FAQ: Building Open MPI

About
Presentations
Open MPI Team
FAQ
Rollup/ALL
General information
General information
Supported systems
Contributing
Developer information
Sysadmin information
Fault Tolerance
Building
Building Open MPI
Removed MPI constructs
Compiling MPI apps
Running Jobs
Running MPI jobs
Troubleshooting
Parallel debugging
rsh/ssh
BProc
Torque / PBS Pro
Slurm
SGE
Large clusters
Tuning
General tuning
Shared memory (Vader)
TCP
IB, RoCE, iWARP
Omni-Path
Performance tools
OMPIO
UDAPL
Myrinet
Platform
OS X
AIX (unsupported)
Contrib
VampirTrace
Languages
Java
CUDA-aware
Building CUDA-aware
Running CUDA-aware
Videos
Performance
Open MPI Software
Download
Documentation
Source Code Access
Bug Tracking
Regression Testing
Version Information
Sub-Projects
Hardware Locality
Network Locality
MPI Testing Tool
Open MPI User Docs
Open Tool for Parameter Optimization
Community
Mailing Lists
Getting Help/Support
Contribute
Contact
License

Table of contents:

---

1. How do I build Open MPI?

If you have obtained a developer's checkout from Git, skip this FAQ question and consult these directions.

For everyone else, in general, all you need to do is expand the tarball, run the provided `configure` script, and then run "[make all install]". For example:

```
shell$ gunzip -c openmpi-4.0.1.tar.gz | tar xf -
shell$ cd openmpi-4.0.1
shell$ ./configure --prefix=/usr/local
<...lots of output...>
shell$ make all install
```

Note that the `configure` script supports a lot of different command line options. For example, the `--prefix` option in the above example tells Open MPI to install under the directory `/usr/local/`.

Other notable configure options are required to support specific network interconnects and back-end run-time environments. More generally, Open MPI supports a wide variety of hardware and environments, but it sometimes needs to be told where support libraries and header files are located.

Consult the `README` file in the Open MPI tarball and the output of "`configure --help`" for specific instructions regarding Open MPI's `configure` command line options.

---

2. Wow — I see a lot of errors during `configure`. Is that normal?

If `configure` finishes successfully — meaning that it generates a bunch of `Makefiles` at the end — then yes, it is completely normal.

The Open MPI `configure` script tests for a lot of things, not all of which are expected to succeed. For example, if you do not have Myrinet's GM library installed, you'll see failures about trying to find the GM library. You'll also see errors and warnings about various operating-system specific tests that are not aimed that the operating system you are running.

These are all normal, expected, and nothing to be concerned about. It just means, for example, that Open MPI will not build Myrinet GM support.

---

3. What are the default build options for Open MPI?

If you have obtained a developer's checkout from Git, you must consult these directions.

The default options for building an Open MPI tarball are:

- Compile Open MPI with all optimizations enabled
- Build shared libraries
- Build components as standalone dynamic shared object (DSO) files (i.e., run-time plugins)
- Try to find support for all hardware and environments by looking for support libraries and header files in standard locations; skip them if not found

Open MPI's configure script has a large number of options, several of which are of the form `--with-<FOO>(=DIR)`, usually with a corresponding `--with-<FOO>-libdir=DIR` option. The `(=DIR)` part means that specifying the directory is optional. Here are some examples (explained in more detail below):

- `--with-openib(=DIR)` and `--with-openib-libdir=DIR`
- `--with-mx(=DIR)` and `--with-mx-libdir=DIR`
- `--with-psm(=DIR)` and `--with-psm-libdir=DIR`
- ...etc.

As mentioned above, by default, Open MPI will try to build support for every feature that it can find on your system. If support for a given feature is not found, Open MPI will simply skip building support for it (this *usually* means not building a specific plugin).

"Support" for a given feature usually means finding both the relevant header and library files for that feature. As such, the command-line switches listed above are used to override default behavior and allow specifying whether you want support for a given feature or not, and if you *do* want support, where the header files and/or library files are located (which is useful if they are not located in compiler/linker default search paths). Specifically:

- If `--without-<FOO>` is specified, Open MPI will not even look for support for feature `FOO`. It will be treated as if support for that feature was not found (i.e., it will be skipped).
- If `--with-<FOO>` is specified with no optional directory, Open MPI's `configure` script will abort if it cannot find support for the `FOO` feature. More specifically, only compiler/linker default search paths will be searched while looking for the relevant header and library files. This option essentially tells Open MPI, "Yes, I want support for `FOO` -- it is an error if you don't find support for it."
- If `--with-<FOO>=/some/path` is specified, it is essentially the same as specifying `--with-<FOO>` but *also* tells Open MPI to add `-I/some/path/include` to compiler search paths, and try (in order) adding `-L/some/path/lib` and `-L/some/path/lib64` to linker search paths when searching for `FOO` support. If found, the relevant compiler/linker paths are added to Open MPI's general build flags. This option is helpful when support for feature `FOO` is not found in default search paths.
- If `--with-<FOO>-libdir=/some/path/lib` is specified, it *only* specifies that if Open MPI searches for `FOO` support, it should use `/some/path/lib` for the linker search path.

In general, it is usually sufficient to run Open MPI's `configure` script with no `--with-<FOO>` options if all the features you need supported are in default compiler/linker search paths. If the features you need are **not** in default compiler/linker search paths, you'll likely need to specify `--with-<FOO>` kinds of flags. However, note that it is safest to add `--with-<FOO>` types of flags if you want to *guarantee* that Open MPI builds support for feature `FOO`, regardless of whether support for `FOO` can be found in default compiler/linker paths or not — `configure` will abort if you can't find the appropriate support for `FOO`. *This may be preferable to unexpectedly discovering at run-time that Open MPI is missing support for a critical feature.*

Be sure to note the difference in the directory specification between `--with-<FOO>` and `--with-<FOO>-libdir`. The former takes a top-level directory (such that "`/include`", "`/lib`", and "`/lib64`" are appended to it) while the latter takes a single directory where the library is assumed to exist (i.e., nothing is suffixed to it).

Finally, note that starting with Open MPI v1.3, `configure` will sanity check to ensure that any directory given to `--with-<FOO>` or `--with-<FOO>-libdir` actually exists and will error if it does not. This prevents typos and mistakes in directory names, and prevents Open MPI from accidentally using a compiler/linker-default path to satisfy `FOO`'s header and library files.

---

4. Open MPI was pre-installed on my machine; should I overwrite it with a new version?

Probably not.

Many systems come with some version of Open MPI pre-installed (e.g., many Linuxes, BSD variants, and OS X. If you download a newer version of Open MPI from this web site (or one of the Open MPI mirrors), you probably do not want to overwrite the system-installed Open MPI. This is because the system-installed Open MPI is typically under the control of some software package management system (rpm, yum, etc.).

Instead, you probably want to install your new version of Open MPI to another path, such as `/opt/openmpi-<version>` (or whatever is appropriate for your system).

This FAQ entry also has much more information about strategies for where to install Open MPI.

---

5. Where should I install Open MPI?

A common environment to run Open MPI is in a "Beowulf"-class or similar cluster (e.g., a bunch of 1U servers in a bunch of racks). Simply stated, Open MPI can run on a group of servers or workstations connected by a network. As mentioned above, there are several prerequisites, however (for example, you typically must have an account on all the machines, you can `ssh` or `ssh` between the nodes without using a password etc.).

This raises the question for Open MPI system administrators: where to install the Open MPI binaries, header files, etc.? This discussion mainly addresses this question for homogeneous clusters (i.e., where all nodes and operating systems are the same), although elements of this discussion apply to heterogeneous clusters as well. Heterogeneous admins are encouraged to read this discussion and then see the heterogeneous section of this FAQ.

There are two common approaches:

1. Have a common filesystem, such as NFS, between all the machines to be used. Install Open MPI such that the installation directory is the *same value* on each node. This will *greatly* simplify user's shell startup scripts (e.g., `.bashrc`, `.cshrc`, `.profile` etc.) — the `PATH` can be set without checking which machine the user is on. It also simplifies the system administrator's job; when the time comes to patch or otherwise upgrade OMPI, only one copy needs to be modified.

