MPICH2 Installer's Guide*

Version 1.0.3

Mathematics and Computer Science Division
Argonne National Laboratory

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

November 23, 2005

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

Contents

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 `mpich2-maint@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. This *Installer's Guide* is for MPICH2 Release 1.0. We are reserving the 1.0 designation for when every last feature of the MPI-2 Standard is implemented, but most features are included. See the `RELEASE_NOTES` file in the top-level directory for details.

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 as many of languages C, C++, Fortran-77, and Fortran-90 compilers as it can find, 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`.
2. A C compiler.

3. A fortran-77, Fortran-90, and/or C++ compiler if you wish to write MPI programs in any of these languages.

4. Python 2.2 or later version, for building the default process management system, MPD. Most systems have Python pre-installed, but you can get it free from `www.python.org`. You may assume it is there unless the `configure` step below complains.

5. 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 we do not currently have access to. 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.)

This default installation procedure builds and installs MPICH2 ready for C, C++, Fortran 77, and Fortran 90 programs, using the MPD process manager (and it builds and installs MPD itself), without debugging options. Regardless of where the source resides, the build takes place on a local file system, where compilation is likely to be much faster than on a network-attached file system, but the installation directory that is accessed by users can be on a shared file system. For other options, see the appropriate sections later in the document.

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.

   ```
   tar xfz mpich2.tar.gz
   ```

   If your tar doesn't accept the z option, use

   ```
   gunzip -c mpich2.tar.gz | tar xf -
   ```

   Let us assume that the directory where you do this is `/home/you/libraries`. It will now contain a subdirectory named `mpich2-1.0`.

2. Choose an installation directory (the default is `/usr/local/bin`):

   ```
   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 *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/mpich2-1.0
   ```

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

   ```
   cd  /tmp/you/mpich2-1.0
   /home/you/libraries/mpich2-1.0/configure \
           -prefix=/home/you/mpich2-install |& tee configure.log
   ```

   where the \ means that this is really one line. (On `sh` and its derivatives, use `2>&1 | tee configure.log` instead of `|& tee configure.log`). Other configure options are described below. Check the `configure.log` file to make sure everything went well. Problems should be self-explanatory, but if not, send `configure.log` to `mpich2-maint@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.

5. Build MPICH2:

   ```
   make |& tee make.log
   ```

   This step should succeed if there were no problems with the preceding step. Check `make.log`. If there were problems, send `configure.log` and `make.log` to `mpich2-maint@mcs.anl.gov`.

6. Install the MPICH2 commands:

```
make install |& tee install.log
```

This step collects all required executables and scripts in the `bin` subdirectory of the directory specified by the prefix argument to configure.

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

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

for `csh` and `tcsh`, or

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

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

```
which mpd
which mpicc
which mpiexec
which mpirun
```

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.

8. MPICH2, unlike MPICH, uses an external process manager for scalable startup of large MPI jobs. The default process manager is called MPD, which is a ring of daemons on the machines where you will run your MPI programs. In the next few steps, you will get this ring up and tested. The instructions given here will probably be enough to get you started. If not, you should refer to Appendix A for troubleshooting help. More details on interacting with MPD can be found by running `mpdhelp` or any mpd command with the `--help` option, or by viewing the README file in `mpich2/src/pm/mpd`. The information provided includes how to list running jobs, kill, suspend, or otherwise signal them, and how to use the `gdb` debugger via special arguments to `mpiexec`.

For security reasons, mpd looks in your home directory for a file named `.mpd.conf` containing the line

```
secretword=<secretword>
```

where `<secretword>` is a string known only to yourself. It should not be your normal Unix password. Make this file readable and writable only by you:

```
cd $HOME
touch .mpd.conf
chmod 600 .mpd.conf
```

Then use an editor to place a line like:

```
secretword=mr45-j9z
```

into the file. (Of course use a different secret word than `mr45-j9z`.)

9. The first sanity check consists of bringing up a ring of one mpd on the local machine, testing one mpd command, and bringing the "ring" down.

```
mpd &
mpdtrace
mpdallexit
```

The output of mpdtrace should be the hostname of the machine you are running on. The `mpdallexit` causes the mpd daemon to exit. If you encounter problems, you should check the troubleshooting section of Appendix A.

10. The next sanity check is to run a non-MPI program using the daemon.

```
mpd &
mpiexec -n 1 /bin/hostname
mpdallexit
```

This should print the name of the machine you are running on. If not, you should check the troubleshooting section in Appendix A.

