Compiling code on NCAR systems

Several C/C++ and Fortran compilers are available on all NCAR HPC systems. The information on this page applies to all of those systems except where noted.

Page contents

Compiler commands

All supported compilers are available via the **module** utility. After loading the compiler module you want to use, refer to the **Compilers available on NCAR systems** table below to identify and run the appropriate compilation wrapper command.

If your script already includes one of the following generic MPI commands, there is no need to change it:

- **mpif90, mpif77, ftn**
- **mpicc, cc**
- **mpiCC, CC**

Build any libraries that you need to support an application with the same compiler, compiler version, and compatible flags used to compile the other parts of the application, including the main executable(s). Also, before you run the applications, be sure you have loaded the same module/version environment i n which you created the applications. This will help you avoid job failures that can result from missing MPI launchers and library routines.

Compiler man pages

To refer to the **man** page for a compiler, log in to the system where you intend to use it, load the module, then execute **man** for the compiler. For example:

```
module load nvhpc
man nvfortran
```

You can also use **-help** flags for a description of the command-line options for each compiler. Follow this example:

```
ifort -help
nvfortran -help[=option]
```

Use compiler diagnostic flags to identify potential problems while compiling the code.

Where to compile

Choose where to compile your code based on where you intend to run your code. Note the types of processors and operating systems as shown in the following table.

System	Processor	Environment/OS

Derecho	AMD EPYC 7763 Milan	Cray Linux Environment / SUSE Linux
Casper DAV nodes	Intel Skylake (default) Intel Cascade Lake	CentOS
Cheyenne	Intel Broadwell	SUSE Enterprise Linux

The operating systems provide different versions of some standard libraries, which may be incompatible with each other. If your code will run on both Skylake and Cascade Lake nodes on Casper, specify Skylake nodes.

Even if you take care to build your programs for portability, you will achieve superior performance if you compile your code on the cluster where you intend for it to run.

Compile on Derecho if…

- You want to aggressively optimize CPU performance.
- Your programs use GPU tools like OpenGL, CUDA, and OpenACC.
- You want to use the Cray compiler suite.

Compile on Casper if...

- You want to ensure that your code will run on Casper nodes.
- You want to use the latest CPU optimizations in Intel's Skylake or Cascade Lake architecture.
- Your programs use GPU tools like OpenGL, CUDA, and OpenACC.

Compile on Cheyenne if…

- You will run your code only on Cheyenne.

You can compile your code in a batch job submitted to any queue using your login environment (**not** on a login node). To ensure smooth operation for everyone, CISL recommends users avoid building their complex codes (e.g. WRF, MPAS, CESM) on login nodes.

Changing compilers

To change from one compiler to another, use **module swap**. In this example, you are switching from Intel to NVIDIA:

```
module swap intel nvhpc
```

When you load a compiler module or change to a different compiler, the system makes other compatible modules available. This helps you establish a working environment and avoid conflicts.

If you need to link your program with a library*, use **module load** to load the library as in this example:

```
module load netcdf
```

Then, you can invoke the desired compilation command without adding link options such as **-l netcdf**. Here's an example:

```
mpif90 foo.f90
```

Compiling CPU code

Using the Cray compiler collection

Derecho users have access to the Cray Compiling Environment (CCE) using the **cce** module. The compiler collection includes **cc**, **CC**, and **ftn** for compiling C, C++, and Fortran codes. To see which versions of the compiler are available, use the **module avail** command:

```
module avail cce
```

CCE base compilers are available by default in the **ncarcompilers** module. Loading the **ncarcompilers** module simplifies building code with dependencies such as netCDF. For example, compiling a simple Fortran code using netCDF is as follows with the compiler wrappers:

```
ftn -o mybin -lnetcdff mycode.f90
```

Meanwhile, if you did not have the **ncarcompilers** module loaded, you would need to run the following command instead, with the linker flags and include-paths:

```
ftn -I/path/to/netcdf/include -L/path/to/netcdf/lib -lnetcdff -o mybin mycode.f90
```

Using Cray MPICH MPI

Unlike other MPI libraries, Cray MPICH does **NOT** provide MPI wrapper commands like **mpicc**, **mpicxx**, and **mpif90**. Rather, use the same **cc**, **CC**, and **ftn** commands you use to compile a serial code. The Cray Programming Environment (CPE) will add MPI build flags to your commands whenever you have the **cray-mpich** module loaded.