   For example, consider a cluster of four machines: `inky`, `blinky`, `pinky`, and `clyde`.

   - Install Open MPI on `inky`'s local hard drive in the directory `/opt/openmpi-4.0.1`. The system administrator then mounts `inky:/opt/openmpi-4.0.1` on the remaining three machines, such that `/opt/openmpi-4.0.1` on all machines is effectively "the same". That is, the following directories all contain the Open MPI installation:

     ```
     1 inky:/opt/openmpi-4.0.1
     2 blinky:/opt/openmpi-4.0.1
     3 pinky:/opt/openmpi-4.0.1
     4 clyde:/opt/openmpi-4.0.1
     ```

   - Install Open MPI on `inky`'s local hard drive in the directory `/usr/local/openmpi-4.0.1`. The system administrator then mounts `inky:/usr/local/openmpi-4.0.1` on *all four* machines in some other common location, such as `/opt/openmpi-4.0.1` (a symbolic link can be installed on `inky` instead of a mount point for efficiency). This strategy is typically used for environments where one tree is NFS

exported, but another tree is typically used for the location of actual installation. For example, the following directories all contain the Open MPI installation:

```
inky:/opt/openmpi-4.0.1
blinky:/opt/openmpi-4.0.1
pinky:/opt/openmpi-4.0.1
clyde:/opt/openmpi-4.0.1
```

Notice that there are the same four directories as the previous example, but on `inky`, the directory is *actually* located in `/usr/local/openmpi-4.0.1`.

There is a bit of a disadvantage in this approach; each of the remote nodes have to incur NFS (or whatever filesystem is used) delays to access the Open MPI directory tree. However, both the administration ease and low cost (relatively speaking) of using a networked file system usually greatly outweighs the cost. Indeed, once an MPI application is past MPI_INIT, it doesn't use the Open MPI binaries very much.

NOTE: Open MPI, by default, uses a plugin system for loading functionality at run-time. Most of Open MPI's plugins are opened during the call to MPI_INIT. This can cause a lot of filesystem traffic, which, if Open MPI is installed on a networked filesystem, may be noticable. Two common options to avoid this extra filesystem traffic are to build Open MPI to not use plugins (see this FAQ entry for details) or to install Open MPI locally (see below).

2. If you are concerned with networked filesystem costs of accessing the Open MPI binaries, you can install Open MPI on the local hard drive of each node in your system. Again, it is *highly* advisable to install Open MPI in the *same* directory on each node so that each user's `PATH` can be set to the same value, regardless of the node that a user has logged on to.

   This approach will save some network latency of accessing the Open MPI binaries, but is typically only used where users are very concerned about squeezing every spare cycle out of their machines, or are running at extreme scale where a networked filesystem may get overwhelmed by filesystem requests for Open MPI binaries when running very large parallel jobs.

---

6. Should I install a new version of Open MPI over an old version?

We do not recommend this.

Before discussing specifics, here are some definitions that are necessary to understand:

- **Source tree**: The tree where the Open MPI source code is located. It is typically the result of expanding an Open MPI distribution source code bundle, such as a tarball.
- **Build tree**: The tree where Open MPI was built. It is always related to a specific source tree, but may actually be a different tree (since Open MPI supports VPATH builds). Specifically, this is the tree where you invoked `configure`, `make`, etc. to build and install Open MPI.
- **Installation tree**: The tree where Open MPI was installed. It is typically the "prefix" argument given to Open MPI's `configure` script; it is the directory from which you run installed Open MPI executables.

In its default configuration, an Open MPI installation consists of several shared libraries, header files, executables, and plugins (dynamic shared objects — DSOs). These installation files act together as a single entity. The specific filenames and contents of these files are subject to change between different versions of Open MPI.

KEY POINT: Installing one version of Open MPI does not uninstall another version.

If you install a new version of Open MPI over an older version, this may not remove or overwrite all the files from the older version. Hence, you may end up with an incompatible muddle of files from two different installations — which can cause problems.

The Open MPI team recommends one of the following methods for upgrading your Open MPI installation:

- Install newer versions of Open MPI into a different directory. For example, install into `/opt/openmpi-a.b.c` and `/opt/openmpi-x.y.z` for versions a.b.c and x.y.z, respectively.
- Completely uninstall the old version of Open MPI before installing the new version. The `make uninstall` process from Open MPI a.b.c build tree should completely uninstall that version from the installation tree, making it safe to install a new version (e.g., version x.y.z) into the same installation tree.
- Remove the old installation directory entirely and then install the new version. For example "`rm -rf /opt/openmpi`" *(assuming that there is nothing else of value in this tree!)* The installation of Open MPI x.y.z will safely re-create the `/opt/openmpi` tree. This method is preferable if you no longer have the source and build trees to Open MPI a.b.c available from which to "`make uninstall`".
- Go into the Open MPI a.b.c installation directory and manually remove all old Open MPI files. Then install Open MPI x.y.z into the same installation directory. This can be a somewhat painful, annoying, and error-prone process. **We do not recommend it.** Indeed, if you no longer have access to the original Open MPI a.b.c source and build trees, it may be far simpler to download Open MPI version a.b.c again from the Open MPI web site, configure it with the same installation prefix, and then run "`make uninstall`". Or use one of the other methods, above.

---

7. Can I disable Open MPI's use of plugins?

Yes.

Open MPI uses plugins for much of its functionality. Specifically, Open MPI looks for and loads plugins as dynamically shared objects (DSOs) during the call to MPI_INIT. However, these plugins can be compiled and installed in several different ways:

1. **As DSOs:** In this mode (the default), each of Open MPI's plugins are compiled as a separate DSO that is dynamically loaded at run time.
   - **Advantage:** this approach is highly flexible — it gives system developers and administrators fine-grained approach to install new plugins to an existing Open MPI installation, and also allows the removal of old plugins (i.e., forcibly disallowing the use of specific plugins) simply by removing the corresponding DSO(s).
   - **Disadvantage:** this approach causes additional filesystem traffic (mostly during MPI_INIT). If Open MPI is installed on a networked filesystem, this can cause noticeable network traffic when a large parallel job starts, for example.

2. **As part of a larger library:** In this mode, Open MPI "slurps up" the plugins and includes them in libmpi (and other libraries). Hence, *all* plugins are included in the main Open MPI libraries that are loaded by the system linker before an MPI process even starts.
   - **Advantage:** Significantly less filesystem traffic than the DSO approach. This model can be much more performant on network installations of Open MPI.
   - **Disadvantage:** Much less flexible than the DSO approach; system administrators and developers have significantly less ability to add/remove plugins from the Open MPI installation at run-time. Note that you still have *some* ability to add/remove plugins (see below), but there are limitations to what can be done.

To be clear: Open MPI's plugins can be built either as standalone DSOs or included in Open MPI's main libraries (e.g., libmpi). Additionally, Open MPI's main libraries can be built either as static or shared libraries.

You can therefore choose to build Open MPI in one of several different ways:

1. **--disable-mca-dso:** Using the `--disable-mca-dso` switch to Open MPI's `configure` script will cause all plugins to be built as part of Open MPI's main libraries — they will *not* be built as standalone DSOs. However, Open MPI will still look for DSOs in the filesystem at run-time. Specifically: this option *significantly* decreases (but does not eliminate) filesystem traffic during MPI_INIT, but does allow the flexibility of adding new plugins to an existing Open MPI installation.

Note that the `--disable-mca-dso` option does not affect whether Open MPI's main libraries are built as static or shared.

2. **--enable-static:** Using this option to Open MPI's `configure` script will cause the building of static libraries (e.g., `libmpi.a`). This option automatically implies `--disable-mca-dso`.

   Note that `--enable-shared` is also the default; so if you use `--enable-static`, Open MPI will build *both* static and shared libraries that contain all of Open MPI's plugins (i.e., libmpi.so and libmpi.a). If you want *only* static libraries (that contain all of Open MPI's plugins), be sure to *also* use `--disable-shared`.