11. Now we will bring up a ring of `mpd`'s on a set of machines. Create a file consisting of a list of machine names, one per line. Name this file `mpd.hosts`. These hostnames will be used as targets for `ssh` or `rsh`, so include full domain names if necessary. Check that you can reach these machines with `ssh` or `rsh` without entering a password. You can test by doing

```
ssh othermachine date
```

or

```
rsh othermachine date
```

If you cannot get this to work without entering a password, you will need to configure `ssh` or `rsh` so that this can be done, or else use the workaround for `mpdboot` in the next step.

12. Start the daemons on (some of) the hosts in the file mpd.hosts

```
mpdboot -n <number to start> -f mpd.hosts
```

The number to start can be less than 1 + number of hosts in the file, but cannot be greater than 1 + the number of hosts in the file. One mpd is always started on the machine where `mpdboot` is run, and is counted in the number to start, whether or not it occurs in the file. By default, `mpdboot` will only start one `mpd` per machine even if the machine name appears in the hosts file multiple times. The `-1` option can be used to override this behavior, but there is typically no reason for a user to need multiple mpds on a single host. The `-1` option exists mostly to support internal testing. The `--help` option explains these as well as other useful options. In particular, if your cluster has multiprocessor nodes, you might want to use the `--ncpus` argument described in Section **??**.

Check to see if all the hosts you listed in mpd.hosts are in the output of

```
mpdtrace
```

and if so move on to the next step.

There is a workaround if you cannot get `mpdboot` to work because of difficulties with `ssh` or `rsh` setup. You can start the daemons "by hand" as follows:

```
mpd &            # starts the local daemon
mpdtrace -l      # makes the local daemon print its host
                 # and port in the form <host>_<port>
```

Then log into each of the other machines, put the `install/bin` directory in your path, and do:

```
mpd -h <hostname> -p <port> &
```

where the hostname and port belong to the original mpd that you started. From each machine, after starting the mpd, you can do

```
mpdtrace
```

to see which machines are in the ring so far. More details on `mpdboot` and other options for starting the mpd's are in `mpich2-1.0/src/pm/mpd/README`.

In case of persistent difficulties getting the ring of mpd's up and running on the machines you want, please see Appendix A. There we discuss the mpd's in more detail, together with some programs for testing the configuration of your systems to make sure that they allow the mpd's to run.

13. Test the ring you have just created:

    ```
    mpdtrace
    ```

    The output should consist of the hosts where MPD daemons are now running. You can see how long it takes a message to circle this ring with

    ```
    mpdringtest
    ```

    That was quick. You can see how long it takes a message to go around many times by giving mpdringtest an argument:

    ```
    mpdringtest 100
    mpdringtest 1000
    ```

14. Test that the ring can run a multiprocess job:

    ```
    mpiexec -n <number> hostname
    ```

    The number of processes need not match the number of hosts in the ring; if there are more, they will wrap around. You can see the effect of this by getting rank labels on the stdout:

```
mpiexec -l -n 30 hostname
```

You probably didn't have to give the full pathname of the hostname command because it is in your path. If not, use the full pathname:

```
mpiexec -l -n 30 /bin/hostname
```

15. Now we will run an MPI job, using the `mpiexec` command as specified in the MPI-2 standard. There are some examples in the install directory, which you have already put in your path, as well as in the directory `mpich2-1.0/examples`. One of them is the classic `cpi` example, which computes the value of π by numerical integration in parallel.

    ```
    mpiexec -n 5 cpi
    ```

    The number of processes need not match the number of hosts. The `cpi` example will tell you which hosts it is running on. By default, the processes are launched one after the other on the hosts in the mpd ring, so it is not necessary to specify hosts when running a job with `mpiexec`.

    There are many options for `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

    ```
    mpiexec --help
    ```

    for details. A typical example is:

    ```
    mpiexec -n 1 master : -n 19 slave
    ```

    or

    ```
    mpiexec -n 1 -host mymachine 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 argments are described in the *User's Guide*.

The `mpirun` command from the original MPICH is still available, although it does not support as many options as `mpiexec`. You might want to use it in the case where you do not have the XML parser required for the use of `mpiexec`.

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

2.3 Common Non-Default Configuration Options

A list of `configure` options is found in Section 9. Here we comment on some of them. This section needs much more work.

