Building and Compiling Software on HPC

Charles Peterson

👋 Let's Unravel the Mysteries of Compiling

Welcome to our workshop on building and compiling software on Hoffman2.

This knowledge can translate to building software on most other HPC resources.

This session is designed to teach you to install and manage most scientific software on Hoffman2, without the need for constant assistance from system admins.

📖 Access the Workshop Files

This presentation and accompanying materials are available on 🔗 UCLA OARC GitHub Repository

You can view the slides in:

- 📄 PDF format - CompilingOnHPC.pdf
- 🌐 HTML format: Workshop Slides

Each file provides detailed instructions and examples on the various topics covered in this workshop.

> **Note**
>
> - This presentation was built using Quarto and RStudio.
> - Quarto markdown file: CompilingOnHPC.qmd

🔩 Basic Concepts: Installing Software

Let’s understand the basic concepts involved in installing software from source code:

- 📖 Compilation: Converting human-readable source code into machine code.
- 🔗 Library Linking: Connecting pre-written library functions to parts of your software.
- 📁 Executable File: The output after compilation and linking, which can be run on your system.
- ⚙️ Autotools: A suite of tools designed to make source code portable to Unix-like systems, often used to create configure scripts.
- 🛠️ Build Automation Tools (Make, CMake): These tools automate the building process, keeping track of dependencies to save time.

 Basic Concepts: Building Software

In most cases, scientific software comes with all these tools and files already set up.

Installing software on most HPC systems generally involves:

- Downloading source code
- Configuring the build environment
  - Getting compilers and shell session ready
- Compiling the code
- Installing the software
  - Moving compiled code to be ready to run

We will explore R/python/anaconda in another workshop.

💡 HPC Compiling Tips

- Hoffman2, like many other HPC resources, are **SHARED** among all users.
  - All Hoffman2 users are login to the same Hoffman2 server.
- This means you can't change the Operating System, modify system level directories, or gain admin privileges.
  - 🚫 No access to sudo
  - 🚫 Cannot install via apt/yum
- Generally, you only have control over your user (or group) owned space
  - You are free to install any software you want, but you'll have to install them in your user-owned directories
  - $HOME, $SCRATCH, any project directories

🔧 HPC compiling Environment

Setting up the Environment

Hoffman2 provides **modulefiles** for different versions of GCC and Intel Compilers.

These **modules** establish your Hoffman2 environment to use compilers, libraries, and other software pre-installed by Hoffman2 admins.

Here's how you list available modules:

```
modules_lookup -h
modules_lookup -a
modules_loodup -m gcc
```

- Use `module load <module_name>/<version>` to load the required compilers into your environment.

```
module load gcc/10.2.0
module load intel/2022.1.1
```

⚙️ GCC (GNU Compiler Collection)

💡 GCC is a free, open-source compiler that supports multiple languages, including C, C++, and Fortran.

- C - `gcc`
- C++ - `g++`
- Fortran - `gfortran`

You can verify its version

```
gcc --version
g++ -v
gfortran -v
```

Setting up GCC

The default GCC version on Hoffman2 (and other CentOS 7 systems) is 4.8.5.

This version may be too outdated for modern software, but Hoffman2 has other versions installed that can be accessed by loading modules

Here's how to list all available GCC versions on Hoffman2:

```
modules_lookup -m gcc
```

Here's how to use a specific GCC version:

```
module load gcc/10.2.0
```

💻 Intel Compilers

💡 Intel's sophisticated compilers for C, C++, and Fortran are optimized to leverage specific features of Intel processors, often resulting in faster and more efficient code.

OneAPI, Intel's comprehensive suite, includes these compilers along with MPI, GPU, Math, and other libraries.