3. **--disable-dlopen:** Using this option to Open MPI's `configure` script will do two things:
   1. Imply `--disable-mca-dso`, meaning that all plugins will be slurped into Open MPI's libraries.
   2. Cause Open MPI to not look for / open *any* DSOs at run time.

   Specifically: this option makes Open MPI not incur any additional filesystem traffic during MPI_INIT. Note that the `--disable-dlopen` option does not affect whether Open MPI's main libraries are built as static or shared.

---

8. How do I build an optimized version of Open MPI?

If you have obtained a developer's checkout from Git (or Mercurial), you must consult these directions.

Building Open MPI from a tarball defaults to building an optimized version. There is no need to do anything special.

---

9. Are VPATH and/or parallel builds supported?

Yes, both VPATH and parallel builds are supported. This allows Open MPI to be built in a different directory than where its source code resides (helpful for multi-architecture builds). Open MPI uses Automake for its build system, so

For example:

```
shell$ gtar zxf openmpi-1.2.3.tar.gz
shell$ cd openmpi-1.2.3
shell$ mkdir build
shell$ cd build
shell$ ../configure ...
<... lots of output ...>
shell$ make -j 4
```

Running `configure` from a different directory from where it actually resides triggers the VPATH build (i.e., it will configure and built itself from the directory where `configure` was run, not from the directory where `configure` resides).

Some versions of `make` support parallel builds. The example above shows GNU make's "`-j`" option, which specifies how many compile processes may be executing at any given time. We, the Open MPI Team, have found that doubling or quadrupling the number of processors in a machine can *significantly* speed up an Open MPI compile (since compiles tend to be much more IO bound than CPU bound).

---

10. Do I need any special tools to build Open MPI?

If you are building Open MPI from a tarball, you need a C compiler, a C++ compiler, and `make`. If you are building the Fortran 77 and/or Fortran 90 MPI bindings, you will need compilers for these languages as well. You do *not* need any special version of the GNU "Auto" tools (Autoconf, Automake, Libtool).

If you are building Open MPI from a Git checkout, you need some additional tools. See the source code access pages for more information.

---

11. How do I build Open MPI as a static library?

As noted above, Open MPI defaults to building shared libraries and building components as dynamic shared objects (DSOs, i.e., run-time plugins). Changing this build behavior is controlled via command line options to Open MPI's `configure` script.

Building static libraries: You can disable building shared libraries and enable building static libraries with the following options:

```
shell$ ./configure --enable-static --disable-shared ...
```

Similarly, you can build *both* static and shared libraries by simply specifying `--enable-static` (and *not* specifying `--disable-shared`), if desired.

Including components in libraries: Instead of building components as DSOs, they can also be "rolled up" and included in their respective libraries (e.g., `libmpi`). This is controlled with the `--enable-mca-static` option. Some examples:

```
shell$ ./configure --enable-mca-static=pml ...
shell$ ./configure --enable-mca-static=pml,btl-openib,btl-self ...
```

Specifically, entire frameworks and/or individual components can be specified to be rolled up into the library in a comma-separated list as an argument to `--enable-mca-static`.

---

12. When I run 'make', it looks very much like the build system is going into a loop.

Open MPI uses the GNU Automake software to build itself. Automake uses a tightly-woven set of file timestamp-based dependencies to compile and link software. This behavior, frequently paired with messages similar to:

```
Warning: File `Makefile.am' has modification time 3.6e+04 s in the future
```

typically means that you are building on a networked filesystem where the local time of the client machine that you are building on does not match the time on the network filesystem server. This will result in files with incorrect timestamps, and Automake degenerates into undefined behavior.

Two solutions are possible:

1. Ensure that the time between your network filesystem server and client(s) is the same. This can be accomplished in a variety of ways and is dependent upon your local setup; one method is to use an NTP daemon to synchronize all machines to a common time server.
2. Build on a local disk filesystem where network timestamps are guaranteed to be synchronized with the local build machine's time.

After using one of the two options, it is likely safest to remove the Open MPI source tree and re-expand the Open MPI tarball. Then you can run `configure`, `make`, and `make install`. Open MPI should then build and install successfully.

---

13. Configure issues warnings about sed and unterminated commands

Some users have reported seeing warnings like this in the final output from `configure`:

```
*** Final output
configure: creating ./config.status
config.status: creating ompi/include/ompi/version.h
sed: file ./confstatA1BhUF/subs-3.sed line 33: unterminated `s' command
sed: file ./confstatA1BhUF/subs-4.sed line 4: unterminated `s' command
config.status: creating orte/include/orte/version.h
```

These messages *usually* indicate a problem in the user's local shell configuration. Ensure that when you run a new shell, no output is sent to stdout. For example, if the output of this simple shell script is more than just the hostname of your computer, you need to go check your shell startup files to see where the extraneous output is coming from (and eliminate it):

```
#!/bin/sh
`hostname`
exit 0
```

---

14. Open MPI configured ok, but I get "Makefile:602: *** missing separator" kinds of errors when building

This is *usually* an indication that `configure` succeeded but really shouldn't have. See this FAQ entry for one possible cause.

---

15. Open MPI seems to default to building with the GNU compiler set. Can I use other compilers?

Yes.

Open MPI uses a standard Autoconf "configure" script to probe the current system and figure out how to build itself. One of the choices it makes it which compiler set to use. Since Autoconf is a GNU product, it defaults to the GNU compiler set. However, this is easily overridden on the `configure` command line. For example, to build Open MPI with the Intel compiler suite:

```
shell$ ./configure CC=icc CXX=icpc F77=ifort FC=ifort ...
```

Note that you can include additional parameters to `configure`, implied by the "..." clause in the example above.

In particular, 4 switches on the `configure` command line are used to specify the compiler suite:

- **CC**: Specifies the C compiler
- **CXX**: Specifies the C++ compiler
- **F77**: Specifies the Fortran 77 compiler
- **FC**: Specifies the Fortran 90 compiler

NOTE: The Open MPI team recommends using a single compiler suite whenever possible. Unexpected or undefined behavior can occur when you mix compiler suites in unsupported ways (e.g., mixing Fortran 77 and Fortran 90 compilers between different compiler suites is almost guaranteed not to work).

Here are some more examples for common compilers:

```
# Portland compilers
shell$ ./configure CC=pgcc CXX=pgCC F77=pgf77 FC=pgf90

# Pathscale compilers
shell$ ./configure CC=pathcc CXX=pathCC F77=pathf90 FC=pathf90

# Oracle Solaris Studio (Sun) compilers
shell$ ./configure CC=cc CXX=CC F77=f77 FC=f90
```

In all cases, the compilers must be found in your PATH and be able to successfully compile and link non-MPI applications before Open MPI will be able to be built properly.

---

16. Can I pass specific flags to the compilers / linker used to build Open MPI?

Yes.

Open MPI uses a standard Autoconf `configure` script to set itself up for building. As such, there are a number of command line options that can be passed to `configure` to customize flags that are passed to the underlying compiler to build Open MPI:

- **CFLAGS**: Flags passed to the C compiler.
- **CXXFLAGS**: Flags passed to the C++ compiler.
- **FFLAGS**: Flags passed to the Fortran 77 compiler.
- **FCFLAGS**: Flags passed to the Fortran 90 compiler.
- **LDFLAGS**: Flags passed to the linker (not language-specific). This flag is rarely required; Open MPI will *usually* pick up all LDFLAGS that it needs by itself.
- **LIBS**: Extra libraries to link to Open MPI (not language-specific). This flag is rarely required; Open MPI will *usually* pick up all LIBS that it needs by itself.
- **LD_LIBRARY_PATH**: Note that we do not recommend setting **LD_LIBRARY_PATH** via `configure`, but it is worth noting that you should ensure that your **LD_LIBRARY_PATH** value is appropriate for your build. Some users have been tripped up, for example, by specifying a non-default Fortran compiler to **FC** and **F77**, but then having Open MPI's configure script fail because the **LD_LIBRARY_PATH** wasn't set properly to point to that Fortran compiler's support libraries.

