FAQ: Compiling MPI applications

Table of contents:

1. In general, how do I build MPI applications with Open MPI?
2. Wait — what is `mpifort`? Shouldn't I use `mpif77` and `mpif90`?
3. I can't / don't want to use Open MPI's wrapper compilers. What do I do?
4. How do I override the flags specified by Open MPI's wrapper compilers? (v1.0 series)
5. How do I override the flags specified by Open MPI's wrapper compilers? (v1.1 series and beyond)
6. How can I tell what the wrapper compiler default flags are?
7. Why does "mpicc --showme <some flags>" not show any MPI-relevant flags?
8. Are there ways to just *add* flags to the wrapper compilers?
9. Why don't the wrapper compilers add "-rpath" (or similar) flags by default? (version v1.7.3 and earlier)
10. Why do the wrapper compilers add "-rpath" (or similar) flags by default? (version v1.7.4 and beyond)
11. Can I build 100% static MPI applications?
12. Can I build 100% static OpenFabrics / OpenIB / OFED MPI applications on Linux?
13. Why does it take soooo long to compile F90 MPI applications?
14. How do I build BLACS with Open MPI?
15. How do I build ScaLAPACK with Open MPI?
16. How do I build PETSc with Open MPI?
17. How do I build VASP with Open MPI?
18. Are other language / application bindings available for Open MPI?
19. Why does my legacy MPI application fail to compile with Open MPI v4.0.0 (and beyond)?

---

1. In general, how do I build MPI applications with Open MPI?

The Open MPI team **strongly** recommends that you simply use Open MPI's "wrapper" compilers to compile your MPI applications. That is, instead of using (for example) `gcc` to compile your program, use `mpicc`. Open MPI provides a wrapper compiler for four languages:

Language	Wrapper compiler name
C	`mpicc`
C++	`mpiCC`, `mpicxx`, or `mpic++` (note that `mpiCC` will not exist on case-insensitive filesystems)
Fortran	`mpifort` (for v1.7 and above) `mpif77` and `mpif90` (for older versions)

Hence, if you expect to compile your program as:

```
shell$ gcc my_mpi_application.c -o my_mpi_application
```

Simply use the following instead:

```
shell$ mpicc my_mpi_application.c -o my_mpi_application
```

Note that Open MPI's wrapper compilers do not do any actual compiling or linking; all they do is manipulate the command line and add in all the relevant compiler / linker flags and then invoke the underlying compiler / linker (hence, the name "wrapper" compiler). More specifically, if you run into a compiler or linker error, check your source code and/or back-end compiler — it is *usually* not the fault of the Open MPI wrapper compiler.

---

2. Wait — what is `mpifort`? Shouldn't I use `mpif77` and `mpif90`?

`mpifort` is a new name for the Fortran wrapper compiler that debuted in Open MPI v1.7.

It supports compiling all versions of Fortran, and *utilizing all MPI Fortran interfaces* (`mpif.h`, `use mpi`, and [use mpi_f08]). There is no need to distinguish between "Fortran 77" (which hasn't existed for 30+ years) or "Fortran 90" — just use `mpifort` to compile all your Fortran MPI applications and don't worry about what dialect it is, nor which MPI Fortran interface it uses.

Other MPI implementations will also soon support a wrapper compiler named `mpifort`, so hopefully we can move the whole world to this simpler wrapper compiler name, and eliminate the use of `mpif77` and `mpif90`.

Specifically: `mpif77` and `mpif90` are deprecated as of Open MPI v1.7. Although `mpif77` and `mpif90` still exist in Open MPI v1.7 for legacy reasons, they will likely be removed in some (undetermined) future release. It is in your interest to convert to `mpifort` now.

Also note that these names are literally just sym links to `mpifort` under the covers. So you're using `mpifort` whether you realize it or not. :-)

Basically, the 1980's called; they want their `mpif77` wrapper compiler back. *Let's let them have it.*

---

3. I can't / don't want to use Open MPI's wrapper compilers. What do I do?

We repeat the above statement: the Open MPI Team **strongly** recommends that you use the wrapper compilers to compile and link MPI applications.

If you find yourself saying, "But I don't **want** to use wrapper compilers!", please humor us and try them. See if they work for you. Be sure to let us know if they do *not* work for you.