2.3.1 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

```
setenv F77 f77
setenv MPI_FFLAGS "-f -N15"
setenv MPI_F90FLAGS "-YALL_NAMES=LCS -YEXT_SFX=_"

./configure ....
```

2.4 Shared Libraries

Shared libraries are currently only supported by gcc and tested under Linux. To have shared libraries created when MPICH2 is built, specify the following when MPICH2 is configured:

```
configure --enable-sharedlibs=gcc
```

2.5 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. (E.g., whether they need to run their own mpd rings or use a system-wide one run by root.)

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 mpd has 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.py -l -machinefile machfile -n 5 hostname
```

produces:

```
0: bp400
1: bp400
2: bp401
3: bp401
4: bp402
```

4 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

mpd This is the default, and the one that is described in Section 2.2. 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.

4.1 mpd

4.1.1 Configuring mpd

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

```
--with-pm=mpd
```

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

4.1.2 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 compiler, 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`.

4.1.3 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
mpdringtest   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
```

4.1.4 Options for `mpd`

–**help** causes mpd to print a list and description of all options

4.1.5 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=4 --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.

4.1.6 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_MPD_ROOT=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).

4.1.7 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 4 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 `mpiexec` command, as described in the *User's Guide*.

4.2 SMPD

4.2.1 Configuration

You may add the following configure options, `--with-pm=smpd --with-pmi=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.

4.2.2 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`:

`smpd -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.

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

`smpd -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.

4.3 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 ...
```

5 Testing

Running basic tests in the examples directory, the MPICH2 tests, obtaining and running the assorted test suites.

5.1 Using the Intel Test Suite

These instructions are local to our test environment at Argonne.

How to run a select set of tests from the Intel test suite:

```
1) checkout the Intel test suite (cvs co IntelMPITEST) (outside users
   should access the most recent version of the test suite from the
   test suite web page).

2) create a testing directory separate from the IntelMPITEST source
directory

3) cd into that testing directory

4) make sure the process manager (e.g., mpd) is running

5) run "<ITS_SRC_DIR>/configure --with-mpich2=<MPICH2_INSTALL_DIR>", where
<ITS_SRC_DIR> is the path to the directory Intel test suite source (e.g.,
/home/toonen/Projects/MPI-Tests/IntelMPITEST) and <MPICH2_INSTALL_DIR> is
the directory containing your MPICH2 installation

6) mkdir Test; cd Test

7) find tests in <ITS_SRC_DIR>/{c,fortran} that you are interested in
running and place the test names in a file.  For example:

% ( cd /home/toonen/Projects/MPI-Tests/IntelMPITEST/Test ; \
    find {c,fortran} -name 'node.*' -print | grep 'MPI_Test'
    | sed -e 's-/node\..*$--' ) |& tee testlist
Test/c/nonblocking/functional/MPI_Test
Test/c/nonblocking/functional/MPI_Testall
Test/c/nonblocking/functional/MPI_Testany
Test/c/nonblocking/functional/MPI_Testsome
Test/c/persist_request/functional/MPI_Test_p
Test/c/persist_request/functional/MPI_Testall_p
Test/c/persist_request/functional/MPI_Testany_p
Test/c/persist_request/functional/MPI_Testsome_p
```

```
Test/c/probe_cancel/functional/MPI_Test_cancelled_false
Test/fortran/nonblocking/functional/MPI_Test
Test/fortran/nonblocking/functional/MPI_Testall
Test/fortran/nonblocking/functional/MPI_Testany
Test/fortran/nonblocking/functional/MPI_Testsome
Test/fortran/persist_request/functional/MPI_Test_p
Test/fortran/persist_request/functional/MPI_Testall_p
Test/fortran/persist_request/functional/MPI_Testany_p
Test/fortran/persist_request/functional/MPI_Testsome_p
Test/fortran/probe_cancel/functional/MPI_Test_cancelled_false
%

8) run the tests using ../bin/mtest:

% ../bin/mtest -testlist testlist -np 6 |& tee mtest.log
%

NOTE: some programs hang if less they are run with less than 6 processes.