Note that the flags you specify must be compatible across all the compilers. In particular, flags specified to one language compiler must generate code that can be compiled and linked against code that is generated by the other language compilers. For example, on a 64 bit system where the compiler default is to build 32 bit executables:

```
# Assuming the GNU compiler suite
shell$ ./configure CFLAGS=-m64 ...
```

will produce 64 bit C objects, but 32 bit objects for C++, Fortran 77, and Fortran 90. These codes will be incompatible with each other, and Open MPI will build successfully. Instead, you must specify building 64 bit objects for *all* languages:

```
# Assuming the GNU compiler suite
shell$ ./configure CFLAGS=-m64 CXXFLAGS=-m64 FFLAGS=-m64 FCFLAGS=-m64 ...
```

The above command line will pass "`-m64`" to all four compilers, and therefore will produce 64 bit objects for all languages.

---

17. I'm trying to build with the Intel compilers, but Open MPI eventually fails to compile with really long error messages. What do I do?

A common mistake when building Open MPI with the Intel compiler suite is to accidentally specify the Intel C compiler as the C++ compiler. Specifically, recent versions of the Intel compiler renamed the C++ compiler "`icpc`" (it used to be "`icc`", the same as the C compiler). Users accustomed to the old name tend to specify "`icc`" as the C++ compiler, which will then cause a failure late in the Open MPI build process because a C++ code will be compiled with the C compiler. Bad Things then happen.

The solution is to be sure to specify that the C++ compiler is "`icpc`", not "`icc`". For example:

```
shell$ ./configure CC=icc CXX=icpc F77=ifort FC=ifort ...
```

For Googling purposes, here's some of the error messages that may be issued during the Open MPI compilation of C++ codes with the Intel C compiler (`icc`), in no particular order:

```
IPO Error: unresolved : _ZNSsD1Ev
IPO Error: unresolved : _ZdlPv
IPO Error: unresolved : _ZNKSs4sizeEv
components.o(.text+0x17): In function `ompi_info::open_components()':
: undefined reference to `std::basic_string<char, std::char_traits<char>, std::allocator<char> >::basic_string()'
components.o(.text+0x64): In function `ompi_info::open_components()':
: undefined reference to `std::basic_string<char, std::char_traits<char>, std::allocator<char> >::basic_string()'
components.o(.text+0x70): In function `ompi_info::open_components()':
: undefined reference to `std::string::size() const'
components.o(.text+0x7d): In function `ompi_info::open_components()':
: undefined reference to `std::string::reserve(unsigned int)'
components.o(.text+0x8d): In function `ompi_info::open_components()':
: undefined reference to `std::string::append(char const*, unsigned int)'
components.o(.text+0x9a): In function `ompi_info::open_components()':
: undefined reference to `std::string::append(std::string const&)'
components.o(.text+0xaa): In function `ompi_info::open_components()':
: undefined reference to `std::string::operator=(std::string const&)'
components.o(.text+0xb3): In function `ompi_info::open_components()':
: undefined reference to `std::basic_string<char, std::char_traits<char>, std::allocator<char> >::~basic_string()'
```

There are many more error messages, but the above should be sufficient for someone trying to find this FAQ entry via a web crawler.

---

18. When I build with the Intel compiler suite, linking user MPI applications with the wrapper compilers results in warning messages. What do I do?

When Open MPI was built with some versions of the Intel compilers on some platforms, you may see warnings similar to the following when compiling MPI applications with Open MPI's wrapper compilers:

```
shell$ mpicc hello.c -o hello
libimf.so: warning: warning: feupdateenv is not implemented and will always fail
shell$
```

This warning is generally harmless, but it can be alarming to some users. To remove this warning, pass either the `-shared-intel` or `-i-dynamic` options when linking your MPI application (the specific option depends on your version of the Intel compilers; consult your local documentation):

```
shell$ mpicc hello.c -o hello -shared-intel
shell$
```

You can also change the default behavior of Open MPI's wrapper compilers to automatically include this `-shared-intel` flag so that it is unnecessary to specify it on the command line when linking MPI applications.

---

19. I'm trying to build with the IBM compilers, but Open MPI eventually fails to compile. What do I do?

Unfortunately there are some problems between Libtool (which Open MPI uses for library support) and the IBM compilers when creating shared libraries. Currently the only workaround is to disable shared libraries and build Open MPI statically. For example:

```
shell$ ./configure CC=xlc CXX=xlc++ F77=xlf FC=xlf90 --disable-shared --enable-static ...
```

For Googling purposes, here's an error message that may be issued when the build fails:

```
xlc: 1501-216 command option --whole-archive is not recognized - passed to ld
xlc: 1501-216 command option --no-whole-archive is not recognized - passed to ld
xlc: 1501-218 file libopen-pal.so.0 contains an incorrect file suffix
xlc: 1501-228 input file libopen-pal.so.0 not found
```

---

20. I'm trying to build with the Oracle Solaris Studio (Sun) compilers on Linux, but Open MPI eventually fails to compile. What do I do?

Below are some known issues that impact Oracle Solaris Studio 12 Open MPI builds. The easiest way to work around them is simply to use the latest version of the Oracle Solaris Studio 12 compilers.

---

21. What configure options should I use when building with the Oracle Solaris Studio (Sun) compilers?

The below `configure` options are suggested for use with the Oracle Solaris Studio (Sun) compilers:

```
--enable-heterogeneous
--enable-cxx-exceptions
--enable-shared
--enable-orterun-prefix-by-default
--enable-mpi-f90
--with-mpi-f90-size=small
--disable-mpi-threads
--disable-progress-threads
--disable-debug
```

Linux only:

```
--with-openib
```

```
--without-udapl
--disable-openib-ibcm (only in v1.5.4 and earlier)
```

Solaris x86 only:

```
CFLAGS="-xtarget=generic -xarch=sse2 -xprefetch -xprefetch_level=2 -xvector=simd -xdepend=yes -xbuiltin=%all -xO5"
FFLAGS="-xtarget=generic -xarch=sse2 -xprefetch -xprefetch_level=2 -xvector=simd -stackvar -xO5"
```

Solaris SPARC only:

```
CFLAGS="-xtarget=ultra3 -m32 -xarch=sparcvis2 -xprefetch -xprefetch_level=2 -xvector=lib -xdepend=yes -xbuiltin=%all -xO5"
FFLAGS="-xtarget=ultra3 -m32 -xarch=sparcvis2 -xprefetch -xprefetch_level=2 -xvector=lib -stackvar -xO5"
```

---

22. When building with the Oracle Solaris Studio 12 Update 1 (Sun) compilers on x86 Linux, the compiler loops on btl_sm.c. Is there a workaround?

Apply Sun patch 141859-04.

You may also consider updating your Oracle Solaris Studio compilers to the latest Oracle Solaris Studio Express.

---

23. How do I build OpenMPI on IBM QS22 cell blade machines with GCC and XLC/XLF compilers?

You can use two following scripts (contributed by IBM) to build Open MPI on QS22.

Script to build OpenMPI using the GCC compiler:

```
#!/bin/bash
export PREFIX=/usr/local/openmpi-1.2.7_gcc

./configure \
        CC=ppu-gcc CPP=ppu-cpp CXX=ppu-c++ CFLAGS=-m64 \
        CXXFLAGS=-m64 FC=ppu-gfortran  FCFLAGS=-m64 \
        FFLAGS=-m64 CCASFLAGS=-m64 LDFLAGS=-m64 \
        --prefix=$PREFIX \
        --with-platform=optimized \
        --disable-mpi-profile \
        --with-openib=/usr \
        --enable-ltdl-convenience \
        --with-wrapper-cflags=-m64 \
        --with-wrapper-ldflags=-m64 \
        --with-wrapper-fflags=-m64 \
        --with-wrapper-fcflags=-m64

make
make install

cat <<EOF >> $PREFIX/etc/openmpi-mca-params.conf
mpi_paffinity_alone = 1
mpi_leave_pinned = 1
btl_openib_want_fork_support = 0
EOF

cp config.status $PREFIX/config.status
```