Many people base their "wrapper compilers suck!" mentality on bad behavior from poorly-implemented wrapper compilers in the mid-1990's. Things are *much* better these days; wrapper compilers can handle almost any situation, and are far more reliable than you attempting to hard-code the Open MPI-specific compiler and linker flags manually.

That being said, there **are** some — very, very few — situations where using wrapper compilers can be problematic — such as nesting multiple wrapper compilers of multiple projects. Hence, Open MPI provides a workaround to find out what command line flags you need to compile MPI applications. There are generally two sets of flags that you need: compile flags and link flags.

```
# Show the flags necessary to compile MPI C applications
shell$ mpicc --showme:compile

# Show the flags necessary to link MPI C applications
shell$ mpicc --showme:link
```

The `--showme:*` flags work with all Open MPI wrapper compilers (specifically: `mpicc`, `mpiCC` / `mpicxx` / `mpic++`, `mpifort`, and if you really must use them, `mpif77`, `mpif90`).

Hence, if you need to use some compiler other than Open MPI's wrapper compilers, we advise you to run the appropriate Open MPI wrapper compiler with the `--showme` flags to see what Open MPI needs to compile / link, and then use those with your compiler.

NOTE: It is *absolutely not sufficient* to simply add "`-lmpi`" to your link line and assume that you will obtain a valid Open MPI executable.

NOTE: It is almost never a good idea to hard-code these results in a Makefile (or other build system). It is almost always best to run (for example) "`mpicc --showme:compile`" in a dynamic fashion to find out what you need. For example, GNU Make allows running commands and assigning their results to variables:

```
MPI_COMPILE_FLAGS = $(shell mpicc --showme:compile)
MPI_LINK_FLAGS = $(shell mpicc --showme:link)

my_app: my_app.c
        $(CC) $(MPI_COMPILE_FLAGS) my_app.c $(MPI_LINK_FLAGS) -o my_app
```

---

4. How do I override the flags specified by Open MPI's wrapper compilers? (v1.0 series)

NOTE: This answer applies to the v1.0 series of Open MPI only. If you are using a later series, please see this FAQ entry.

The wrapper compilers each construct command lines in the following form:

```
[compiler] [xCPPFLAGS] [xFLAGS] user_arguments [xLDFLAGS] [xLIBS]
```

Where `<compiler>` is replaced by the default back-end compiler for each language, and "`x`" is customized for each language (i.e., C, C++, F77, and F90).

By setting appropriate environment variables, a user can override default values used by the wrapper compilers. The table below lists the variables for each of the wrapper compilers; the *Generic* set applies to any wrapper compiler if the corresponding wrapper-specific variable is not set. For example, the value of `$OMPI_LDFLAGS` will be used with `mpicc` only if `$OMPI_MPICC_LDFLAGS` is not set.

Wrapper Compiler	Compiler	Preprocessor Flags	Compiler Flags	Linker Flags	Linker Library Flags
Generic		`OMPI_CPPFLAGS` `OMPI_CXXPPFLAGS` `OMPI_F77PPFLAGS` `OMPI_F90PPFLAGS`	`OMPI_CFLAGS` `OMPI_CXXFLAGS` `OMPI_F77FLAGS` `OMPI_F90FLAGS`	`OMPI_LDFLAGS`	`OMPI_LIBS`
mpicc	`OMPI_MPICC`	`OMPI_MPICC_CPPFLAGS`	`OMPI_MPICC_CFLAGS`	`OMPI_MPICC_LDFLAGS`	`OMPI_MPICC_LIBS`
mpicxx	`OMPI_MPICXX`	`OMPI_MPICXX_CXXPPFLAGS`	`OMPI_MPICXX_CXXFLAGS`	`OMPI_MPICXX_LDFLAGS`	`OMPI_MPICXX_LIBS`
mpif77	`OMPI_MPIF77`	`OMPI_MPIF77_F77PPFLAGS`	`OMPI_MPIF77_F77FLAGS`	`OMPI_MPIF77_LDFLAGS`	`OMPI_MPIF77_LIBS`
mpif90	`OMPI_MPIF90`	`OMPI_MPIF90_F90PPFLAGS`	`OMPI_MPIF90_F90FLAGS`	`OMPI_MPIF90_LDFLAGS`	`OMPI_MPIF90_LIBS`

NOTE: If you set a variable listed above, Open MPI will entirely replace the default value that was originally there. Hence, it is advisable to only replace these values when absolutely necessary.