As many application build systems expect the MPI wrappers, our **ncarcompilers** module will translate a call to "**mpicc**" to "**cc**" (and likewise for the other languages) as a convenience, typically eliminating the need to alter pre-existing build scripts.

Cray MPICH also supports GPU devices. If you are using an MPI application compiled with GPU support, enable CUDA functionality by loading a **cuda** mo dule and setting or exporting this environment variable before calling the MPI launcher in your job by including this in your script:

```
MPICH_GPU_SUPPORT_ENABLED=1
```

Also, if your GPU-enabled MPI application makes use of managed memory, you also need to set this environment variable:

```
MPICH_GPU_MANAGED_MEMORY_SUPPORT_ENABLED=1
```

At runtime, you will also need to pass information about job parallelism to the **mpiexec** (or **mpirun** / **aprun**) launcher because this information is not automatically taken from the PBS job script. You can pass this information by setting environment variables or by using **mpiexec** options. Full details of runtime settings for launching parallel programs can be found by running **man mpiexec**.

The primary settings you will need are:

- the number of mpi ranks (-n / PALS_NRANKS)
- the number of ranks per node (-ppn / PALS_PPN)
- the number of OpenMP threads or CPUs to associate with each rank (-d / PALS_DEPTH)
- binding options (--cpu-bind / PALS_CPU_BIND)

Cray MPICH has many tunable parameters you can set through environment variables. Run **man mpi** for a complete listing of these environment variables.

Example PBS select statements and corresponding MPI launch options are shown below for binding a hybrid MPI + OpenMP application (144 MPI ranks, and 4 OpenMP threads per MPI rank, which requires 5 nodes but does not fully subscribe the last node). Examples of both methods – setting environment variables and passing options to **mpiexec** – are provided.

Environment variable example

```
#PBS -l select=5:ncpus=128:mpiprocs=32:ompthreads=4
export PALS_NRANKS=144
export PALS_PPN=32
export PALS_DEPTH=4
export PALS_CPU_BIND=depth
mpiexec ./a.out
```

mpiexec options example

```
#PBS -l select=5:ncpus=128:mpiprocs=32:ompthreads=4
mpiexec --cpu-bind depth -n 144 -ppn 32 -d 4 ./a.out
```

Other compilers

These additional compilers are available on Derecho.

- NVIDIA's HPC SDK
- Intel compilers
- the GNU Compiler Collection (GCC)

When using non-Cray compilers, you can use either the compiler collection's own commands (e.g., **ifort**, **nvfortran**) or the equivalent CPE command (e.g., **ftn**) as long as you have loaded your desired compiler module. If you do not have the **ncarcompilers** module loaded and you are using the **cray-mpich** M PI, you will need to use a CPE command.

Compilers available on NCAR systems

Compiler	Language	Commands for serial programs	Commands for programs using MPI	Flags to enable OpenMP (for serial and MPI)
Intel (Classic/OneAPI)**	Fortran	ifort / ifx foo.f90 **	mpif90 foo.f90	-qopenmp
	C	icc / icx foo.c **	mpicc foo.c	-qopenmp
	C++	icpc / icpx foo.C **	mpicxx foo.C	-qopenmp
NVIDIA HPC SDK	Fortran	nvfortran foo.f90	mpif90 foo.f90	-mp
	C	nvc foo.c	mpicc foo.c	-mp
	C++	nvc++ foo.C	mpicxx foo.C	-mp
GNU (GCC)	Fortran	gfortran foo.f90	mpif90 foo.f90	-fopenmp
	C	gcc foo.c	mpicc foo.c	-fopenmp
	C++	g++ foo.C	mpicxx foo.C	-fopenmp
Cray Compiler (Derecho only)	Fortran	ftn foo.f90	mpif90 foo.f90***	-fopenmp
	C	cc foo.c	mpicc foo.c***	-fopenmp
	C++	CC foo.C	mpicxx foo.C***	-fopenmp