Script to build OpenMPI using XLC and XLF compilers:

```
#!/bin/bash
#
export PREFIX=/usr/local/openmpi-1.2.7_xl

./configure --prefix=$PREFIX \
            --with-platform=optimized \
            --disable-shared --enable-static \
            CC=ppuxlc CXX=ppuxlc++ F77=ppuxlf FC=ppuxlf90 LD=ppuld \
            --disable-mpi-profile \
            --disable-heterogeneous \
            --with-openib=/usr \
            CFLAGS="-q64 -O3 -qarch=cellppu -qtune=cellppu" \
            CXXFLAGS="-q64 -O3 -qarch=cellppu -qtune=cellppu" \
            FFLAGS="-q64 -O3 -qarch=cellppu -qtune=cellppu" \
            FCFLAGS="-q64 -O3 -qarch=cellppu -qtune=cellppu" \
            CCASFLAGS="-q64 -O3 -qarch=cellppu -qtune=cellppu" \
            LDFLAGS="-q64 -O3 -qarch=cellppu -qtune=cellppu" \
            --enable-ltdl-convenience \
            --with-wrapper-cflags="-q64 -O3 -qarch=cellppu -qtune=cellppu" \
            --with-wrapper-ldflags="-q64 -O3 -qarch=cellppu -qtune=cellppu" \
            --with-wrapper-fflags="-q64 -O3 -qarch=cellppu -qtune=cellppu" \
            --with-wrapper-fcflags="-q64 -O3 -qarch=cellppu -qtune=cellppu" \
            --enable-contrib-no-build=libnbc,vt

make
make install

cat <<EOF >> $PREFIX/etc/openmpi-mca-params.conf
mpi_paffinity_alone = 1
mpi_leave_pinned = 1
btl_openib_want_fork_support = 0
EOF

cp config.status $PREFIX/config.status
```

---

24. I'm trying to build with the PathScale 3.0 and 3.1 compilers on Linux, but all Open MPI commands seg fault. What do I do?

The PathScale compiler authors have identified a bug in the v3.0 and v3.1 versions of their compiler; you must disable certain "builtin" functions when building Open MPI:

1. With PathScale 3.0 and 3.1 compilers use the workaround options `-O2` and `-fno-builtin` in CFLAGS across the Open MPI build. For example:

```
shell$ ./configure CFLAGS="-O2 -fno-builtin" ...
```

2. With PathScale 3.2 beta and later, no workaround options are required.

---

25. All MPI C++ API functions return errors (or otherwise fail) when Open MPI is compiled with the PathScale compilers. What do I do?

This is an old issue that *seems* to be a problem when PathScale uses a back-end GCC 3.x compiler. Here's a proposed solution from the PathScale support team (from July 2010):

> The proposed work-around is to install gcc-4.x on the system and use the pathCC `-gnu4` option. Newer versions of the compiler (4.x and beyond) should have this fixed, but we'll have to test to confirm it's actually fixed and working correctly.

We don't anticipate that this will be much of a problem for Open MPI users these days (our informal testing shows that not many users are still using GCC 3.x), but this information is provided so that it is Google-able for those still using older compilers.

---

26. How do I build Open MPI with support for [my favorite network type]?

To build support for high-speed interconnect networks, you generally only have to specify the directory where its support header files and libraries were installed to Open MPI's `configure` script. You can specify where multiple packages were installed if you have support for more than one kind of interconnect — Open MPI will build support for as many as it can.

You tell `configure` where support libraries are with the appropriate `--with` command line switch. Here is the list of available switches:

- **--with-libfabric=<dir>:** Build support for OpenFabrics Interfaces (OFI), commonly known as "libfabric" (starting with the v1.10 series).
- **--with-ucx=<dir>:** Build support for the UCX library.
- **--with-mxm=<dir>:** Build support for the Mellanox Messaging (MXM) library (starting with the v1.5 series).
- **--with-verbs=<dir>:** Build support for OpenFabrics verbs (previously known as "Open IB", for Infiniband and iWARP networks). **NOTE:** Up through the v1.6.x series, this option was previously named **--with-openib**. In the v1.8.x series, it was renamed to be **--with-verbs**.
- **--with-portals4=<dir>:** Build support for the Portals v4 library (starting with the v1.7 series).
- **--with-psm=<dir>:** Build support for the PSM library.
- **--with-psm2=<dir>:** Build support for the PSM 2 library (starting with the v1.10 series).
- **--with-usnic:** Build support for usNIC networks (starting with the v1.8 series). In the v1.10 series, usNIC support is included in the libfabric library, but this option can still be used to ensure that usNIC support is specifically available.

For example:

```
shell$ ./configure --with-ucx=/path/to/ucx/installation \
  --with-libfabric=/path/to/libfabric/installation
```

These switches enable Open MPI's `configure` script to automatically find all the right header files and libraries to support the various networks that you specified.

You can verify that `configure` found everything properly by examining its output — it will test for each network's header files and libraries and report whether it will build support (or not) for each of them. Examining `configure`'s output is the **first** place you should look if you have a problem with Open MPI not correctly supporting a specific network type.

If `configure` indicates that support for your networks will be included, after you build and install Open MPI, you can run the "`ompi_info`" command and look for components for your networks. For example:

```
shell$ ompi_info | egrep ': ofi|ucx'
                 MCA rml: ofi (MCA v2.1.0, API v3.0.0, Component v4.0.0)
                 MCA mtl: ofi (MCA v2.1.0, API v2.0.0, Component v4.0.0)
                 MCA pml: ucx (MCA v2.1.0, API v2.0.0, Component v4.0.0)
                 MCA osc: ucx (MCA v2.1.0, API v2.0.0, Component v4.0.0)
```

Here's some network types that are no longer supported in current versions of Open MPI:

- **--with-scif=<dir>:** Build support for the SCIF library
  Last supported in the v3.1.x series.
- **--with-elan=<dir>:** Build support for Elan.
  Last supported in the v1.6.x series.
- **--with-gm=<dir>:** Build support for GM (Myrinet).
  Last supported in the v1.4.x series.
- **--with-mvapi=<dir>:** Build support for mVAPI (Infiniband).
  Last supported in the v1.3 series.
- **--with-mx=<dir>:** Build support for MX (Myrinet).
  Last supported in the v1.8.x series.
- **--with-portals=<dir>:** Build support for the Portals library.
  Last supported in the v1.6.x series.

---

27. How do I build Open MPI with support for Slurm / XGrid?

Slurm support is built automatically; there is nothing that you need to do.

XGrid support is built automatically if the XGrid tools are installed.

---

28. How do I build Open MPI with support for SGE?

Support for SGE first appeared in the Open MPI v1.2 series. The method for configuring it is slightly different between Open MPI v1.2 and v1.3.

For Open MPI v1.2, no extra `configure` arguments are needed as SGE support is built in automatically. After Open MPI is installed, you should see two components named *gridengine*.

```
shell$ ompi_info | grep gridengine
                 MCA ras: gridengine (MCA v1.0, API v1.3, Component v1.2.5)
                 MCA pls: gridengine (MCA v1.0, API v1.3, Component v1.2.5)
```

For Open MPI v1.3, you need to explicitly request the SGE support with the "`--with-sge`" command line switch to the Open MPI `configure` script. For example:

```
shell$ ./configure --with-sge
```

After Open MPI is installed, you should see one component named *gridengine*.

```
shell$ ompi_info | grep gridengine
                 MCA ras: gridengine (MCA v2.0, API v2.0, Component v1.3)
```

Open MPI v1.3 only has the one specific gridengine component as the other functionality was rolled into other components.

Component versions may vary depending on the version of Open MPI 1.2 or 1.3 you are using.

---

29. How do I build Open MPI with support for PBS Pro / Open PBS / Torque?

Support for PBS Pro, Open PBS, and Torque must be explicitly requested with the "`--with-tm`" command line switch to Open MPI's `configure` script. In general, the procedure is the same building support for high-speed interconnect networks, except that you use `--with-tm`. For example:

```
shell$ ./configure --with-tm=/path/to/pbs_or_torque/installation
```

After Open MPI is installed, you should see two components named "`tm`":

```
shell$ ompi_info | grep tm
                 MCA pls: tm (MCA v1.0, API v1.0, Component v1.0)
                 MCA ras: tm (MCA v1.0, API v1.0, Component v1.0)
```

Specific frameworks and version numbers may vary, depending on your version of Open MPI.