9) examine the summary.xml file.  look for '<STATUS>fail</STATUS>' to see if
any failures occurred.  (search for '>fail<' works as well)
```

6 Benchmarking

netpipe, mpptest, others (SkaMPI).

7 MPE

This section describes what MPE is and its potentially separate installation. It includes discussion of Java-related problems.

8 Windows Version

8.1 Binary distribution

The Windows binary distribution uses the Microsoft Installer. Download and execute mpich2-1.x.xxx.msi to install the binary distribution. The de-

fault installation path is `C:\Program Files\MPICH2`. You must have administrator privileges to install mpich2.msi. The installer installs a Windows service to launch MPICH applications and only administrators may install services. This process manager is called 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: `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 the MPI job launcher, `mpiexec.exe`. The dlls that implement MPICH2 are copied to the Windows system32 directory.

You can install MPICH in unattended mode by executing

```
msiexec /q /I mpich2-1.x.xxx.msi
```

The 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. smpd for Windows has additional options:

`smpd -start`
: Start the Windows `smpd` service.

`smpd -stop`
: Stop the Windows `smpd` service.

`smpd -install`
: Install the `smpd` service.

`smpd -remove`
: Remove the `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 individualy by a domain administrator.

8.2 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.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 `shm`, `ssm`, `sshm`, `ib`, `essm`, and `mt`. The shm channel is for small numbers of processes that will run on a single machine using shared memory. The shm channel should not be used for more than about 8 processes. The sshm (scalable shared memory) is for use with more than 8 processes. The `ssm` (sock shared memory) channel is for clusters of smp nodes. This channel should not be used if you plan to over-subscribe the CPU's. If you plan on launching more processes than you have processors you should use the default `sock` channel or the `essm` channel. The `ssm` channel uses a polling progress engine that can perform poorly when multiple processes compete for individual processors. The `essm` channel is derived from the ssm channel with the addition of OS event objects to avoid spinning in the progress engine. The `mt channel` is

the multi-threaded socket channel. The `ib` channel is for clusters with Infiniband interconnects from Mellanox.

8.3 `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 All Configure Options

Here is a list of all the configure options currently recognized by the top-level configure. It is the MPICH-specific part of the output of

```
configure --help
```

Not all of these options may be fully supported yet.

```
Optional Features:
  --disable-FEATURE       do not include FEATURE (same as --enable-FEATURE=no)
  --enable-FEATURE[=ARG]  include FEATURE [ARG=yes]
--enable-cache  - Turn on configure caching
--enable-echo  - Turn on strong echoing. The default is enable=no.
--enable-strict - Turn on strict debugging with gcc
--enable-coverage - Turn on coverage analysis using gcc and gcov
--enable-error-checking=level - Control the amount of error checking.
level may be
    no        - no error checking
    runtime   - error checking controllable at runtime through environment
                variables
    all       - error checking always enabled
--enable-error-messages=level - Control the amount of detail in error
  messages.  Level may be
    all       - Maximum amount of information
    generic   - Only generic messages (no information about the specific
                instance)
    class     - One message per MPI error class
    none      - No messages
```

```
--enable-timing=level - Control the amount of timing information
collected by the MPICH implementation.  level may be
    none    - Collect no data
    all     - Collect lots of data
    runtime - Runtime control of data collected
The default is none.
--enable-threads=level - Control the level of thread support in the
MPICH implementation.  The following levels are supported.
    single - No threads (MPI_THREAD_SINGLE)
    funneled - Only the main thread calls MPI (MPI_THREAD_FUNNELED)
    serialized - User serializes calls to MPI (MPI_THREAD_SERIALIZED)
    multiple:impl - Fully multi-threaded (MPI_THREAD_MULTIPLE)
The default is funneled.  If enabled and no level is specified, the
level is set to multiple.  If disabled, the level is set to single.
When the level is set to multiple, an implementation may also be
specified.  The following implementations are supported.
    global_mutex - a single global lock guards access to all MPI functions.
The default implementation is global_mutex.
--enable-g=option - Control the level of debugging support in the MPICH
implementation.  option may be a list of common separated names including
    none     - No debugging
    mem      - Memory usage tracing
    handle   - Trace handle operations
    dbg      - Add compiler -g flags
    log      - Enable debug event logging
    meminit  - Preinitialize memory associated structures and unions to
               eliminate access warnings from programs like valgrind
    all      - All of the above choices
--enable-fast - pick the appropriate options for fast execution.  This
                turns off error checking and timing collection
--enable-f77 - Enable Fortran 77 bindings
--enable-f90 - Enable Fortran 90 bindings
--enable-cxx - Enable C++ bindings
--enable-romio - Enable ROMIO MPI I/O implementation
--enable-nmpi-as-mpi - Use MPI rather than PMPI routines for MPI routines,
 such as the collectives, that may be implemented in terms of other MPI
 routines
--enable-mpe - Build the MPE (MPI Parallel Environment) routines
--enable-weak-symbols - Use weak symbols to implement PMPI routines (default)
--enable-sharedlibs=kind - Enable shared libraries.  kind may be
    gcc     - Standard gcc and GNU ld options for creating shared libraries
    osx-gcc - Special options for gcc needed only on OS/X
    solaris-cc - Solaris native (SPARC) compilers for 32 bit systems
    cygwin-gcc - Special options for gcc needed only for cygwin
    none    - same as --disable-sharedlibs
```

