Shutdown the smpd on the local host or specified host. Warning: this will cause the smpd to exit and no mpiexec or smpd commands can be issued to the host until smpd is started again.

### 5.4 gforker

`gforker` is a simple process manager that runs all processes on a single node; it’s version of `mpiexec` uses the system `fork` and `exec` calls to create the new processes. To configure with the `gforker` process manager, use

```
configure --with-pm=gforker ...
```

### 6 Testing

Once MPICH2 has been installed, you can test it by running some of the example programs in the `examples` directory. A more thorough test can be run with the command `make testing`. This will produce a summary on standard output, along with an XML version of the test results in `mpich2/test/mpi`. In addition, running `make testing` from the top-level (mpich2) directory will run tests of the commands, such as `mpicc` and `mpiexec`, that are included with MPICH2.
Other MPI test suites are available from http://www.mcs.anl.gov/mpi/mpi-test/tsuite.html. As part of the MPICH2 development, we run the MPICH1, MPICH2, C++, and Intel test suites every night and post the results on http://www.mcs.anl.gov/mpi/mpich1/micronotes/mpich2-status/. Other tests are run on an occasional basis.

7 Benchmarking

There are many benchmarking programs for MPI implementations. Three that we use are mpptest (http://www.mcs.anl.gov/mpi/mpptest), netpipe (http://www.scl.ameslab.gov/netpipe), and SkaMPI (http://liinwww.ira.uka.de/~skampi). Each of these has different strengths and weaknesses and reveals different properties of the MPI implementation.

In addition, the MPICH2 test suite contains a few programs to test for performance artifacts in the directory test/mpi/perf. An example of a performance artifact is markedly different performance for the same operation when performed in two different ways. For example, using an MPI datatype for a non-contiguous transfer should not be much slower than packing the data into a contiguous buffer, sending it as a contiguous buffer, and then unpacking it into the destination buffer. An example of this from the MPI-1 standard illustrates the use of MPI datatypes to transpose a matrix “on the fly,” and one test in test/mpi/perf checks that the MPI implementation performs well in this case.

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. The MPE tools are built and installed by default and should be available without requiring any additional steps. The installation of MPE is documented in mpich2/src/mpe2/INSTALL and the usage of MPE is documented in mpich2/src/mpe2/README and MPICH2 user’s guide.
9 Windows Version

9.1 Binary distribution

The Windows binary distribution uses the Microsoft Installer. Download and execute \texttt{mpich2-1.x.xxx.msi} to install the binary distribution. The default installation path is \texttt{C:\Program Files\MPICH2}. You must have administrator privileges to install \texttt{mpich2-1.x.xxx.msi}. The installer installs a Windows service to launch MPICH applications and only administrators may install services. This process manager is called \texttt{smpd.exe}. Access to the process manager is passphrase protected. The installer asks for this passphrase. Do not use your user password. The same passphrase must be installed on all nodes that will participate in a single MPI job.

Under the installation directory are three sub-directories: \texttt{include}, \texttt{bin}, and \texttt{lib}. The \texttt{include} and \texttt{lib} directories contain the header files and libraries necessary to compile MPI applications. The \texttt{bin} directory contains the process manager, \texttt{smpd.exe}, and the the MPI job launcher, \texttt{mpiexec.exe}. The dlls that implement MPICH2 are copied to the Windows system32 directory.

You can install MPICH in unattended mode by executing

\texttt{msiexec /q /I mpich2-1.x.xxx.msi}

The \texttt{smpd} process manager for Windows runs as a service that can launch jobs for multiple users. It does not need to be started like the unix version does. The service is automatically started when it is installed and when the machine reboots. \texttt{smpd} for Windows has additional options:

\texttt{smpd -start}
\begin{itemize}
  \item Start the Windows \texttt{smpd} service.
\end{itemize}

\texttt{smpd -stop}
\begin{itemize}
  \item Stop the Windows \texttt{smpd} service.
\end{itemize}

\texttt{smpd -install}
\begin{itemize}
  \item Install the \texttt{smpd} service.
\end{itemize}

\texttt{smpd -remove}
\begin{itemize}
  \item Remove the \texttt{smpd} service.
smpd -register_spn
  Register the Service Principal Name with the domain controller. This command enables passwordless authentication using kerberos. It must be run on each node individually by a domain administrator.

9.2 Source distribution

In order to build MPICH2 from the source distribution under Windows, you must have MS Developer Studio .NET 2003 or later, perl and optionally Intel Fortran 8 or later.