**** Intel OneAPI is a cross-platform toolkit that supports C, C++, Fortran, and Python programming languages and replaces Intel Parallel Studio. Derecho supports both Intel OneAPI and Intel Classic Compilers. Intel is planning to retire the Intel Classic compilers and is moving toward Intel OneAPI. Intel Classic Compiler commands (ifort, icc, and icpc) will be replaced by the Intel OneAPI compilers (ifx, icx, and icpx).**

***** Please note that mpi wrappers are not available by default using Cray compilers but the ncarcompilers module will translate a call to "mpicc" to "cc" (and likewise for the other languages) as a convenience.**

Using Intel compilers

The Intel compiler suite is available via the **intel** module. It includes compilers for C, C++, and Fortran codes. It is **NOT** loaded by default.

To see which versions are available, use the **module avail** command.

```
module avail intel
```

To load the default Intel compiler, use **module load** without specifying a version.

```
module load intel
```

To load a different version, specify the version number when loading the module.

Similarly, you can swap your current compiler module to Intel by using the **module swap** command.

```
module swap cce/14.0.3 intel
```

The table below provides a quick summary of the compile commands or flags needed to compile your C, C++, and Fortran codes using the Intel compilers.

LANGUAGE	SERIAL PROGRAMS COMPILE COMMAND	COMMANDS FOR PROGRAMS USING MPI	FLAGS TO ENABLE OPENMP (FOR SERIAL AND MPI)
Fortran	ifort foo.f90	mpif90 foo.f90	-qopenmp

C	icc foo.c	mpicc foo.c	-qopenmp
C++	icpc foo.C	mpicxx foo.C	-qopenmp

Extensive documentation for using the Intel compilers is available online here. To review the manual page for a compiler, run the **man** command for it as in this example:

```
man ifort
```

Optimizing your code with Intel compilers

Intel compilers provide several different optimization and vectorization options. By default, they use the **-O2** option, which includes some optimizations.

Using **-O3** instead will provide more aggressive optimizations that may not improve the performance of some programs, while **-O1** enables minimal optimization. A higher level of optimization might increase your compile time significantly.

You can also disable any optimization by using **-O0**.

Be aware that compiling CPU code with the Intel compiler on Derecho is significantly different from using the Intel compiler on the Cheyenne system. Flags that are commonly used on Cheyenne might cause Derecho jobs to fail or run much more slowly than otherwise possible.

- **DO use on Derecho:** -march=core-avx2
- **Do NOT use on Derecho:** -xHost , -axHost , -xCORE-AVX2 , -axCORE-AVX2

Examples

To compile and link a single Fortran program and create an executable, follow this example:

```
ifort filename.f90 -o filename.exe
```

To enable multi-threaded parallelization (OpenMP), include the **-qopenmp** flag as shown here:

```
ifort -qopenmp filename.f90 -o filename.exe
```

Compiling GPU code

On Derecho, GPU applications should be built with either the Cray compilers or the NVIDIA HPC SDK compilers and libraries. In the following examples, we demonstrate the use of NVIDIA's tools.

Additional compilation flags for GPU code will depend in large part on which GPU-programming paradigm is being used (e.g., OpenACC, OpenMP, CUDA) and which compiler collection you have loaded. The following examples show basic usage, but note that many customizations and optimizations are possible. You are encouraged to read the relevant man page for the compiler you choose.