NOTE: Update to the note below (May 2006): Torque 2.1.0p0 now includes support for shared libraries and the workarounds listed below are no longer necessary. However, this version of Torque changed other things that require upgrading Open MPI to 1.0.3 or higher (as of this writing, v1.0.3 has not yet been released — nightly snapshot tarballs of what will become 1.0.3 are available at https://www.open-mpi.org/nightly/v1.0/).

NOTE: As of this writing (October 2006), Open PBS, and PBS Pro do not (i.e., they only include static libraries). Because of this, you may run into linking errors when Open MPI tries to create dynamic plugin components for TM support on some platforms. Notably, on at least some 64 bit Linux platforms (e.g., AMD64), trying to create a dynamic plugin that links against a static library will result in error messages such as:

```
relocation R_X86_64_32S against `a local symbol' can not be used when making a shared object; recompile with -fPIC
```

Note that recent versions of Torque (as of October 2006) have started shipping shared libraries and this issue does not occur.

There are two possible solutions in Open MPI 1.0.x:

1. Recompile your PBS implementation with "`-fPIC`" (or whatever the relevant flag is for your compiler to generate position-independent code) and re-install. This will allow Open MPI to generate dynamic plugins with the PBS/Torque libraries properly.

   PRO: Open MPI enjoys the benefits of shared libraries and dynamic plugins.

   CON: Dynamic plugins can use more memory at run-time (e.g., operating systems tend to align each plugin on a page, rather than densely packing them all into a single library).

   CON: This is not possible for binary-only vendor distributions (such as PBS Pro).

2. Configure Open MPI to build a static library that includes all of its components. Specifically, all of Open MPI's components will be included in its libraries — none will be discovered and opened at run-time. This does not affect user MPI code at all (i.e., the location of Open MPI's plugins is transparent to MPI applications). Use the following options to Open MPI's `configure` script:

   ```
   shell$ ./configure --disable-shared --enable-static ...
   ```

   Note that this option only changes the location of Open MPI's _default set_ of plugins (i.e., they are included in `libmpi` and friends rather than being standalone dynamic shared objects that are found/opened at run-time). This option does **not** change the fact that Open MPI will still try to open other dynamic plugins at run-time.

   PRO: This works with binary-only vendor distributions (e.g., PBS Pro).

   CON: User applications are statically linked to Open MPI; if Open MPI — or any of its default set of components — is updated, users will need to re-link their MPI applications.

Both methods work equally well, but there are tradeoffs; each site will likely need to make its own determination of which to use.

---

30. How do I build Open MPI with support for LoadLeveler?

Support for LoadLeveler will be automatically built if the LoadLeveler libraries and headers are in the default path. If not, support must be explicitly requested with the "`--with-loadleveler`" command line switch to Open MPI's `configure` script. In general, the procedure is the same building support for high-speed interconnect networks, except that you use `--with-loadleveler`. For example:

```
shell$ ./configure --with-loadleveler=/path/to/LoadLeveler/installation
```

After Open MPI is installed, you should see one or more components named "`loadleveler`":

```
shell$ ompi_info | grep loadleveler
                 MCA ras: loadleveler (MCA v1.0, API v1.3, Component v1.3)
```

Specific frameworks and version numbers may vary, depending on your version of Open MPI.

---

31. How do I build Open MPI with support for Platform LSF?

Note that only Platform LSF 7.0.2 and later is supported.

Support for LSF will be automatically built if the LSF libraries and headers are in the default path. If not, support must be explicitly requested with the "`--with-lsf`" command line switch to Open MPI's `configure` script. In general, the procedure is the same building support for high-speed interconnect networks, except that you use `--with-lsf`. For example:

```
shell$ ./configure --with-lsf=/path/to/lsf/installation
```

After Open MPI is installed, you should see a component named "`lsf`":

```
shell$ ompi_info | grep lsf
                 MCA ess: lsf (MCA v2.0, API v1.3, Component v1.3)
                 MCA ras: lsf (MCA v2.0, API v1.3, Component v1.3)
                 MCA plm: lsf (MCA v2.0, API v1.3, Component v1.3)
```

Specific frameworks and version numbers may vary, depending on your version of Open MPI.

---

32. How do I build Open MPI with processor affinity support?

Open MPI supports processor affinity for many platforms. In general, processor affinity will automatically be built if it is supported — no additional command line flags to `configure` should be necessary.

However, Open MPI will fail to build processor affinity if the appropriate support libraries and header files are not available on the system on which Open MPI is being built. Ensure that you have all appropriate "development" packages installed. For example, Red Hat Enterprise Linux (RHEL) systems typically require the `numactl-devel` packages to be installed before Open MPI will be able to build full support for processor affinity. Other OS's / Linux distros may have different packages that are required.

See this FAQ entry for more details.

---

33. How do I build Open MPI with memory affinity / NUMA support (e.g., libnuma)?

Open MPI supports memory affinity for many platforms. In general, memory affinity will automatically be built if it is supported — no additional command line flags to `configure` should be necessary.

However, Open MPI will fail to build memory affinity if the appropriate support libraries and header files are not available on the system on which Open MPI is being built. Ensure that you have all appropriate "development" packages installed. For example, Red Hat Enterprise Linux (RHEL) systems typically require the `numactl-devel` packages to be installed before Open MPI will be able to build full support for memory affinity. Other OS's / Linux distros may have different packages that are required.

See this FAQ entry for more details.

---

34. How do I build Open MPI with CUDA-aware support?

CUDA-aware support means that the MPI library can send and receive GPU buffers directly. This feature exists in the Open MPI 1.7 series and later. The support is being continuously updated so different levels of support exist in different versions. We recommend you use the latest 1.8 version for best support.

Configuring the Open MPI 1.8 series and Open MPI 1.7.3, 1.7.4, 1.7.5

With Open MPI 1.7.3 and later the `libcuda.so` library is loaded dynamically so there is no need to specify a path to it at configure time. Therefore, all you need is the path to the `cuda.h` header file.

1. Searches in default locations. Looks for `cuda.h` in /usr/local/cuda/include.

```
shell$ ./configure --with-cuda
```

2. Searches for `cuda.h` in /usr/local/cuda-v6.0/cuda/include.

```
shell$ ./configure --with-cuda=/usr/local/cuda-v6.0/cuda
```

Note that you cannot configure with **--disable-dlopen** as that will break the ability of the Open MPI library to dynamically load `libcuda.so`.

Configuring Open MPI 1.7, MPI 1.7.1 and 1.7.2

```
  --with-cuda(=DIR)       Build cuda support, optionally adding DIR/include,
                          DIR/lib, and DIR/lib64
  --with-cuda-libdir=DIR  Search for cuda libraries in DIR
```

Here are some examples of `configure` commands that enable CUDA support.

1. Searches in default locations. Looks for `cuda.h` in /usr/local/cuda/include and `libcuda.so` in /usr/lib64.

```
shell$ ./configure --with-cuda
```

2. Searches for `cuda.h` in /usr/local/cuda-v4.0/cuda/include and `libcuda.so` in default location of /usr/lib64.

```
shell$ ./configure --with-cuda=/usr/local/cuda-v4.0/cuda
```

3. Searches for `cuda.h` in /usr/local/cuda-v4.0/cuda/include and `libcuda.so` in /usr/lib64. (Same as previous example.)

```
shell$ ./configure --with-cuda=/usr/local/cuda-v4.0/cuda --with-cuda-libdir=/usr/lib64
```

If the `cuda.h` or `libcuda.so` files cannot be found, then the configure will abort.

Note: There is a bug in Open MPI 1.7.2 such that you will get an error if you configure the library with **--enable-static**. To get around this error, add the following to your `configure` line and reconfigure. This disables the build of the PML BFO which is largely unused anyways. This bug is fixed in Open MPI 1.7.3.

```
 --enable-mca-no-build=pml-bfo
```

See this FAQ entry for detals on how to use the CUDA support.