- Download mpich2-1.x.y.tar.gz and unzip it.
- Bring up a Visual Studio Command prompt with the compiler environment variables set.
- Run winconfigure.wsf. If you don’t have a Fortran compiler add the --remove-fortran option to winconfigure to remove all the Fortran projects and dependencies. Execute winconfigure.wsf /? to see all available options.
- open mpich2\mpich2.sln
- build the ch3sockRelease mpich2 solution
- build the ch3sockRelease mpich2s project
- build the Release mpich2 solution
- build the fortRelease mpich2 solution
- build the gfortRelease mpich2 solution
- build the sfortRelease mpich2 solution
- build the channel of your choice. The options are sock and nemesis. If you plan on launching more processes than you have processors you should use the default sock channel. The nemesis channel uses a polling progress engine that can perform poorly when multiple processes compete for individual processors.
9.3 cygwin

MPICH2 can also be built under cygwin using the source distribution and the Unix commands described in previous sections. This will not build the same libraries as described in this section. It will build a “Unix” distribution that runs under cygwin.

9.4 Vista/Windows 7/Windows Server 2008 Users

To install MPICH version 1.3 or later please follow the instructions provided in \texttt{9.1}. Please follow the steps below to install older versions of MPICH on Windows flavors with User Account Control.

1. Right-click on the command prompt icon and choose ”Run as administrator”

2. You can install MPICH from the administrator command prompt by executing

\begin{verbatim}
msiexec /I mpich2-1.x.xxx.msi
\end{verbatim}

10 All Configure Options

To get the latest list of all the configure options recognized by the top-level configure, use:

\begin{verbatim}
configure --help
\end{verbatim}

Not all of these options may be fully supported yet.

Notes on the configure options. The \texttt{--with-hmldir} and \texttt{--with-docdir} options specify the directories into which the documentation will be installed by \texttt{make install}. 
A Troubleshooting MPDs

A.1 Getting Started with mpd

mpd stands for multi-purpose daemon. We sometimes use the term mpd to refer to the combination of the mpd daemon and its helper programs that collectively form a process management system for executing parallel jobs, including mpich jobs. The mpd daemon must run on each host where you wish to execute parallel programs. The mpd daemons form a ring to facilitate rapid process startup. Even a single mpd on a single host forms a loop. Therefore, each host must be configured in such a way that the mpds can connect to each other and pass messages via sockets.

It can be rather tricky to configure one or more hosts in such a way that they adequately support client-server applications like mpd. In particular, each host must not only know its own name, but must identify itself correctly to other hosts when necessary. Further, certain information must be readily accessible to each host. For example, each host must be able to map another host’s name to its IP address. In this section, we will walk slowly through a series of steps that will help to ensure success in running mpds on a single host or on a large cluster.

If you can ssh from each machine to itself, and from each machine to each other machine in your set (and back), then you probably have an adequate environment for mpd. However, there may still be problems. For example, if you are blocking all ports except the ports used by ssh/sshd, then mpd will still fail to operate correctly.

To begin using mpd, the sequence of steps that we recommend is this:

1. get one mpd working alone on a first test node
2. get one mpd working alone on a second test node
3. get two new mpds to work together on the two test nodes
4. boot two new mpds on the two test nodes via mpdboot

A.1.1 Following the steps

1. Install mpich2, and thus mpd.
2. Make sure the mpich2 bin directory is in your path. Below, we will refer to it as MPDDIR.

3. Kill old mpd processes. If you are coming to this guide from elsewhere, e.g. a Quick Start guide for mpich2, because you encountered mpd problems, you should make sure that all mpd processes are terminated on the hosts where you have been testing. mpdallexit may assist in this, but probably not if you were having problems. You may need to use the Unix kill command to terminate the processes.

4. Run a first mpd (alone on a first node). As mentioned above, mpd uses client-server communications to perform its work. So, before running an mpd, let’s run a simpler program (mpdcheck) to verify that these communications are likely to be successful. Even on hosts where communications are well supported, sometimes there are problems associated with hostname resolution, etc. So, it is worth the effort to proceed a bit slowly. Below, we assume that you have installed mpd and have it in your path.

Select a test node, let’s call it n1. Login to n1.