---

5. How do I override the flags specified by Open MPI's wrapper compilers? (v1.1 series and beyond)

NOTE: This answer applies to the v1.1 and later series of Open MPI only. If you are using the v1.0 series, please see this FAQ entry.

The Open MPI wrapper compilers are driven by text files that contain, among other things, the flags that are passed to the underlying compiler. These text files are generated automatically for Open MPI and are customized for the compiler set that was selected when Open MPI was configured; it is *not* recommended that users edit these files.

> **Note that changing the underlying compiler may not work at all.** For example, C++ and Fortran compilers are notoriously binary incompatible with each other (sometimes even within multiple releases of the same compiler). If you compile/install Open MPI with C++ compiler XYZ and then use the `OMPI_CXX` environment variable to change the `mpicxx` wrapper compiler to use the ABC C++ compiler, your application code may not compile and/or link. The traditional method of using multiple different compilers with Open MPI is to install Open MPI multiple times; each installation should be built/installed with a different compiler. This is annoying, but it is beyond the scope of Open MPI to be able to fix.

However, there are cases where it may be necessary or desirable to edit these files and add to or subtract from the flags that Open MPI selected. These files are installed in `$pkgdatadir`, which defaults to `$prefix/share/openmpi/<wrapper_name>-wrapper-data.txt`. A few environment variables are available for run-time replacement of the wrapper's default values (from the text files):

Wrapper Compiler	Compiler	Preprocessor Flags	Compiler Flags	Linker Flags	Linker Library Flags	Data File
Open MPI wrapper compilers						
`mpicc`	`OMPI_CC`	`OMPI_CPPFLAGS`	`OMPI_CFLAGS`	`OMPI_LDFLAGS`	`OMPI_LIBS`	`mpicc-wrapper-data.txt`
`mpic++`	`OMPI_CXX`	`OMPI_CPPFLAGS`	`OMPI_CXXFLAGS`	`OMPI_LDFLAGS`	`OMPI_LIBS`	`mpic++-wrapper-data.txt`
`mpiCC`	`OMPI_CXX`	`OMPI_CPPFLAGS`	`OMPI_CXXFLAGS`	`OMPI_LDFLAGS`	`OMPI_LIBS`	`mpiCC-wrapper-data.txt`

`mpifort`	`OMPI_FC`	`OMPI_CPPFLAGS`	`OMPI_FCFLAGS`	`OMPI_LDFLAGS`	`OMPI_LIBS`	`mpifort-wrapper-data.txt`
`mpif77` **(deprecated as of v1.7)**	`OMPI_F77`	`OMPI_CPPFLAGS`	`OMPI_FFLAGS`	`OMPI_LDFLAGS`	`OMPI_LIBS`	`mpif77-wrapper-data.txt`
`mpif90` **(deprecated as of v1.7)**	`OMPI_FC`	`OMPI_CPPFLAGS`	`OMPI_FCFLAGS`	`OMPI_LDFLAGS`	`OMPI_LIBS`	`mpif90-wrapper-data.txt`
OpenRTE wrapper compilers						
`ortecc`	`ORTE_CC`	`ORTE_CPPFLAGS`	`ORTE_CFLAGS`	`ORTE_LDFLAGS`	`ORTE_LIBS`	`ortecc-wrapper-data.txt`
`ortec++`	`ORTE_CXX`	`ORTE_CPPFLAGS`	`ORTE_CXXFLAGS`	`ORTE_LDFLAGS`	`ORTE_LIBS`	`ortec++-wrapper-data.txt`
OPAL wrapper compilers						
`opalcc`	`OPAL_CC`	`OPAL_CPPFLAGS`	`OPAL_CFLAGS`	`OPAL_LDFLAGS`	`OPAL_LIBS`	`opalcc-wrapper-data.txt`
`opalc++`	`OPAL_CXX`	`OPAL_CPPFLAGS`	`OPAL_CXXFLAGS`	`OPAL_LDFLAGS`	`OPAL_LIBS`	`opalc++-wrapper-data.txt`

