Compiling & Debugging

Compiling

At NYU HPC, most compiling is done using the Intel Compiler Suite. To get the best performance, we recommend that you use these compilers because the executable files perform better than the executable files compiled with the GNU compilers. The table below shows the available compilers and their invocation:

<table>
<thead>
<tr>
<th>Compiler Type</th>
<th>Invocation</th>
<th>Source</th>
<th>Documentation</th>
<th>Cluster availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>C Compilers</td>
<td>c99</td>
<td>Standard C</td>
<td>man c99</td>
<td>usq, bowery, cardiac</td>
</tr>
<tr>
<td></td>
<td>gcc</td>
<td>GNU C</td>
<td>man gcc</td>
<td>usq, bowery, cardiac</td>
</tr>
<tr>
<td></td>
<td>icc</td>
<td>Intel C</td>
<td>man icc</td>
<td>usq, bowery, cardiac</td>
</tr>
<tr>
<td>C++ Compilers</td>
<td>g++</td>
<td>GNU C++</td>
<td>man g++</td>
<td>usq, bowery, cardiac</td>
</tr>
<tr>
<td></td>
<td>icpc</td>
<td>Intel C++</td>
<td>man icpc</td>
<td>usq, bowery, cardiac</td>
</tr>
</tbody>
</table>

Click here for the updated HPC Wiki
Fortran Compilers

<table>
<thead>
<tr>
<th>Name</th>
<th>Invocation</th>
<th>Source</th>
<th>Documentation</th>
<th>Cluster availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>gfortan</td>
<td>man gfortan</td>
<td>usq, bowery, cardiac</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ifort</td>
<td>man ifort</td>
<td>usq, bowery, cardiac</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Useful Information**
Other compiler versions can be loaded using the `modules`.

**Handy Hint**
Add this line before loading any modules in your batch script to declare where the "module" program is located.

For batch scripts in sh/bash

```
source /etc/profile.d/env-modules.sh
```

For batch scripts in csh/tcsh

```
source /etc/profile.d/env-modules.csh
```

**Note**
For online manuals and guides, please see our Finding documentation section.

**MPI Wrappers**

NYU HPC provides two major families of MPI wrappers used to compile MPI parallel source code: These are MVAPICH and OpenMPI. OpenMPI is a product of the OpenMPI open source organization; whereas MVAPICH is a product of Ohio State University. Both MPI compilers are available in the GNU version. We recommend the Intel version to get better performance. The table below shows the wrappers available at NYU HPC.

<table>
<thead>
<tr>
<th>MPI Wrapper</th>
<th>Invocation</th>
<th>Source</th>
<th>Documentation</th>
<th>Cluster availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVAPICH</td>
<td>mpif77</td>
<td>MPI Fortran77</td>
<td>man mpif77</td>
<td>usq, bowery, cardiac</td>
</tr>
<tr>
<td>MVAPICH</td>
<td>mpif90</td>
<td>MPI Fortran90</td>
<td>man mpif90</td>
<td>usq, bowery, cardiac</td>
</tr>
<tr>
<td>MVAPICH</td>
<td>mpicc</td>
<td>MPI C</td>
<td>man mpicc</td>
<td>usq, bowery, cardiac</td>
</tr>
<tr>
<td>MVAPICH</td>
<td>mpiCC</td>
<td>MPI C++</td>
<td>man mpiCC</td>
<td>usq, bowery, cardiac</td>
</tr>
<tr>
<td>OpenMPI</td>
<td>mpif77</td>
<td>MPI Fortran77</td>
<td>man mpif77</td>
<td>usq, bowery, cardiac</td>
</tr>
<tr>
<td>OpenMPI</td>
<td>mpif90</td>
<td>MPI Fortran90</td>
<td>man mpif90</td>
<td>usq, bowery, cardiac</td>
</tr>
<tr>
<td>OpenMPI</td>
<td>mpicc</td>
<td>MPI C</td>
<td>man mpicc</td>
<td>usq, bowery, cardiac</td>
</tr>
<tr>
<td>OpenMPI</td>
<td>mpiCC</td>
<td>MPI C++</td>
<td>man mpiCC</td>
<td>usq, bowery, cardiac</td>
</tr>
</tbody>
</table>

**Note**
For online manuals and guides, please see our Finding documentation section.

**Use Compute Nodes to Compile Jobs**

All compilers are available on both the login nodes and compute nodes. However, for all source code development, compilation, debugging and test, please initiate an interactive session using the compute node(s). This is to prevent the overloading of the login nodes. Access to all compute nodes is controlled by the scheduler.

Please use good housekeeping practices: After compiling, save your source code to the /home file system, and move important files to /archive on a regular basis. If you are working with a version control system, move files to an SVN system.