First, we will run mpdcheck as a server and a client. To run it as a server, get into a window with a command-line and run this:

```
n1 $ mpdcheck -s
```

It will print something like this:

```
server listening at INADDR_ANY on: n1 1234
```

Now, run the client side (in another window if convenient) and see if it can find the server and communicate. Be sure to use the same hostname and portnumber printed by the server (above: n1 1234):

```
n1 $ mpdcheck -c n1 1234
```

If all goes well, the server will print something like:

```
server has conn on
  <socket._socketobject object at 0x40200f2c>
  from ('192.168.1.1', 1234)
server successfully recvd msg from client:
  hello_from_client_to_server
```
and the client will print:

```
client successfully recv'd ack from server:
  ack_from_server_to_client
```

If the experiment failed, you have some network or machine configuration problem which will also be a problem later when you try to use mpd. Even if the experiment succeeded, but the hostname printed by the server was `localhost`, then you will probably have problems later if you try to use mpd on n1 in conjunction with other hosts. In either case, skip to Section “Debugging host/network configuration problems.”

If the experiment succeeded, then you should be ready to try mpd on this one host. To start an mpd, you will use the mpd command. To run parallel programs, you will use the mpiexec program. All mpd commands accept the `-h` or `--help` arguments, e.g.:

```
  n1 $ mpd --help
  n1 $ mpiexec --help
```

Try a few tests:

```
  n1 $ mpd &
  n1 $ mpiexec -n 1 /bin/hostname
  n1 $ mpiexec -l -n 4 /bin/hostname
  n1 $ mpiexec -n 2 PATH_TO_MPICH2_EXAMPLES/cpi
```

where `PATH_TO_MPICH2_EXAMPLES` is the path to the `mpich2-1.0.3/examples` directory.

To terminate the mpd:

```
  n1 $ mpdallexit
```

5. Run a second mpd (alone on a second node). To verify that things are fine on a second host (say n2), login to n2 and perform the same set of tests that you did on n1. Make sure that you use mpdallexit to terminate the mpd so you will be ready for further tests.
6. Run a ring of two mpds on two hosts. Before running a ring of mpds on n1 and n2, we will again use mpdcheck, but this time between the two machines. We do this because the two nodes may have trouble locating each other or communicating between them and it is easier to check this out with the smaller program.

First, we will make sure that a server on n1 can service a client from n2. On n1:

```
n1 $ mpdcheck -s
```

which will print a hostname (hopefully n1) and a portnumber (say 3333 here). On n2:

```
n2 $ mpdcheck -c n1 3333
```

If this experiment fails, skip to Section A.2 “Debugging host/network configuration problems”.

Second, we will make sure that a server on n2 can service a client from n1. On n2:

```
n2 $ mpdcheck -s
```

which will print a hostname (hopefully n2) and a portnumber (say 7777 here). On n2:

```
n2 $ mpdcheck -c n2 7777
```

If this experiment fails, skip to Section A.2 “Debugging host/network configuration problems”.

If all went well, we are ready to try a pair of mpds on n1 and n2. First, make sure that all mpds have terminated on both n1 and n2. Use mpdallexit or simply kill them with:

```
kill -9 PID_OF_MPD
```

where you have obtained the `PID_OF_MPD` by some means such as the `ps` command.

On n1:
n1 $ mpd &
n1 $ mpdtrace -l

This will print a list of machines in the ring, in this case just n1. The output will be something like:

n1_6789 (192.168.1.1)

The 6789 is the port that the mpd is listening on for connections from other mpds wishing to enter the ring. We will use that port in a moment to get an mpd from n2 into the ring. The value in parentheses should be the IP address of n1.

On n2:

n2 $ mpd -h n1 -p 6789 &

where 6789 is the listening port on n1 (from mpdtrace above). Now try:

n2 $ mpdtrace -l

You should see both mpds in the ring.

To run some programs in parallel:

n1 $ mpiexec -n 2 /bin/hostname
n1 $ mpiexec -n 4 /bin/hostname
n1 $ mpiexec -l -n 4 /bin/hostname
n1 $ mpiexec -l -n 4 PATH_TO_MPICH2_EXAMPLES/cpi

where PATH_TO_MPICH2_EXAMPLES is the path to the mpich2-1.3/examples directory.

To bring down the ring of mpds:

n1 $ mpdallexit

7. Boot a ring of two mpds via mpdboot. Please be aware that mpdboot uses ssh by default to start remote mpds. It will expect that you can run ssh from n1 to n2 (and from n2 to n1) without entering a password. First, make sure that you terminate the mpd processes from any prior tests.