Note that the values of these fields can be directly influenced by passing flags to Open MPI's `configure` script. The following options are available to `configure`:

- **--with-wrapper-cflags:** Extra flags to add to CFLAGS when using `mpicc`.
- **--with-wrapper-cxxflags:** Extra flags to add to CXXFLAGS when using `mpiCC`.
- **--with-wrapper-fflags:** Extra flags to add to FFLAGS when using `mpif77` **(this option has disappeared in Open MPI 1.7 and will not return; see this FAQ entry for more details)**.
- **--with-wrapper-fcflags:** Extra flags to add to FCFLAGS when using `mpif90` and `mpifort`.
- **--with-wrapper-ldflags:** Extra flags to add to LDFLAGS when using any of the wrapper compilers.
- **--with-wrapper-libs:** Extra flags to add to LIBS when using any of the wrapper compilers.

The files cited in the above table are fairly simplistic "key=value" data formats. The following are several fields that are likely to be interesting for end-users:

- **project_short:** Prefix for all environment variables. See below.
- **compiler_env:** Specifies the base name of the environment variable that can be used to override the wrapper's underlying compiler at run-time. The full name of the environment variable is of the form <project_short>_<compiler_env>; see table above.
- **compiler_flags_env:** Specifies the base name of the environment variable that can be used to override the wrapper's compiler flags at run-time. The full name of the environment variable is of the form <project_short>_<compiler_flags_env>; see table above.
- **compiler:** The executable name of the underlying compiler.
- **extra_includes:** Relative to $installdir, a list of directories to also list in the preprocessor flags to find header files.
- **preprocessor_flags:** A list of flags passed to the preprocessor.
- **compiler_flags:** A list of flags passed to the compiler.
- **linker_flags:** A list of flags passed to the linker.
- **libs:** A list of libraries passed to the linker.
- **required_file:** If non-empty, check for the presence of this file before continuing. If the file is not there, the wrapper will abort saying that the language is not supported.
- **includedir:** Directory containing Open MPI's header files. The proper compiler "include" flag is prepended to this directory and added into the preprocessor flags.
- **libdir:** Directory containing Open MPI's library files. The proper compiler "include" flag is prepended to this directory and added into the linker flags.
- **module_option:** This field only appears in `mpif90`. It is the flag that the Fortran 90 compiler requires to declare where module files are located.

---

6. How can I tell what the wrapper compiler default flags are?

If the corresponding environment variables are not set, the wrappers will add `-I$includedir` and `-I$includedir/openmpi` (which usually map to `$prefix/include` and `$prefix/include/openmpi`, respectively) to the xFLAGS area, and add `-L$libdir` (which usually maps to `$prefix/lib`) to the xLDFLAGS area.

To obtain the values of the other flags, there are two main methods:

1. Use the `--showme` option to any wrapper compiler. For example (lines broken here for readability):

   ```
   shell$ mpicc prog.c -o prog --showme
   gcc -I/path/to/openmpi/include -I/path/to/openmpi/include/openmpi/ompi \
   prog.c -o prog -L/path/to/openmpi/lib -lmpi \
   -lopen-rte -lopen-pal -lutil -lnsl -ldl -Wl,--export-dynamic -lm
   ```

   This shows a coarse-grained method for getting the entire command line, but does not tell you what each set of flags are (xFLAGS, xCPPFLAGS, xLDFLAGS, and xLIBS).

2. Use the `ompi_info` command. For example:

   ```
   shell$ ompi_info --all | grep wrapper
      Wrapper extra CFLAGS:
    Wrapper extra CXXFLAGS:
      Wrapper extra FFLAGS:
     Wrapper extra FCFLAGS:
     Wrapper extra LDFLAGS:
        Wrapper extra LIBS: -lutil -lnsl -ldl -Wl,--export-dynamic -lm
   ```

   This installation is *only* adding options in the `xLIBS` areas of the wrapper compilers; all other values are blank (remember: the `-I`'s and `-L`'s are implicit).