- C - `icc`
  - `icx` for newer Intel versions
- C++ - `icpc`
  - `icpx` for newer Intel versions
- Fortran - `ifort`

Setting up Intel

List all available Intel versions on Hoffman2

```
1 modules_lookup -m intel
```

Use a certain GCC version

```
1 module load intel/2022.1.1
```

> **Note**
>
> Intel compilers may relay on GCC libraries. You may need to load a more recent GCC to use Intel compilers
>
> ```
> 1 module load gcc/10.2.0
> 2 module load inel/2022.1.1
> ```

Choosing a Compiler

The choice of compiler depends on the specific requirements of your software and your project's needs.

For general use, GCC is an excellent choice due to its wide support and availability.

For Intel-specific optimizations, or if your software specifically requires it, use Intel Compilers.

Other Compilers:

- CUDA: For compiling GPU based code
  - `modules_lookup -m cuda`
- HPC SDK (Software Development Kit): Toolbox from Nvidia that contains
  - PGI (Portland Group, Inc) PGI compilers
  - MPI, Math libraries, and CUDA
  - `modules_lookup -m hpcsdk`

🔗 Introduction to MPI

💡 MPI (Message Passing Interface) is a standardized library specification for message-passing between different processes, typically in parallel computing environments.

It allows programs to run on multiple processors or compute nodes at once, communicating as needed, which can greatly speed up computation times.

If you code is setup to run in parallel with MPI, you can compile the software with MPI compilers and libraries

Intel MPI

Intel MPI is Intel's proprietary implementation of the MPI specification.

It is optimized to leverage the capabilities of Intel processors and networking technologies, offering high performance and scalability.

If you have the `intel` module loaded, it comes with Intel MPI.

Intel MPI commands with Intel compilers

```
mpiifort # Fortran
mpiicc   # C
mpiicpc  # C++
```

Intel MPI commands with GCC compilers

```
mpif90/mpif77 # Fortran
mpicc         # C
mpicxx        # C++
```

Open MPI

Open MPI is a popular, open-source MPI implementation that is widely used in the HPC community. It supports a wide range of platforms and networking technologies. Use mpicc, mpic++, or mpifort to compile your programs with Open MPI.

List all available OpenMPI on Hoffman2

```
modules_lookup -m openmpi
```

OpenMPI compilers:

```
mpif90/mpif77.  # Fortran
mpicc           # C
mpicxx          # C++
```

Selecting the Right MPI Implementation

The decision between Intel MPI and Open MPI is primarily influenced by the specific requirements of your application, including the hardware architecture, performance considerations, and code compatibility.

Intel MPI tends to deliver enhanced performance on Intel hardware, while Open MPI offers wide-ranging compatibility and is open-source, providing flexibility for customization.

Always refer to your software's documentation to understand their recommendations and what they have tested on. While some codes might operate with any MPI implementation or version, others may have specific requirements.

📈 Leveraging Mathematical Libraries

Mathematical libraries, providing precompiled and performance-optimized mathematical routines, are key accelerators of scientific computations. Many scientific software packages rely on them for efficient numerical computation.

🔢 Common Mathematical Libraries

- In HPC environments, the following mathematical libraries are often utilized:
  - ATLAS (Automatically Tuned Linear Algebra Software): An adaptive, high-performing library for linear algebra, optimizing itself for the specific hardware it's running on.
  - BLAS (Basic Linear Algebra Subprograms): Provides basic vector and matrix operation routines, forming the foundation for other high-level libraries.
  - LAPACK (Linear Algebra Package): Extends BLAS, offering routines to solve simultaneous linear equations, least-squares solutions, eigenvalue problems, and singular value problems.
  - Intel Math Kernel Library (MKL): Intel's highly optimized math processing library, encompassing linear algebra routines, fast Fourier transforms, vector math, and more.

Be sure to check your software's documentation for any specific library requirements.

🚀 Intel Math Kernel Library (MKL)

Intel MKL is a leading math library for high-performance computing. It provides optimized versions of BLAS, LAPACK, and other libraries.

The `intel` module on Hoffman2 already includes these libraries.

Most software will automatically locate and link these Math libraries during the build process.