On n1, create a file named mpd.hosts containing the name of n2:
n2

Then, on n1 run:

```
n1 $ mpdboot -n 2
n1 $ mpdtrace -l
n1 $ mpiexec -l -n 2 /bin/hostname
```

The `mpdboot` command should read the `mpd.hosts` file created above and run an `mpd` on each of the two machines. The `mpdtrace` and `mpiexec` show the ring up and functional. Options that may be useful are:

- `--help` use this one for extra details on all options
- `-v` (verbose)
- `--chkup` tries to verify that the hosts are up before starting `mpds`
- `--chkuponly` only performs the verify step, then ends

To bring the ring down:

```
n1 $ mpdallexit
```

If `mpdboot` works on the two machines n1 and n2, it will probably work on your others as well. But, there could be configuration problems using a new machine on which you have not yet tested `mpd`. An easy way to check, is to gradually add them to `mpd.hosts` and try an `mpdboot` with a `-n` arg that uses them all each time. Use `mpdallexit` after each test.

### A.2 Debugging host/network configuration problems

We use `mpdcheck` as our first attempt to debug host or network configuration problems. If you run:

```
n1 $ mpdcheck --help
```

you should receive a fairly long help message describing a wide variety of arguments which can be supplied to `mpdcheck` to help you debug.

The first thing to try is to simply login to a node, say n1 and run:
n1 $ mpdcheck

mpdcheck will produce no output here if it finds no problems. If mpdcheck
does find potential problems, it will print them with *** at the beginning
of the line. You can cause mpdcheck to be verbose by using the -v option,
e.g.:

n1 $ mpdcheck -v

Also, if mpdcheck offers comments about how you might repair certain prob-
lems, you can get a longer version of those messages by using the -l option,
e.g.:

n1 $ mpdcheck -l

If you run mpdcheck on each node and find no problems, you may still
wish to use it further to debug issues between two nodes. For example, you
might login to n1 and create file named mpd.hosts which contains the name
of another node which is having trouble communicating with n1, e.g. n2.
Then, you may want to run:

n1 $ mpdcheck -f mpd.hosts

This test will see if n1 is having trouble discovering information about n2.
If not, you wish to try:

n1 $ mpdcheck -f mpd.hosts -ssh

This will also try to test ssh support between n1 and n2.

If these 2 experiments go OK, you should probably try them again but
this time logged into n2 and trying to connect back to n1. Do not forget to
change the contents of mpd.hosts to contain the name of n1.

If none of these get you past the problems, you may need to ask for help.
If so, it will probably useful to run mpdcheck once more on each of the nodes
which are of concern:

n1 $ mpdcheck -pc
n2 $ mpdcheck -pc
These will produce quite a bit of output which may be useful in determining
the problem. The -pc option does not really try to offer any comments about
what may be wrong. It merely prints potentially useful debugging info.

A.3 Firewalls, etc.

If the output from any of mpdcheck, mpd, or mpdboot leads you to believe
that one or more of your hosts are having trouble communicating due to
firewall issues, we can offer a few simple suggestions. If the problems are
due to an “enterprise” firewall computer, then we can only point you to your
local network admin for assistance.

In other cases, there are a few quick things that you can try to see if there
some common protections in place which may be causing your problems.

First, you might see if iptables is active. You will probably need to be
root to do this:

n1 # iptables -L

This will show a set of 3 current iptables chains being applied for each
of INPUT, FORWARD, and OUTPUT. If the chains are non-empty, then
you may have something blocked. This could be a result of a software
firewall package you are running (e.g. shorewall) or some home-grown set of
chains. If you are unfamiliar with iptables, you will need to get local help
to decipher the rules and determine if any of them may be affecting you.
There are options such as -F to iptables that will disable the chains, but
that is dangerous of course if you require them for protection.

Next, you might see if any tcp-wrappers are active. You may need to be
root to do this:

n1 # cat /etc/hosts.deny /etc/hosts.allow

If there are any uncommented lines, they likely designate any (or ALL)
daemons which have their tcp communications blocked. This can be partic-
ularly problematic for mpdboot which uses ssh (and thus the ssh daemon,
sshd).

Next, you might wish to see if you have available ephemeral ports:
A TROUBLESHOOTING MPDS

n1 $ cat /proc/sys/net/ipv4/ip_local_port_range

This should print a range something like:

32768   61000