MPICH2 Installer's Guide[*]
Version 0.4
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

September 9, 2004

[*]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 preferred, "standard" installation in a step-by-step fashion, and later describe alternative possibilities. This *Installer's Guide* is for MPICH2 Release 0.971. 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. PyXML and an XML parser like expat (in order to use mpiexec with the MPD process manager). Most systems have Python, PyXML, and expat pre-installed, but you can get them free from `www.python.org`. Assume they are there unless the `configure` step below complains.

5. Any one of a number of Unix operating systems, such as 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 and Fortran (-77) 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 mpich2.tar
   ```

Let us assume that the directory where you do this is `/home/you/libraries`. It will now contain a subdirectory named `mpich2-0.971`.

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 it remains clean and can be reused to build other copies on other machines.

   ```
   mkdir /tmp/you/mpich2-0.971
   ```

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

   ```
   cd  /tmp/you/mpich2-0.971
   /home/you/libraries/mpich2-0.971/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, sent `configure.log` to `mpich2-maint@mcs.anl.gov`.

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. More details on interacting with MPD can be found in the README file in `mpich2/src/pm/mpd`, such as how to list running jobs, kill, suspend, or otherwise signal them, and how to use the `mpigdb` debugger. The instructions given here should be enough to get you started.

   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
echo "secretword=mr45-j9z" >> .mpd.conf
```

(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.

10. 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.

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

    ```
    mpdboot -n <number to start>
    ```

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.

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 `mpich2-0.971/src/pm/mpd/README`.

12. 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
```

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

    ```
    mpdrun -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:

    ```
    mpdrun -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:

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

14. 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 `mpich2-0.97/examples`. One of them is the classic `cpi` example, which computes the value of π by numerical integration in parallel.

    ```
    mpiexec -n 5 cpi
    ```

    As with `mpdrun` (which is used internally by `mpiexec`), 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 : -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. **Change this to match new global and local arguments described in User's Guide.** There is an extra argument set for arguments that apply to all the processes, introduced by the -default argument. For example, to get rank labels on standard output, use

```
mpiexec -default -l : -n 3 cpi
```

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

`enable-g, enable-fast, devices, pms, etc.`

Reference Section 8.

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 --disable-cxx
```

since currently shared libraries are incompatible with the C++ libraries. It is currently also necessary to build the Fortran libraries, which is the default. so just be sure not to configure with `--disable-f77`.

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 Installing and Managing Process Managers

3.1 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
mpdrun        start a parallel job
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

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
```

3.1.1 Options for `mpd`

-procspernode is used when allowing MPD to pick the hosts: it tells MPD how many processes should be started by each MPD in the ring as the processes are started in round-robin fashion. `-ppn` is an alias for `-procspernode`.

3.1.2 Running MPD as Root

How to run mpd as root for other people to use. Test whether all that is necessary is for root to be the one who runs the install step.

3.2 SMPD

3.2.1 Configuration

You may add the following configure options, `--with-pm=smpd --with-pmi=smpd`, to build and install the smpd process manager. The sprocess 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.

3.2.2 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.

3.3 Forker

3.4 Other Process Managers

4 Testing

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

4.1 Using the Intel Test Suite

These instructions may be partly 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)

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)
```

5 Benchmarking

netpipe, mpptest, others (SkaMPI).

6 MPE

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

7 Windows Version

7.1 Binary distribution

The Windows binary distribution uses the Microsoft Installer. Download and execute mpich2.msi to install the binary distribution. The default 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.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.

7.2 Source distribution

If you want to use a channel other than the default socket channel you need to download the mpich2 source distribution and build an alternate channel. You must have MS Developer Studio .NET 2003 or later, perl and optionally Intel Fortran 8 or later.

- Download mpich2.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 ch3sockDebug mpich2 solution
- build the ch3sockDebug mpich2s project
- build the ch3sockRelease mpich2 solution
- build the ch3sockRelease mpich2s project
- build the Debug mpich2 solution
- build the Release mpich2 solution
- build the fortDebug mpich2 solution
- build the fortRelease mpich2 solution
- build the gfortDebug mpich2 solution
- build the gfortRelease mpich2 solution
- build the sfortDebug mpich2 solution
- build the sfortRelease mpich2 solution
- build the channel of your choice. The options are shm, ssm, sshm, ib. 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. The ssm channel uses a polling progress engine that can perform poorly when multiple processes compete for individual processors. The ib channel is for clusters with Infiniband interconnects from Mellanox.

7.3 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.

8 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. Explain all of them . . .

```
--enable and --with options recognized:
--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.
    global_monitor - a single monitor 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
    handle   - Trace handle operations
    dbg      - Add compiler -g flags
    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
--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-thread-package=package - Thread package to use.  Supported thread
packages include:
    posix or pthreads - POSIX threads
    solaris - Solaris threads (Solaris OS only)
The default package is posix.
--with-logging=name - Specify the logging library for MPICH.
--with-mpe - Build the MPE (MPI Parallel Environment) routines
--enable-weak-symbols - Use weak symbols to implement PMPI routines (default)
--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-flavor=name - Set the name to associate with this flavor of MPICH
--with-namepublisher=name - Choose the system that will support
                             MPI_PUBLISH_NAME and MPI_LOOKUP_NAME.  Options
                             include
                               no (no service available)
                               pmiext  (service using a pmi extension,
                                        usually only within the same MPD ring)
                               file:directory
                               ldap:ldapservername
                             Only no and file have been implemented so far.
--enable-sharedlibs=kind - Enable shared libraries.  kind may be
    gcc     - Standard gcc and GNU ld options for creating shared libraries
    libtool - GNU libtool
    none    - same as --disable-sharedlibs
Only gcc is currently supported

--enable-dependencies - Generate dependencies for sourcefiles.  This
            requires that the Makefile.in files are also created
            to support dependencies (see maint/updatefiles)
```