---

35. How do I *not* build a specific plugin / component for Open MPI?

The `--enable-mca-no-build` option to Open MPI's `configure` script enables you to specify a list of components that you want to skip building. This allows you to not include support for specific features in Open MPI if you do not want to.

It takes a single argument: a comma-delimited list of framework/component pairs inidicating which specific components you do not want to build. For example:

```
shell$ ./configure --enable-mca-no-build=paffinity-linux,timer-solaris
```

Note that this option is really only useful for components that would otherwise be built. For example, if you are on a machine without Myrinet support, it is not necessary to specify:

```
shell$ ./configure --enable-mca-no-build=btl-gm
```

because the `configure` script will naturally see that you do not have support for GM and will automatically skip the `gm` BTL component.

---

36. What other options to configure exist?

There are *many* options to Open MPI's `configure` script. Please run the following to get a full list (including a short description of each option):

```
shell$ ./configure --help
```

---

37. Why does compiling the Fortran 90 bindings take soooo long?

> **NOTE:** Starting with Open MPI v1.7, if you are not using gfortran, building the Fortran 90 and 08 bindings do not suffer the same performance penalty that previous versions incurred. The Open MPI developers encourage all users to upgrade to the new Fortran bindings implementation — including the new MPI-3 Fortran'08 bindings — when possible.

This is actually a design problem with the MPI F90 bindings themselves. The issue is that since F90 is a strongly typed language, we have to overload each function that takes a choice buffer with a typed buffer. For example, `MPI_SEND` has many different overloaded versions — one for each type of the user buffer. Specifically, there is an `MPI_SEND` that has the following types for the first argument:

- logical*1, logical*2, logical*4, logical*8, logical*16 (if supported)
- integer*1, integer*2, integer*4, integer*8, integer*16 (if supported)
- real*4, real*8, real*16 (if supported)
- complex*8, complex*16, complex*32 (if supported)
- character

On the surface, this is 17 bindings for `MPI_SEND`. Multiply this by every MPI function that takes a choice buffer (50) and you 850 overloaded functions. However, the problem gets worse — for each type, we also have to overload for each array dimension that needs to be supported. Fortran allows up to 7 dimensional arrays, so this becomes (17x7) = 119 versions of every MPI function that has a choice buffer argument. This makes (17x7x50) = 5,950 MPI interface functions.

To make matters even worse, consider the ~25 MPI functions that take **2** choice buffers. Functions have to be provided for all possible combinations of types. This then becomes exponential — the total number of interface functions balloons up to 6.8M.

Additionally, F90 modules must all have their functions in a single source file. Hence, all 6.8M functions must be in one `.f90` file and compiled as a single unit (currently, no F90 compiler that we are aware of can handle 6.8M interface functions in a single module).

To limit this problem, Open MPI, by default, does not generate interface functions for any of the 2-buffer MPI functions. Additionally, we limit the maximum number of supported dimensions to 4 (instead of 7). This means that we're generating (17x4*50) = 3,400 interface functions in a single F90 module. So it's far smaller than 6.8M functions, but it's still quite a lot.

This is what makes compiling the F90 module take so long.

Note, however, you can limit the maximum number of dimensions that Open MPI will generate for the F90 bindings with the configure switch `--with-f90-max-array-dim=DIM`, where `DIM` is an integer <= 7. The default value is 4. Decreasing this value makes the compilation go faster, but obviously supports fewer dimensions.

Other than this limit on dimension size, there is little else that we can do — the MPI-2 F90 bindings were unfortunately not well thought out in this regard.

Note, however, that the Open MPI team has proposed Fortran '03 bindings for MPI in a paper that was presented at the Euro PVM/MPI'05 conference. These bindings avoid all the scalability problems that are described above and have some other nice properties.

This is something that is being worked on in Open MPI, but there is currently have no estimated timeframe on when it will be available.

---

38. Does Open MPI support MPI_REAL16 and MPI_COMPLEX32?

It depends. Note that these datatypes are optional in the MPI standard.

Prior to v1.3, Open MPI supported `MPI_REAL16` and `MPI_COMPLEX32` if a portable C integer type could be found that was the same size (measured in bytes) as Fortran's `REAL*16` type. It was later discovered that even though the sizes may be the same, the bit representations between C and Fortran may be different. Since Open MPI's reduction routines are implemented in C, calling MPI_REDUCE (and related functions) with `MPI_REAL16` or `MPI_COMPLEX32` would generate undefined results (although message passing with these types in homogeneous environments generally worked fine).

As such, Open MPI v1.3 made the test for supporting `MPI_REAL16` and `MPI_COMPLEX32` more stringent: Open MPI will support these types only if:

- An integer C type can be found that has the same size (measured in bytes) as the Fortran `REAL*16` type.
- The bit representation is the same between the C type and the Fortran type.

Version 1.3.0 only checks for portable C types (e.g., `long double`). A future version of Open MPI may include support for compiler-specific / non-portable C types. For example, the Intel compiler has specific options for creating a C type that is the same as REAL*16, but we did not have time to include this support in Open MPI v1.3.0.

---

39. Can I re-locate my Open MPI installation without re-configuring/re-compiling/re-installing from source?

Starting with Open MPI v1.2.1, yes.

Background: Open MPI hard-codes some directory paths in its executables based on installation paths specified by the `configure` script. For example, if you configure with an installation prefix of `/opt/openmpi/`, Open MPI encodes in its executables that it should be able to find its help files in `/opt/openmpi/share/openmpi`.

The "installdirs" functionality in Open MPI lets you change any of these hard-coded directory paths at run time (**assuming** that you have already adjusted your `PATH` and/or `LD_LIBRARY_PATH` environment variables to the new location where Open MPI now resides). There are three methods:

1. Move an existing Open MPI installation to a new prefix: Set the `OPAL_PREFIX` environment variable before launching Open MPI. For example, if Open MPI had initially been installed to `/opt/openmpi` and the entire `openmpi` tree was later moved to `/home/openmpi`, setting `OPAL_PREFIX` to `/home/openmpi` will enable Open MPI to function properly.

2. "Stage" an Open MPI installation in a temporary location: When *creating* self-contained installation packages, systems such as RPM install Open MPI into temporary locations. The package system then bundles up everything under the temporary location into a package that can be installed into its real location later. For example, when *creating* an RPM that will be installed to `/opt/openmpi`, the RPM system will transparently prepend a "destination directory" (or "destdir") to the installation directory. As such, Open MPI will think that it is installed in `/opt/openmpi`, but it is actually temporarily installed in (for example) `/var/rpm/build.1234/opt/openmpi`. If it is necessary to *use* Open MPI while it is installed in this staging area, the `OPAL_DESTDIR` environment variable can be used; setting `OPAL_DESTDIR` to `/var/rpm/build.1234` will automatically prefix every directory such that Open MPI can function properly.

3. Overriding individual directories: Open MPI uses the GNU-specified directories (per Autoconf/Automake), and can be overridden by setting environment variables directly related to their common names. The list of environment variables that can be used is:

    - OPAL_PREFIX
    - OPAL_EXEC_PREFIX

- OPAL_BINDIR
- OPAL_SBINDIR
- OPAL_LIBEXECDIR
- OPAL_DATAROOTDIR
- OPAL_DATADIR
- OPAL_SYSCONFDIR
- OPAL_SHAREDSTATEDIR
- OPAL_LOCALSTATEDIR
- OPAL_LIBDIR
- OPAL_INCLUDEDIR
- OPAL_INFODIR
- OPAL_MANDIR
- OPAL_PKGDATADIR
- OPAL_PKGLIBDIR
- OPAL_PKGINCLUDEDIR

Note that not all of the directories listed above are used by Open MPI; they are listed here in entirety for completeness.

Also note that several directories listed above are defined in terms of other directories. For example, the `$bindir` is defined by default as `$prefix/bin`. Hence, overriding the `$prefix` (via `OPAL_PREFIX`) will automatically change the first part of the `$bindir` (which is how method 1 described above works). Alternatively, `OPAL_BINDIR` can be set to an absolute value that ignores `$prefix` altogether.