🌐 Downloading Software Code

The source code of most research software is typically available on the internet. Refer to your software's documentation to find out where you can download the code.

💻 Transferring Code from Your Local Machine

If the code is on your local computer, you can transfer it to Hoffman2 using the scp command:

```
scp mycode.tar.gz joebruin@hoffman2.idre.ucla.edu:~/
```

⬇️ Downloading Code Directly to Hoffman2

Alternatively, you can download the code directly to Hoffman2 from the internet using wget or curl:

```
wget https://URL-to-code.com/mycode.tar.gz
```

🌐 Setting up your Software

📦 Unpacking Compressed Files

If your code is compressed in a .tar.gz or .zip file, you can uncompress it using the tar or unzip command:

```
tar -xzvf mycode.tar.gz
unzip mycode.zip
```

📦 Cloning from GitHub

If your code is hosted on a git repository (like GitHub), you can clone the repository:

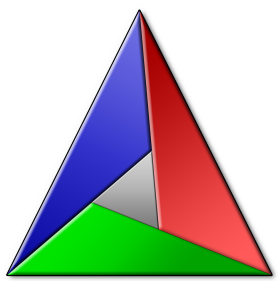
```
git clone https://github.com/PATH-TO-Code
```

We've now learned different methods to get our software code onto Hoffman2, whether it's transferring from our local machine, downloading directly, or cloning from a git repository.

In the next steps, we'll look at how to build and compile this code.

Make/CMake

GNU Make



CMake

Introduction to Make

💡 `make` is a powerful tool for build automation, leveraging a `Makefile` to guide program compilation and linking processes.

The `Makefile` is a file that outlines rules defining dependencies between files and the commands to create or update them.

The common usage pattern is `make <target>`, where specifies the build objective.

- Pre-written Makefiles are included with most software, reducing the need to create your own.
  - But it's crucial to understand the process in case you need to modify it.

Unraveling ./configure Scripts

Often, when you download source code, you'll encounter a `configure` script in the top level directory of your software.

Part of the GNU build system or autotools, the configure script ensures the build environment is properly set up, dependencies are checked, and a suitable Makefile is created for your system.

This configure script is usually pre-included with software packages, though it can be generated using the `autoconf` tool.

⚙️ Making Use of ./configure

- Running ./configure typically initiates the following actions:
  - 📁 Verifies that your system possesses necessary prerequisites for software build.
  - 🔍 Discovers system-dependent features and libraries essential for compilation.
  - 📄 Produces a Makefile customized for your system and setup.

💡 Running ./configure

All you need to do is the run the configure script to start your install

Execute the configure script to initiate the installation process. You can customize the build by passing options to ./configure.

```
FC=gfortran CC=gcc CXX=g++ ./configure --prefix=$HOME/apps/myapp
```

- The command above runs ./configure with these configurations:
  - Fortran - `gfortran`
  - C - `gcc`
  - C++ - `g++`
  - Installation directory - `$HOME/apps/myapp`

Use ./configure –help to see a list of options.

- In this workshop, the –prefix option in ./configure is of **key importance** of installing software on HPC.
  - It indicates the location for the software installation.
  - If unspecified, it defaults to /usr/local, which is inaccessible to a standard HPC user.

🚀 After Running ./configure

Upon the successful completion of `./configure`:

- 🛠️ Generally, you'd execute `make` to initiate software compilation.
  - This step forms the core of the compilation process.
  - Depending on the software size, this might take minutes or even hours.
- 📥 Subsequently, you execute `make install` to relocate the software to the final directory.
  - This involves moving all necessary files for running your code to the directory specified by your `--prefix`.

Installing Quantum ESPRESSO

Quantum ESPRESSO is an open-source software package for electronic structure calculations using density functional theory. It simulates and analyzes atomic and molecular behavior at the quantum level, contributing to our understanding of various physical and chemical phenomena.