   Note that the `--parsable` option can be used to obtain machine-parsable versions of this output. For example:

   ```
   shell$ ompi_info --all --parsable | grep wrapper:extra
   option:wrapper:extra_cflags:
   option:wrapper:extra_cxxflags:
   option:wrapper:extra_fflags:
   option:wrapper:extra_fcflags:
   option:wrapper:extra_ldflags:
   option:wrapper:extra_libs:-lutil -lnsl  -ldl  -Wl,--export-dynamic -lm
   ```

---

7. Why does "mpicc --showme <some flags>" not show any MPI-relevant flags?

The output of commands similar to the following may be somewhat surprising:

```
shell$ mpicc -g --showme
gcc -g
shell$
```

Where are all the MPI-related flags, such as the necessary -I, -L, and -l flags?

The short answer is that these flags are not included in the wrapper compiler's underlying command line unless the wrapper compiler sees a filename argument. Specifically (output artificially wrapped below for readability)

```
shell$ mpicc -g --showme
gcc -g
shell$ mpicc -g foo.c --showme
gcc -I/opt/openmpi/include/openmpi -I/opt/openmpi/include -g foo.c
-Wl,-u,_munmap -Wl,-multiply_defined,suppress -L/opt/openmpi/lib -lmpi
-lopen-rte -lopen-pal -ldl
```

The second command had the filename "foo.c" in it, so the wrapper added all the relevant flags. This behavior is specifically to allow behavior such as the following:

```
shell$ mpicc --version --showme
gcc --version
shell$ mpicc --version
i686-apple-darwin8-gcc-4.0.1 (GCC) 4.0.1 (Apple Computer, Inc. build 5363)
```

```
Copyright (C) 2005 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

shell$
```

That is, the wrapper compiler does not behave differently when constructing the underlying command line if `--showme` is used or not. The *only* difference is whether the resulting command line is displayed or executed.

Hence, this behavior allows users to pass arguments to the underlying compiler without intending to actually compile or link (such as passing `--version` to query the underlying compiler's version). If the wrapper compilers added more flags in these cases, some underlying compilers emit warnings.

---

8. Are there ways to just *add* flags to the wrapper compilers?

Yes!

Open MPI's `configure` script allows you to add command line flags to the wrappers on a permanent basis. The following `configure` options are available:

- **--with-wrapper-cflags=<flags>:** These flags are added into the `CFLAGS` area in the `mpicc` wrapper compiler.
- **--with-wrapper-cxxflags=<flags>:** These flags are added into the `CXXFLAGS` area in the `mpicxx` wrapper compiler.
- **--with-wrapper-fflags=<flags>:** These flags are added into the `FFLAGS` area in the `mpif77` wrapper compiler **(this option has disappeared in Open MPI 1.7 and will not return; see this FAQ entry for more details)**.
- **--with-wrapper-fcflags=<flags>:** These flags are added into the `FCFLAGS` area in the `mpif90` wrapper compiler.
- **--with-wrapper-ldflags=<flags>:** These flags are added into the `LDFLAGS` area in all the wrapper compilers.
- **--with-wrapper-libs=<flags>:** These flags are added into the `LIBS` area in all the wrapper compilers.

These configure options can be handy if you have some optional compiler/linker flags that you need both Open MPI and all MPI applications to be compiled with. Rather than trying to get all your users to remember to pass the extra flags to the compiler when compiling their applications, you can specify them with the `configure` options shown above, thereby silently including them in the Open MPI wrapper compilers — your users will therefore be using the correct flags without ever knowing it.

---

9. Why don't the wrapper compilers add "-rpath" (or similar) flags by default? (version v1.7.3 and earlier)

NOTE: Starting with Open MPI v1.7.4, the wrapper compilers *do* include `-rpath` by default. See this FAQ entry for more information.

The default installation of Open MPI tries very hard to not include any non-essential flags in the wrapper compilers. This is the most conservative setting and allows the greatest flexibility for end-users. If the wrapper compilers started adding flags to support specific features (such as run-time locations for finding the Open MPI libraries), such flags — no matter how useful to some portion of users — would almost certainly break assumptions and functionality for other users.