---

40. How do I statically link to the libraries of Intel compiler suite?

The Intel compiler suite, by default, dynamically links its runtime libraries against the Open MPI binaries and libraries. This can cause problems if the Intel compiler libraries are installed in non-standard locations. For example, you might get errors like:

```
error while loading shared libraries: libimf.so: cannot open shared object file:
No such file or directory
```

To avoid such problems, you can pass flags to Open MPI's configure script that instruct the Intel compiler suite to statically link its runtime libraries with Open MPI:

```
shell$ ./configure CC=icc CXX=icpc FC=ifort LDFLAGS=-Wc,-static-intel ...
```

---

41. Why do I get errors about hwloc or libevent not found?

Sometimes you may see errors similar to the following when attempting to build Open MPI:

```
...
PPFC     profile/pwin_unlock_f08.lo
PPFC     profile/pwin_unlock_all_f08.lo
PPFC     profile/pwin_wait_f08.lo
FCLD     libmpi_usempif08.la
ld: library not found for -lhwloc
collect2: error: ld returned 1 exit status
make[2]: *** [libmpi_usempif08.la] Error 1
```

This error can happen when a number of factors occur together:

1. If Open MPI's `configure` script chooses to use an "external" installation of hwloc and/or Libevent (i.e., outside of Open MPI's source tree).
2. If Open MPI's `configure` script chooses C and Fortran compilers from different suites/installations.

Put simply: if the default search library search paths differ between the C and Fortran compiler suites, the C linker may find a system-installed `libhwloc` and/or `libevent`, but the Fortran linker may not.

This may tend to happen more frequently starting with Open MPI v4.0.0 on Mac OS because:

1. In v4.0.0, Open MPI's `configure` script was changed to "prefer" system-installed versions of hwloc and Libevent (vs. preferring the hwloc and Libevent that are bundled in the Open MPI distribution tarballs).
2. In MacOS, it is common for Homebrew or MacPorts to install:
   - hwloc and/or Libevent
   - gcc and gfortran

For example, as of July 2019, Homebrew:

- Installs hwloc v2.0.4 under `/usr/local`
- Installs the Gnu C and Fortran compiler suites v9.1.0 under `/usr/local`. **However**, the C compiler executable is named `gcc-9` (not `gcc`!), whereas the Fortran compiler executable is named `gfortran`.

These factors, taken together, result in Open MPI's `configure` script deciding the following:

- The C compiler is `gcc` (which is the MacOS-installed C compiler).
- The Fortran compiler is `gfortran` (which is the Homebrew-installed Fortran compiler).
- There is a suitable system-installed hwloc in `/usr/local`, which can be found -- by the C compiler/linker -- without specifying any additional linker search paths.

The careful reader will realize that the C and Fortran compilers are from two entirely different installations. Indeed, their default library search paths are different:

- The MacOS-installed `gcc` will search `/usr/local/lib` by default.
- The Homebrew-installed `gfortran` will *not* search `/usr/local/lib` by default.

Hence, since the majority of Open MPI's source code base is in C, it compiles/links against hwloc successfully. But when Open MPI's Fortran code for the `mpi_f08` module is compiled and linked, the Homebrew-installed `gfortran` -- which does not search `/usr/local/lib` by default -- cannot find `libhwloc`, and the link fails.

There are a few different possible solutions to this issue:

1. The best solution is to always ensure that Open MPI uses a C and Fortran compiler from the same suite/installation. This will ensure that both compilers/linkers will use the same default library search paths, and all behavior should be consistent. For example, the following instructs Open MPI's `configure` script to use `gcc-9` for the C compiler, which (as of July 2019) is the Homebrew executable name for its installed C compiler:

```
shell$ ./configure CC=gcc-9 ...

# You can be precise and specify an absolute path for the C
```

```
# compiler, and/or also specify the Fortran compiler:
shell$ ./configure CC=/usr/local/bin/gcc-9 FC=/usr/local/bin/gfortran ...
```

Note that this will likely cause `configure` to *not* find the Homebrew-installed hwloc, and instead fall back to using the bundled hwloc in the Open MPI source tree (see this FAQ question for more information about the bundled hwloc and/or Libevent vs. system-installed versions).

2. Alternatively, you can simply force `configure` to select the bundled versions of hwloc and libevent, which avoids the issue altogether:

```
shell$ ./configure --with-hwloc=internal --with-libevent=internal ...
```

3. Finally, you can tell `configure` exactly where to find the external hwloc library. This can have some unintended consequences, however, because it will prefix both the C and Fortran linker's default search paths with `/usr/local/lib`:

```
shell$ ./configure --with-hwloc-libdir=/usr/local/lib ...
```

Be sure to also see this FAQ question for more information about using the bundled hwloc and/or Libevent vs. system-installed versions.

---

42. Should I use the bundled hwloc and Libevent, or system-installed versions?

From a performance perspective, there is no significant reason to choose the bundled vs. system-installed hwloc and Libevent installations. Specifically: both will likely give the same performance.

There are other reasons to choose one or the other, however.

First, some background: Open MPI has internally used hwloc and/or Libevent for almost its entire life. Years ago, it was not common for hwloc and/or Libevent to be available on many systems, so the Open MPI community decided to bundle entire copies of the hwloc and Libevent source code in Open MPI distribution tarballs.

This system worked well: Open MPI used the bundled copies of hwloc and Libevent which a) guaranteed that those packages would be available (vs. telling users that they had to separately download/install those packages before installing Open MPI), and b) guaranteed that the versions of hwloc and Libevent were suitable for Open MPI's requirements.

In the last few years, two things have changed:

1. hwloc and Libevent are now installed on many more systems by default.
2. The hwloc and Libevent APIs have stabilized such that a wide variety of hwloc/Libevent release versions are suitable for Open MPI's requirements.

While not *all* systems have hwloc and Libevent available by default (cough cough MacOS cough cough), it is now common enough that -- with the suggestion from Open MPI's downstream packagers -- starting with v4.0.0, Open MPI "prefers" system-installed hwloc and Libevent installations over its own bundled copies.

Meaning: if `configure` finds a suitable system-installed hwloc and/or Libevent, `configure` will chose to use those installations instead of the bundled copies in the Open MPI source tree.

That being said, there definitely are obscure technical corner cases and philosophical reasons to force the choice of one or the other. As such, Open MPI provides `configure` command line options that can be used to specify exact behavior in searching for hwloc and/or Libevent:

- `--with-hwloc=VALUE`: `VALUE` can be one of the following:
  - `internal`: use the bundled copy of hwloc from Open MPI's source tree.
  - `external`: use an external copy of hwloc (e.g., a system-installed copy), but only use default compiler/linker search paths to find it.
  - A directory: use an external copy of hwloc that can be found at `dir/include` and `dir/lib` or `dir/lib64`.

  Note that Open MPI *requires* hwloc -- it is invalid to specify `--without-hwloc` or `--with-hwloc=no`. Similarly, it is meaningless to specify `--with-hwloc` (with no value) or `--with-hwloc=yes`.
- `--with-hwloc-libdir=DIR`: When used with `--with-hwloc=external`, default compiler search paths will be used to find hwloc's header files, but `DIR` will be used to specify the location of the hwloc libraries. This can be necessary, for example, if both 32 and 64 bit versions of the hwloc libraries are available, and default linker search paths would find the "wrong" one.

`--with-libevent` and `--with-libevent-libdir` behave the same as the hwloc versions described above, but influence `configure`'s behavior with respect to Libevent, not hwloc.

From Open MPI's perspective, it is always safe to use the bundled copies. If there is ever a problem or conflict, you can specify `--with-hwloc=internal` and/or `--with-libevent=internal`, and this will likely solve your problem.

Additionally, note that Open MPI's `configure` will check some version and functionality aspects from system-installed hwloc / Libevent, and may still choose the bundled copies over system-installed copies (e.g., the system-installed version is too low, the system-installed version is not thread safe, ... etc.).

---

43. I'm still having problems / my problem is not listed here. What do I do?

Please see this FAQ category for troubleshooting tips and the Getting Help page — it details how to send a request to the Open MPI mailing lists.