Below is a step-by-step guide to install Quantum ESPRESSO:

- Set up your environment: Load necessary modules

```
module load intel/2022.1.1
module load gcc/10.2.0
```

- Download the source code: Clone the Quantum ESPRESSO repository and navigate to the version you want to install

```
git clone https://github.com/QEF/q-e.git
cd q-e/
git checkout qe-6.7.0
```

- Configure the build: Initiate the installation process using the ./configure command. Specify the compilers and installation directory (–prefix)

```
./configure MPIF90=mpiifort F90=ifort F77=ifort CC=icc   CC=mpiicc  --prefix=$HOME/apps/espress
```

- Compile and install the software: Finally, compile and install Quantum ESPRESSO using make and make install commands.

```
make pw
make install
```

⚒ Introduction to CMake

💡 You may often find the tool `cmake` is for build automation.

It uses CMakeLists.txt files to define how your program should be built.

Unlike make, cmake is platform independent and can generate native build scripts (e.g., makefiles on Unix and projects/workspaces in MSVC).

Normally, you will first need to to create a build directory for the compilation, then run `cmake <path to source>`, and finally `make`.

🔬 GROMACS Example

GROMACS (GROningen MAchine for Chemical Simulations) is a comprehensive package to execute molecular dynamics simulations, primarily of biochemical molecules like proteins and lipids. It is highly optimized for a range of hardware platforms and provides a plethora of calculation types, integration algorithms, and analysis tools.

We'll utilize GROMACS to guide you through this build process.

💾 Download the code:

```
wget https://ftp.gromacs.org/gromacs/gromacs-2023.2.tar.gz
tar -vxf gromacs-2023.2.tar.gz
cd gromacs-2023.2
```

📁 Create a build directory:

```
mkdir -pv build
cd build
```

At this point, we're currently located in the directory `/../../gromacs-2023.2/build`. The main source is still accessible at `/../../gromacs-`

`2023.2` or simply `..`

🔧 Configure the build:

```
module load cmake
module load gcc/10.2.0
FC=gfortran CC=gcc CXX=g++ cmake .. -DGMX_BUILD_OWN_FFTW=ON -DREGRESSIONTEST_DOWNLOAD=ON -DCMAK
```

Here, `-DGMX_BUILD_OWN_FFTW=ON` and `-DREGRESSIONTEST_DOWNLOAD=ON` are GROMACS-specific options.

The `-DCMAKE_INSTALL_PREFIX` directive instructs `cmake` on the destination directory for the final compiled code.

After you run `cmake`, you will run `make` to compile the code, then `make install` to install to the final location.

```
make
make install
```

QUILL: Handling Software Dependencies

Certain software applications may require additional libraries or tool packages to function correctly. In such cases, you need to compile these dependency packages or libraries before you compile your main software.

On HPC environments like Hoffman2, many pre-compiled libraries can be conveniently loaded using the module load command.

Consider the following example of QUILL, a computational chemistry SCF code:

QUILL was an early attempt I had at learning Hartree-Fock in Graducate school, which you can check out at https://github.com/charliecpeterson/QUILL

- To install, we need
  - Python with the PySCF package
  - Eigen3, a Linear Algebra library
  - At least GCC version 7

Installing QUILL

- Set up your environment: Load necessary modules

```
module load cmake
module load python/3.9.6
module load gcc/10.2.0
module load eigen/3.3.9
```

- Download the source code: Clone the QUILL repository

```
git clone https://github.com/charliecpeterson/QUILL
cd QUILL
```

- Prepare for the build: Create a build directory and navigate into it

```
mkdir build
cd build
```

- Configure the build: Use cmake command, specifying the installation directory

```
cmake .. -DCMAKE_INSTALL_PREFIX=$HOME/apps/QUILL/1.0
```

- Compile and install the software: Finally, compile and install QUILL using make and make install commands.

✅ Post-compilation Steps

Once your software is compiled, ensure that you load the EXACT SAME modules during runtime as you did during the compilation.