As a workaround, Open MPI provides several mechanisms for users to manually override the flags in the wrapper compilers:

1. First and simplest, you can add your own flags to the wrapper compiler command line by simply listing them on the command line. For example:
   ```
   shell$ mpicc my_mpi_application.c -o my_mpi_application -rpath /path/to/openmpi/install/lib
   ```
2. Use the `--showme` options to the wrapper compilers to dynamically see what flags the wrappers are adding, and modify them as appropiate. See this FAQ entry for more details.
3. Use environment variables to override the arguments that the wrappers insert. If you are using Open MPI 1.0.x, see this FAQ entry, otherwise see this FAQ entry.
4. If you are using Open MPI 1.1 or later, you can modify text files that provide the system-wide default flags for the wrapper compilers. See this FAQ entry for more details.
5. If you are using Open MPI 1.1 or later, you can pass additional flags in to the system-wide wrapper compiler default flags through Open MPI's `configure` script. See this FAQ entry for more details.

You can use one of more of these methods to insert your own flags (such as `-rpath` or similar).

---

10. Why do the wrapper compilers add "-rpath" (or similar) flags by default? (version v1.7.4 and beyond)

Prior to v1.7.4, the Open MPI wrapper compilers did not automatically add `-rpath` (or similar) flags when linking MPI application executables (for all the reasons in this FAQ entry).

Due to popular user request, Open MPI changed its policy starting with v1.7.4: by default on supported systems, Open MPI's wrapper compilers *do* insert `-rpath` (or similar) flags when linking MPI applications. You can see the exact flags added by the `--showme` functionality described in this FAQ entry.

This behavior can be disabled by configuring Open MPI with the `--disable-wrapper-rpath` CLI option.

---

11. Can I build 100% static MPI applications?

Fully static linking is not for the weak, and it is not recommended. But it is possible, with some caveats.

1. You must have static libraries available for **everything** that your program links to. This includes Open MPI; you must have used the `--enable-static` option to Open MPI's `configure` or otherwise have available the static versions of the Open MPI libraries (note that Open MPI static builds default to including all of its plugins **in** its libraries — as opposed to having each plugin in its own dynamic shared object file. So **all** of Open MPI's code will be contained in the static libraries — even what are normally contained in Open MPI's plugins). Note that some popular Linux libraries do not have static versions by default (e.g., libnuma), or require additional RPMs to be installed to get the equivalent libraries.
2. Open MPI must have been built without a memory manager. This means that Open MPI must have been configured with the `--without-memory-manager` flag. This is irrelevant on some platforms for which Open MPI does not have a memory manager, but on some platforms it is necessary (Linux). It is harmless to use this flag on platforms where Open MPI does not have a memory manager. Not having a memory manager means that Open MPI's `mpi_leave_pinned` behavior for OS-bypass networks such as InfiniBand will not work.
3. On some systems (Linux), you may see linker warnings about some files requiring dynamic libraries for functions such as `gethostname` and `dlopen`. These are ok, but do mean that you need to have the shared libraries installed. You can disable all of Open MPI's `dlopen` behavior (i.e., prevent it from trying to open any plugins) by specifying the `--disable-dlopen` flag to Open MPI's `configure` script). This will eliminate the linker warnings about `dlopen`.

For example, this is how to configure Open MPI to build static libraries on Linux:

```
shell$ ./configure --without-memory-manager --without-libnuma \
  --enable-static [...your other configure arguments...]
```

Some systems may have additional constraints about their support libraries that require additional steps to produce working 100% static MPI applications. For example, the `libibverbs` support library from OpenIB / OFED has its own plugin system (which, by default, won't work with an otherwise-static application); MPI applications need additional compiler/linker flags to be specified to create a working 100% MPI application. See this FAQ entry for the details.

---

12. Can I build 100% static OpenFabrics / OpenIB / OFED MPI applications on Linux?

Fully static linking is not for the weak, and it is not recommended. But it is possible. First, you must read this FAQ entry.

For an OpenFabrics / OpenIB / OFED application to be built statically, you must have libibverbs v1.0.4 or later (v1.0.4 was released after OFED 1.1, so if you have OFED 1.1, you will manually need to upgrade your libibverbs). Both libibverbs and your verbs hardware plugin must be available in static form.

Once all of that has been setup, run the following (artificially wrapped sample output shown below — your output may be slightly different):

```
shell$ mpicc your_app.c -o your_app --showme
gcc -I/opt/openmpi/include/openmpi \
-I/opt/openmpi/include -pthread ring.c -o ring \
-L/usr/local/ofed/lib -L/usr/local/ofed/lib64/infiniband \
-L/usr/local/ofed/lib64 -L/opt/openmpi/lib -lmpi -lopen-rte \
-lopen-pal -libverbs -lrt -Wl,--export-dynamic -lnsl -lutil -lm -ldl
```

(Or use whatever wrapper compiler is relevant — the `--showme` flag is the important part here.)