OpenACC

To compile with OpenACC directives, simply add the **-acc** flag to your invocation of nvc, nvc++, or nvfortan. A Fortran example:

```
nvfortran -o acc_bin -acc acc_code.f90
```

You can gather more insight into GPU acceleration decisions made by the compiler by adding **-Minfo=accel** to your invocation. Using compiler options, you can also specify which GPU architecture to target. This example will request compilation for both V100 (as on Casper) and A100 GPUs (as on Derecho):

```
nvfortran -o acc_bin -acc -gpu=cc70,cc80 acc_code.f90
```

Specifying multiple acceleration targets will increase the size of the binary and the time it takes to compile the code.

OpenMP

Using OpenMP to offload code to the GPU is similar to using OpenACC. To compile a code with OpenMP offloading, use the **-mp=gpu** flag. The aforementioned diagnostic and target flags also apply to OpenMP offloading.

```
nvfortran -o omp_gpu -mp=gpu omp.f90
```

CUDA

The process for compiling CUDA code depends on whether you are using C++ or Fortran. For C++, the process often involves multiple stages in which you first use **nvcc**, the NVIDIA CUDA compiler, and then your C++ compiler of choice.

```
nvcc -c -arch=sm_80 cuda_code.cu
g++ -o cuda_bin -lcuda -lcudart main.cpp cuda_code.o
```

Using the **nvcc** compiler driver with a non-NVIDIA C++ compiler requires loading a **cuda** environment module in addition to the compiler of choice.

The compiler handles CUDA code directly, so the compiler you use must support CUDA. This means you should use **nvfortran**. If your source code file ends with the **.cuf** extension, nvfortran will enable CUDA automatically. Otherwise, you can specify the **-Mcuda** flag to the compiler.

```
nvfortran -Mcuda -o cf_bin cf_code.f90
```

GPU compilers for Casper

To compile CUDA code to run on the Casper data analysis and visualization nodes, use the appropriate NVIDIA compiler command:

- **nvc** – NVIDIA C compiler
- **nvcc** – NVIDIA CUDA compiler (Using nvcc requires a C compiler to be present in the background; nvc, icc, or gcc, for example.)
- **nvfortran** – CUDA Fortran

For more information on compiling code on Casper nodes, see:

- Compiling GPU code on Casper nodes
- Compiling multi-GPU MPI-CUDA code on Casper

Native commands

We recommend using the module wrapper commands described above. However, if you prefer to invoke the compilers directly, unload the NCAR default compiler wrapper environment by entering this on your command line:

```
module unload ncarcompilers
```

You can still use the environment variables that are set by the modules that remain loaded, as shown in the following examples of invoking compilers directly to compile a Fortran program.

Intel compiler

```
ifort -o a.out $NCAR_INC_<PROGRAM> program_name.f $NCAR_LDFLAGS_<PROGRAM> $NCAR_LIBS_<PROGRAM>
```

NVIDIA HPC compiler

```
nvfortran -o a.out $NCAR_INC_<PROGRAM> program_name.f $NCAR_LDFLAGS_<PROGRAM> $NCAR_LIBS_<PROGRAM>
```

GNU compiler collection (GCC)

```
gfortran -o a.out $NCAR_INC_<PROGRAM> program_name.f $NCAR_LDFLAGS_<PROGRAM> $NCAR_LIBS_<PROGRAM>
```

*** In addition to multiple compilers, CISL keeps available multiple versions of libraries to accommodate a wide range of users' needs. Rather than rely on the environment variable LD_LIBRARY_PATH to find the correct libraries dynamically, we encode library paths within the binaries when you build Executable and Linkable Format (ELF) executables. To do this, we use RPATH rather than LD_LIBRARY_PATH to set the necessary paths to shared libraries.**

This enables your executable to work regardless of updates to new default versions of the various libraries; it doesn't have to search dynamically at run time to load them. It also means you don't need to worry about setting the variable or loading another module, greatly reducing the likelihood of runtime errors.