Update your $PATH and $LD_LIBRARY_PATH variables to incorporate your newly compiled software:

- $PATH ensures Hoffman2 can locate the executables:

```
1  export PATH=$HOME/apps/myapps/bin:$PATH
```

$LD_LIBRARY_PATH ensures correct linking of the libraries:

```
1  export $LD_LIBRARY_PATH=$HOME/apps/myapps/lib:$LD_LIBRARY_PATH
```

These export commands update $PATH and $LD_LIBRARY_PATH to add the directories of your new software.

Tips

Warning

Exercise caution when adding export commands to your `$HOME/.bashrc` file.

Modifications to this file can inadvertently create conflicts and cause errors during future software installations. As a general rule, it's best to avoid altering `$HOME/.bashrc` unless necessary.

Note

Creating dedicated bash scripts to load modules and update environment variables for each software can simplify your workflow and minimize potential conflicts. Simply source the appropriate script when needed.

For instance, consider a `start-nwchem-7.0.2.sh` file:

```
module load gcc/10.2.0
module load intel/2022.1.1
export PATH=$HOME/apps/nwchem/7.0.2/bin:$PATH
export NWCHEM_BASIS_LIBRARY=$HOME/apps/nwchem/7.0.2/data
```

To use, run:

```
source start-nwchem-7.0.2.sh
```

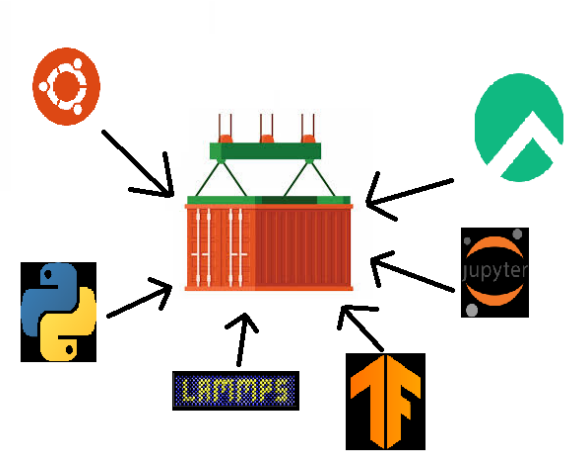
You can create multiple of these scripts to different software and versions. This will help in case there are conflicts when installing different software

Containers

📦 Embracing Apptainer

💡 Apptainer, previously known as Singularity, is a container platform tailored for High-Performance Computing (HPC) environments.

- It allows you to create portable, self-contained operating systems with precompiled software.
  - You can effortlessly transfer this container to any system with
  - Apptainer installed, eliminating the need for repeated compilations.
  - Apptainer containers can be shared, enabling your collaborators to use the exact same software builds you are using, thus fostering consistent and reproducible research workflows.



Check out my previous workshops on USING and BUILDING container for HPC.

🏆 Key Takeaways

Here's the general workflow for installing most software on HPC resources:

1. Download the code: Use commands like `git`, `scp`, `wget` to transfer your code to your HPC environment.

2 Load the necessary environment: Setup your shell environment with required modules using the `module load` command.

3. Configure the build: Use `./configure` to initialize your installation process. This script sets up the build environment correctly, checks for dependencies, and creates a Makefile suitable for your system.
4. Utilize CMake (if supported): Some software packages support `cmake` which is another powerful build system.
5. Compile the code: Use the `make` command to compile your code. This can take anywhere from a few minutes to several hours, depending on the size of your software.
6. Install the software: Use `make install` to transfer your compiled software to a specified directory. Remember to indicate this directory during the configuration stage.

Remember to update $PATH and $LD_LIBRARY_PATH when you are ready to use your code

Note

👏 Thanks for Joining! ❤️

Questions? Comments?

- cpeterson@oarc.ucla.edu
- Look at for more Hoffman2 workshops at https://idre.ucla.edu/calendar