```
Only gcc, osx-gcc, and solaris-cc are currently supported

--enable-dependencies - Generate dependencies for sourcefiles.  This
            requires that the Makefile.in files are also created
            to support dependencies (see maint/updatefiles)
--enable-runtimevalues - Determine various parameters of the Fortran
                         environment at run time, such as the values for
                         .TRUE. and .FALSE. .  This allows a single MPICH
                         library to be used with multiple Fortran compilers
--enable-timer-type=name - Select the timer to use
for MPI_Wtime and internal timestamps.  name may be one of
    gethrtime - Solaris timer (Solaris systems only)
    clock_gettime   - Posix timer (where available)
    gettimeofday - Most Unix systems
    linux86_cycle - Linux x86; returns cycle counts, not time in seconds
    linuxalpha_cycle - Like linux86_cycle, but for Linux Alpha
    gcc_ia64_cycle - IPF ar.itc timer


Optional Packages:
  --with-PACKAGE[=ARG]    use PACKAGE [ARG=yes]
  --without-PACKAGE       do not use PACKAGE (same as --with-PACKAGE=no)
--with-device=name - Specify the communication device for MPICH.
--with-pmi=name - Specify the pmi interface for MPICH.
--with-pm=name - Specify the process manager for MPICH.
      Multiple process managers may be specified as long as they all use
      the same pmi interface by separating them with colons.  The
      mpiexec for the first named process manager will be installed.
      Example: --with-pm=forker:mpd:remshell builds the three process
      managers forker, mpd, and remshell; only the mpiexec from forker
      is installed into the bin directory.
--with-logging=name - Specify the logging library for MPICH.
--with-mpe - Build the MPE (MPI Parallel Environment) routines
--with-htmldir=dir - Specify the directory for html documentation
--with-docdir=dir - Specify the directory for documentation
--with-cross=file - Specify the values of variables that configure cannot
determine in a cross-compilation environment
--with-namepublisher=name - Choose the system that will support
                            MPI_PUBLISH_NAME and MPI_LOOKUP_NAME.  Options
                            include
                              no (no service available)
                              mpd
       file:directory (optional directory)

--with-cxxlibname=name  - Specify name of library containing C++ interface
```

```
routines
--with-flibname=name  - Specify name of library containing Fortran interface
routines
--with-fwrapname=name  - Specify name of library containing Fortran interface
routines
--with-thread-package=package - Thread package to use.  Supported thread
packages include:
    posix or pthreads - POSIX threads
    solaris - Solaris threads (Solaris OS only)
    none - no threads
If the option is not specified, the default package is ${MPE_THREAD_DEFAULT}.
If the option is specified, but a package is not given, then the default
is posix.

Some influential environment variables:
  CC          C compiler command
  CFLAGS      C compiler flags
  LDFLAGS     linker flags, e.g. -L<lib dir> if you have libraries in a
              nonstandard directory <lib dir>
  CPPFLAGS    C/C++ preprocessor flags, e.g. -I<include dir> if you have
              headers in a nonstandard directory <include dir>
  F77         Fortran 77 compiler command
  FFLAGS      Fortran 77 compiler flags
  F90         Fortran 90 compiler command
  F90FLAGS    Fortran 90 compiler flags
  CXX         C++ compiler command
  CXXFLAGS    C++ compiler flags
  CPP         C preprocessor

Use these variables to override the choices made by 'configure' or to help
it to find libraries and programs with nonstandard names/locations.
```

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 recvd 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 the 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
```

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 the section "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 the section "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 listeneing 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 listeneing 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
```

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 potenital 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 problems, 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 comemnts 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 particularly problematic for mpdboot which uses ssh (and thus the ssh daemon, sshd).

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

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

This should print a range something like:

```
32768   61000
```