Use the following command to initiate an interactive session on a compute node on the cluster:

```
$ qsub -I -q interactive -l nodes=1:ppn=8,walltime=04:00:00
```
Use the following command to initiate an interactive session that uses an X Windows session to a compute node on the cluster:

```
$ qsub -I -X -q interactive -l nodes=1:ppn=8,walltime=04:00:00
```

**Note**
If you did not enable X Windows when you logged on, you will not be able to enable it here. For the procedure on enabling X Windows at login, please see the section on Accessing the Cluster > Login Steps > Step 5 Access.

After you have finished compiling your job and have logged out of the compute node, you will be dropped back into the login node. Remember, the maximum time per interactive session is four hours, after which time the scheduler will end your session without warning.

### Environment Modules

NYU HPC uses an open source software package called "Environment Modules," (or Modules for short) which allows you to add various path definitions to your shell environment. Default compilers, applications and libraries can be set by individual or combinations of Modules commands. Modules are not applications, rather they simply add the location of applications to your environment. You can list the available Modules using the command:

```
$ module avail
```

You can load a module, in this case the C/C++ Intel Compiler module, using the command:

```
$ module load intel/11.1.046
```

Once the module for Intel C/C++ is added to your shell environment, Intel C/C++ binaries, headers, libraries and help pages will be available to your session.

You can load multiple modules in one line. For example, to load intel compilers and OpenMPI:

```
$ module load intel/11.1.046 openmpi/intel/1.4.3
```

You can view the added modules using the command:

```
$ module list
```

To see specific module information such as what other modules needed to be loaded first to load a specific module you can use the command `module show <module name>`. For example, to show more information on OpenMPI module:

```
$ module show openmpi
```

To unload a module you can use the command `module unload <module name>`. For example, to unload a loaded intel compiler module:

```
$ module unload intel
```

To unload more than one out of many loaded modules you can use `module unload <module name> <module name>`. For example, out of many loaded modules if you want to unload intel and openmpi:

```
$ module unload openmpi intel
```

To unload all the loaded modules, you can do:

```
$ module purge
```

You can use help option to know more information on module command:

```
$ module --help
```

You may refer to the Available Software page if you need some instructions on how to load modules.

### Compiling Serial Code Using Intel Compilers

If you are compiling a serial job, follow these steps:

**1. Load the appropriate Intel Compiler into your environment, C or Fortran, using one of these Module commands.**

Load both Fortran and C/C++ compilers as well as the Intel MKL (Math Kernel Library) together:

```
$ module load intel/11.1.046
```

At this point, verify that the Intel C/C++ module is loaded using the `which` Unix command. The command will return the location of the Intel C
binary file, icc.

```bash
$ which icc
/share/apps/intel/Compiler/11.1/046/bin/intel64/icc
```

In case the module did not load or a different module was loaded by mistake, the `which` command will return an error:

```bash
$ which icc
/usr/bin/which: no icc
```

2. Compile your code with the appropriate Intel compiler.

Once the appropriate module is loaded, the `icc` for C, `icpc` for C++ and `ifort` for Fortran commands will invoke the respective compiler.

To compile using the Intel Fortran Compiler:

```bash
$ ifort -o outfile prog.f90
```

To compile using the C Compiler:

```bash
$ icc -o outfile prog.c
```

To compile using the C++ Compiler:

```bash
$ icpc -o outfile prog.C
```

Using Optimization Flags

A common practice in compiling code is to use the Intel Compiler optimization flags in order to improve performance. For instance: `-O2 -fPIC -align -Zp8 -axP -unroll -xP -ip`. Here's an example of what your compile command might look like when you use these flags. Remember, this is just an example. Your command may not look exactly like this:

```bash
$ ifort -O2 -fPIC -align -Zp8 -axP -unroll -xP -ip -o outfile prog.f90
```

For Cardiac, we recommend using these flags: `-O2 -fPIC -align -Zp8 -xO -unroll -axO -ip`. In this case, your compile command might look something like this:

```bash
$ ifort -O2 -fPIC -align -Zp8 -xO -unroll -axO -ip -o outfile prog.f90
```

You can find information about what the above flags mean by entering the below command to see Fortran Help:

```bash
$ ifort -help
```

Compiling OpenMP Code with Intel Compilers

OpenMP code runs exceptionally well on our large cache CPU nodes on the Union Square cluster (USQ) and on the Nehalem CPUs on the Bowery cluster. Follow the procedure below when compiling OpenMP code using the Intel Compilers.

1. Load the appropriate Intel Compiler into your environment -- Fortran, C or C++ -- using one of the following Module commands.