This example shows the steps for the GNU compiler suite, but other compilers will be similar. This example also assumes that the OpenFabrics / OpenIB / OFED install was rooted at `/usr/local/ofed`; some distributions install under `/usr/ofed` (or elsewhere). Finally, some installations use the library directory "lib64" while others use "lib". Adjust your directory names as appropriate.

Take the output of the above command and run it manually to compile and link your application, adding the following highlighted arguments:

```
shell$ gcc -static -I/opt/openmpi/include/openmpi \
  -I/opt/openmpi/include -pthread ring.c -o ring \
  -L/usr/local/ofed/lib -L/usr/local/ofed/lib64/infiniband \
  -L/usr/local/ofed/lib64 -L/opt/openmpi/lib -lmpi -lopen-rte \
  -lopen-pal -Wl,--whole-archive -libverbs /usr/local/ofed/lib64/infiniband/mthca.a \
  -Wl,--no-whole-archive -lrt -Wl,--export-dynamic -lnsl -lutil \
  -lm -ldl
```

Note that the **mthca.a** file is the verbs plugin for Mellanox HCAs. If you have an HCA from a different vendor (such as IBM or QLogic), use the appropriate filename (look in `$ofed_libdir/infiniband` for verbs plugin files for your hardware).

Specifically, these added arguments do the following:

- `-static`: Tell the linker to generate a static executable.
- `-Wl,--whole-archive`: Tell the linker to include the entire `ibverbs` library in the executable.
- `$ofed_root/lib64/infiniband/mthca.a`: Include the Mellanox verbs plugin in the executable.
- `-Wl,--no-whole-archive`: Tell the linker the return to the default of not including entire libraries in the executable.

You can either add these arguments in manually, or you can see this FAQ entry to modify the default behavior of the wrapper compilers to hide this complexity from end users (but be aware that if you modify the wrapper compilers' default behavior, **all** users will be creating static applications!).

13. Why does it take soooo long to compile F90 MPI applications?

> **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 unfortunately due to a design flaw in the MPI F90 bindings themselves.

The answer to this question is exactly the same as it is for why it takes so long to compile the MPI F90 bindings in the Open MPI implementation; please see this FAQ entry for the details.

14. How do I build BLACS with Open MPI?

The `blacs_install.ps` file (available from that web site) describes how to build BLACS, so we won't repeat much of it here (especially since it might change in future versions). These instructions only pertain to making Open MPI work correctly with BLACS.

After selecting the appropriate starting `Bmake.inc`, make the following changes to Sections 1, 2, and 3. The example below is from the `Bmake.MPI-SUN4SOL2`; your `Bmake.inc` file may be different.

```
# Section 1:
# Ensure to use MPI for the communication layer

   COMMLIB = MPI

# The MPIINCdir macro is used to link in mpif.h and
# must contain the location of Open MPI's mpif.h.
# The MPILIBdir and MPILIB macros are irrelevant
# and should be left empty.

   MPIdir = /path/to/openmpi-4.0.2
   MPILIBdir =
   MPIINCdir = $(MPIdir)/include
   MPILIB =

# Section 2:
# Set these values:

   SYSINC =
   INTFACE = -Df77IsF2C
   SENDIS =
   BUFF =
   TRANSCOMM = -DUseMpi2
   WHATMPI =
   SYSERRORS =

# Section 3:
# You may need to specify the full path to
# mpif77 / mpicc if they aren't already in
# your path.

   F77            = mpif77
   F77LOADFLAGS   =

   CC             = mpicc
   CCLOADFLAGS    =
```

The remainder of the values are fairly obvious and irrelevant to Open MPI; you can set whatever optimization level you want, etc.

If you follow the rest of the instructions for building, BLACS will build correctly and use Open MPI as its MPI communication layer.