For the Intel Fortran compiler:

```bash
$ module load intel-fortran/fce/11.1.046
```

For the Intel C compiler:

```bash
$ module load intel-c/cce/11.1.046
```

For the Intel C++ compiler (same as for C):

```bash
$ module load intel-c/cce/11.1.046
```

Or alternatively, load both the Fortran and C/C++ compilers as well as the Intel MKL (Math Kernel Library) together:

```bash
$ module load intel/11.1.046
```

2. Compile your OpenMP Code with the Intel Compiler.

For the Intel Fortran compiler:

```bash
$ ifort -openmp -o outifile prog.f90
```

For the Intel C compiler:

```bash
$ icc -openmp -o outfile prog.c
```

For the Intel C++ Compiler:

```bash
$ icpc -openmp -o outfile prog.C
```
For the Intel C++ compiler:

```
$ icpc -openmp -o outfile prog.C
```

For OpenMP code, it's important to set CPU, thread, and process counts. Please see the Running your Code section in this Wiki.

**Useful Information**
On Cardiac, if you are using gcc to compile OpenMP programs higher versions of gcc (4.5.3 & 4.6.2 > 4.1.2) are recommended. Which can be loaded using `modules`.

### Compiling MPI Parallel Code

NYU HPC provides two standard MPI compilers for parallel compilation: OpenMPI from the Open MPI Forum and MVAPICH from Network-Based Computing Laboratory at Ohio State University. (Intel MPI is available in test mode.)

To compile MPI parallel code:

1. Load the appropriate compiler for your source code from Environment Modules.
2. Load the appropriate MPI wrapper from Environment Modules. Both the compiler and wrapper need to be loaded in order for the binaries, headers and libraries to be available to your source code.

To load and compile MPI code using the MVAPICH Fortran wrapper:

```
$ module load intel-fortran/fce/11.1.046
$ module load mvapich/intel/1.1.0
$ mpif90 -O2 prog.f90
```

or

For FORTRAN 77

```
$ mpif77 -O2 prog.f
```

To load and compile MPI code using the OpenMPI Fortran wrapper:

```
$ module load intel-fortran/fce/11.1.046
$ module load openmpi/intel/1.3.3
$ mpif90 -O2 prog.f90
```

To load and compile MPI code using the MVAPICH C wrapper:

```
$ module load intel-c/cce/11.1.046
$ module load mvapich/intel/1.1.0
$ mpicc -O2 prog.c
```

To load and compile MPI code using the MVAPICH C++ wrapper:

```
$ module load intel-c/cce/11.1.046
$ module load mvapich/intel/1.1.0
$ mpiCC -O2 prog.C
```

To load and compile MPI code using the OpenMPI C wrapper:

```
$ module load intel-c/cce/11.1.046
$ module load openmpi/intel/1.3.3
$ mpicc -O2 prog.c
```

To load and compile MPI code using the OpenMPI C++ wrapper:

```
$ module load intel-c/cce/11.1.046
$ module load openmpi/intel/1.3.3
$ mpiCC -O2 prog.C
```
Troubleshooting Compiling Sessions

The key to success in compiling code is in carefully specifying the path to the various headers and libraries required by your source code. One of the most frequently occurring mistakes in compiling is to enter the wrong path, library or header, with the result that the source files, the Makefile/make process or the resulting binary cannot find the correct information it needs.

It is important to enter the entire environment path when you compile source or execute binaries by using manual entries in the shell environments i.e., bash or tcsh, or by using preset values offered by Environment Modules.

Below are two typical errors frequently made in compiling and running code:

**Common Compile Time Error**

Compile time errors, such as icc, ifort, mpicc, mpi90 "not found" type errors indicate a missing path to the compiler binaries or libraries.

**Sample Error**

For example, trying to compile test.c will result in such an error because the path to the C/C++ compiler has not been loaded. Verify the presence of the Intel C compiler using the `which` Unix command:

```
$ icc test.c
bash: icc: command not found
$ which icc
/usr/bin/which: no icc
```

**Sample Resolution**

This error can be resolved by invoking the appropriate `module load` command which specifies the path to the desired compiler, in this case the C/C++ compiler. To do so, use the command below:

```
$ module load intel-c/cce/11.1.046
$ which icc
/share/apps/intel/Compiler/11.1/046/bin/intel64/icc
```

**Common Runtime Error**

Errors can also occur after your source code has been compiled and a test run is initiated. Library "not found" errors are the most common and are usually caused by the environment path to library and include files not being defined.

**Note**

Please keep in mind that the following error and resolutions are simply examples. For different data, jobs and compilers, you will, of course, get different results.

**Sample Error**

We managed to compile our source code but when we try running our binary "test" on the cluster, the program crashes. A common error is a missing library dependency.

Use the `ldd` Unix command to verify library dependencies. `ldd` prints out shared library dependencies, and in our case it is easy to see which libraries are mapped, and which are NOT mapped correctly:

```
$ ldd test
libmpi.so.0 => /usr/lib64/lam/libmpi.so.0 (0x000002ae836a92000)
libopen-rte.so.0 => not found
libopen-pal.so.0 => not found
libdl.so.2 => /lib64/libdl.so.2 (0x00000037eba00000)
libnsl.so.1 => /lib64/libnsl.so.1 (0x00000037ede00000)
libutil.so.1 => /lib64/libutil.so.1 (0x00000037eca00000)
libm.so.6 => /lib64/libm.so.6 (0x00000037eb600000)
libpthread.so.0 => /lib64/libpthread.so.0 (0x000000037eb00000)
```

Use the `ldd` Unix command to verify library dependencies. `ldd` prints out shared library dependencies, and in our case it is easy to see which libraries are mapped, and which are NOT mapped correctly:

```
$ ldd test
libmpi.so.0 => /usr/lib64/lam/libmpi.so.0 (0x000002ae836a92000)
libopen-rte.so.0 => not found
libopen-pal.so.0 => not found
libdl.so.2 => /lib64/libdl.so.2 (0x00000037eba00000)
libnsl.so.1 => /lib64/libnsl.so.1 (0x00000037ede00000)
libutil.so.1 => /lib64/libutil.so.1 (0x00000037eca00000)
libm.so.6 => /lib64/libm.so.6 (0x00000037eb600000)
libpthread.so.0 => /lib64/libpthread.so.0 (0x000000037eb00000)
/lib64/ld-linux-x86-64.so.2 (0x000000037eae0000)
```
Sample Resolution

Loading the correct Module, in our test case the Intel version of the OpenMPI module (which we used originally to compile our code with), will add the required library path to the environment:

```
$ module load openmpi/intel/1.2.8
$ ldd test
libmpi.so.0 => /usr/mpi/intel/openmpi-1.2.8/lib64/libmpi.so.0 (0x00002b7afbf39000)
libopen-rte.so.0 => /usr/mpi/intel/openmpi-1.2.8/lib64/libopen-rte.so.0
(0x00002b7afccf6000)
libopen-pal.so.0 => /usr/mpi/intel/openmpi-1.2.8/lib64/libopen-pal.so.0
(0x00002b7afca4f000)
    libdl.so.2 => /lib64/libdl.so.2 (0x00000037eba00000)
    libnsl.so.1 => /lib64/libnsl.so.1 (0x00000037ede00000)
libutil.so.1 => /lib64/libutil.so.1 (0x00000037eca00000)
    libm.so.6 => /lib64/libm.so.6 (0x00000037eb600000)
    libpthread.so.0 => /lib64/libpthread.so.0 (0x00000037ebe00000)
    libc.so.6 => /lib64/libc.so.6 (0x00000037eb200000)
    libimf.so => /share/apps/intel/Compiler/11.1/046/lib/intel64/libimf.so (0x00002b7afccf2000)
    libsvml.so => /share/apps/intel/Compiler/11.1/046/lib/intel64/libsvml.so
(0x00000037eae00000)
libthread.so.0 => /lib64/libthread.so.0 (0x00000037ebe00000)
lib.so.6 => /lib64/lib.so.6 (0x00000037eb800000)
    libintel.c.so.5 => /share/apps/intel/Compiler/11.1/046/lib/intel64/libintel.c.so.5
(0x00002b7afcc5b000)
    libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x00000037ed200000)
    /lib64/ld-linux-x86-64.so.2 (0x00000037eae00000)
```

For parallel jobs in general, it is important that both the compiler and the MPI wrapper libraries are loaded.

Debugging

Overview

Debugging and optimization of code help not only the individual user's code but also results in systems to run more efficiently. A 20% increase in a code's efficiency can greatly reduce time required for a batch job and can result in significantly more research to be accomplished over time.

Debuggers

Intel Debugger for Linux (IDB)
The Intel Debugger for Linux is a component of the Intel compiler products. Some introductions can be found at,

GNU GDB
http://www.gnu.org/software/gdb/documentation/

Totalview
NYU HPC maintains a site-license of the Totalview debugger. The license allows students and faculty to use the debugger on our clusters or on individual laptops and desktops.

If you have any questions about compilers, please contact us at hpc@nyu.edu.

Note
The `ldd` command is used to identify the problem, not to fix it.
PBS Script Generator
An interactive tool that generates PBS script based on user's input. Check this page for more details.

Front-Line HPC Consulting
HPC consultations are available once a week, Monday 1-3 PM. Appointments are required. Please make an appointment at hpc@nyu.edu.