15. How do I build ScaLAPACK with Open MPI?

The `scalapack_install.ps` file (available from that web site) describes how to build ScaLAPACK, so we won't repeat much of it here (especially since it might change in future versions). These instructions only pertain to making Open MPI work correctly with ScaLAPACK. These instructions assume that you have built and installed BLACS with Open MPI.

```
# Make sure you follow the instructions to build BLACS with Open MPI,
# and put its location in the following.

   BLACSdir      = ...path where you installed BLACS...

# The MPI section is commented out.  Uncomment it. The wrapper
# compiler will handle SMPLIB, so make it blank. The rest are correct
# as is.

   USEMPI        = -DUsingMpiBlacs
   SMPLIB        =
   BLACSFINIT    = $(BLACSdir)/blacsF77init_MPI-$(PLAT)-$(BLACSDBGLVL).a
   BLACSCINIT    = $(BLACSdir)/blacsCinit_MPI-$(PLAT)-$(BLACSDBGLVL).a
   BLACSLIB      = $(BLACSdir)/blacs_MPI-$(PLAT)-$(BLACSDBGLVL).a
   TESTINGdir    = $(home)/TESTING

# The PVMBLACS setup needs to be commented out.

   #USEMPI        =
   #SMPLIB        = $(PVM_ROOT)/lib/$(PLAT)/libpvm3.a -lnsl -lsocket
   #BLACSFINIT    =
   #BLACSCINIT    =
   #BLACSLIB      = $(BLACSdir)/blacs_PVM-$(PLAT)-$(BLACSDBGLVL).a
   #TESTINGdir    = $(HOME)/pvm3/bin/$(PLAT)

# Make sure that the BLASLIB points to the right place.  We built this
# example on Solaris, hence the name below.  The Linux version of the
# library (as of this writing) is blas_LINUX.a.

   BLASLIB       = $(LAPACKdir)/blas_solaris.a

# You may need to specify the full path to mpif77 / mpicc if they
# aren't already in your path.

   F77            = mpif77
   F77LOADFLAGS   =

   CC             = mpicc
   CCLOADFLAGS    =
```

The remainder of the values are fairly obvious and irrelevant to Open MPI; you can set whatever optimization level you want, etc.

If you follow the rest of the instructions for building, ScaLAPACK will build correctly and use Open MPI as its MPI communication layer.

---

16. How do I build PETSc with Open MPI?

The only special configuration that you need to build PETSc is to ensure that Open MPI's wrapper compilers (i.e., `mpicc` and `mpif77`) are in your `$PATH` before running the PETSc `configure.py` script.

PETSc should then automatically find Open MPI's wrapper compilers and correctly build itself using Open MPI.

---

17. How do I build VASP with Open MPI?

The following was reported by an Open MPI user who was able to successfully build and run VASP with Open MPI:

I just compiled the latest VASP v4.6 using Open MPI v1.2.1, ifort v9.1, ACML v3.6.0, BLACS with patch-03 and Scalapack v1.7.5 built with ACML.

I configured Open MPI with `--enable-static` flag.

I used the VASP supplied `makefile.linux_ifc_opt` and only corrected the paths to the ACML, scalapack, and BLACS dirs (I didn't lower the optimization to `-O0` for mpi.f like I suggested before). The `-D`'s are standard except I get a little better performance with `-DscaLAPACK` (I tested it with out this option too):

```
CPP    = $(CPP_) -DMPI  -DHOST="LinuxIFC" -DIFC \
     -Dkind8 -DNGZhalf -DCACHE_SIZE=4000 -DPGF90 -Davoidalloc \
     -DMPI_BLOCK=2000  \
     -Duse_cray_ptr -DscaLAPACK
```

Also, Blacs and Scalapack used the `-D`'s suggested in the Open MPI FAQ.

---

18. Are other language / application bindings available for Open MPI?

Other MPI language bindings and application-level programming interfaces have been been written by third parties. Here are a link to some of the available packages:

...we used to maintain a list of links here. But the list changes over time; projects come, and projects go. Your best bet these days is simply to use Google to find MPI bindings and application-level programming interfaces.

---

19. Why does my legacy MPI application fail to compile with Open MPI v4.0.0 (and beyond)?

Starting with v4.0.0, Open MPI — by default — removes the prototypes for MPI symbols that were deprecated in 1996 and finally removed from the MPI standard in MPI-3.0 (2012).

See this FAQ category for much more information.