PPPL PRINCETON PLASMA PHYSICS LABORATORY

Deploying Community Codes

Part 1

OSCER Virtual Residency Workshop
June 28, 2023

Prentice Bisbal
Senior HPC Engineer
Computational Sciences Department
Princeton Plasma Physics Laboratory

Outline

1. What are community codes?
2. Why do we need to compile code for HPC?
3. Steps of compilation
4. Where to find and put files (The Filesystem Hierarchy Standard)
5. How to tell the compiler or final program where to find files (environment variables)
6. How to build community codes (configure and CMake)

What are “Community Codes”?

“Community Codes” are programs of the community, by the community, for the community:

- Written for a specific research community
- Often written by that research community
- Open-source for community contributions

Why do we need to compile codes

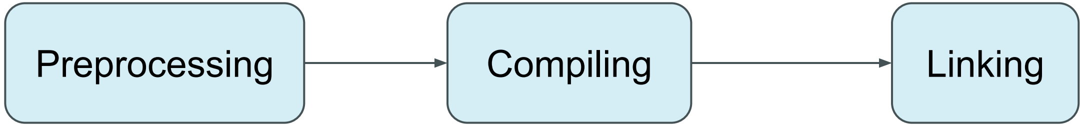
- Historical requirement:
  - Different Unixes → different processors
  - Different Unixes → different compilers
  - Different Unixes → different library names
  - Different Unixes → different library implementations
  - Different Unixes → different paths to files
- Today:
  - Different distros → different compiler versions
  - Different distros → different paths to files
  - Different processors
    - Arm variants
    - x86 _64 w/ different levels of AVX Support
    - POWER processors
- Performance
  - Make sure code is compiled with correct optimizations for your processors
  - Precompiled binaries target “lowest common denominator” processor
  - Compare performance of code optimized with different compilers
  - Compare performance of different implementations of the same library

Combinatorial builds

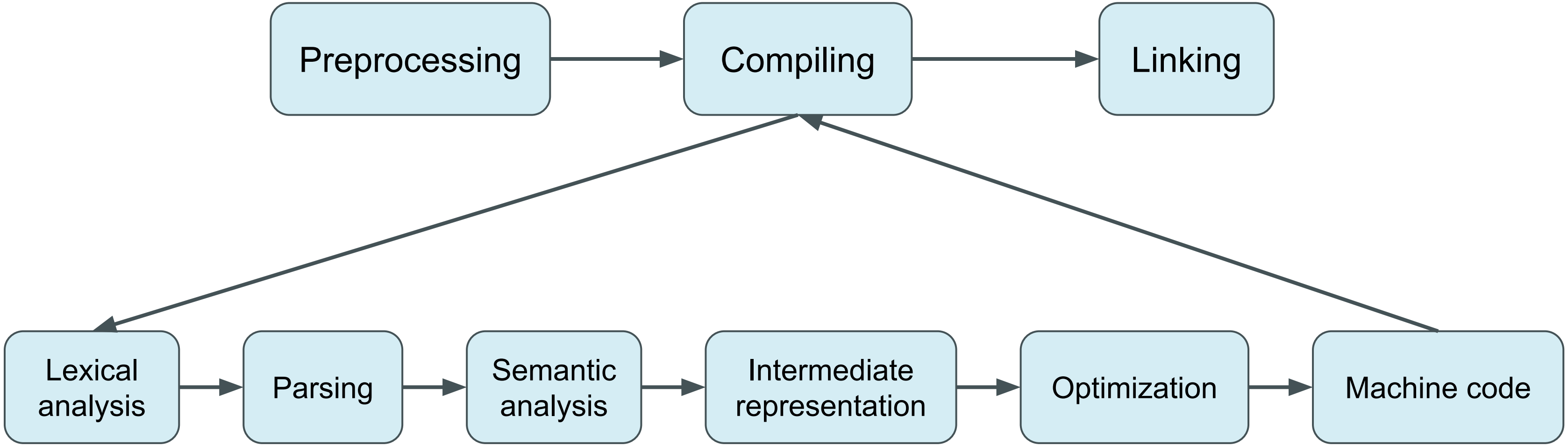
Building multiple instances of an application using different combinations of compilers and/or the underlying libraries

- Find the libraries or compilers that provide the best performance for an application
- Ensure the portability/interoperability/validity of the code your producing
- Leads to many more builds than non-combinatoric builds
- Check reproducibility of results between different versions (2.0 vs. 1.0)
- Leads to A LOT of repeated effort

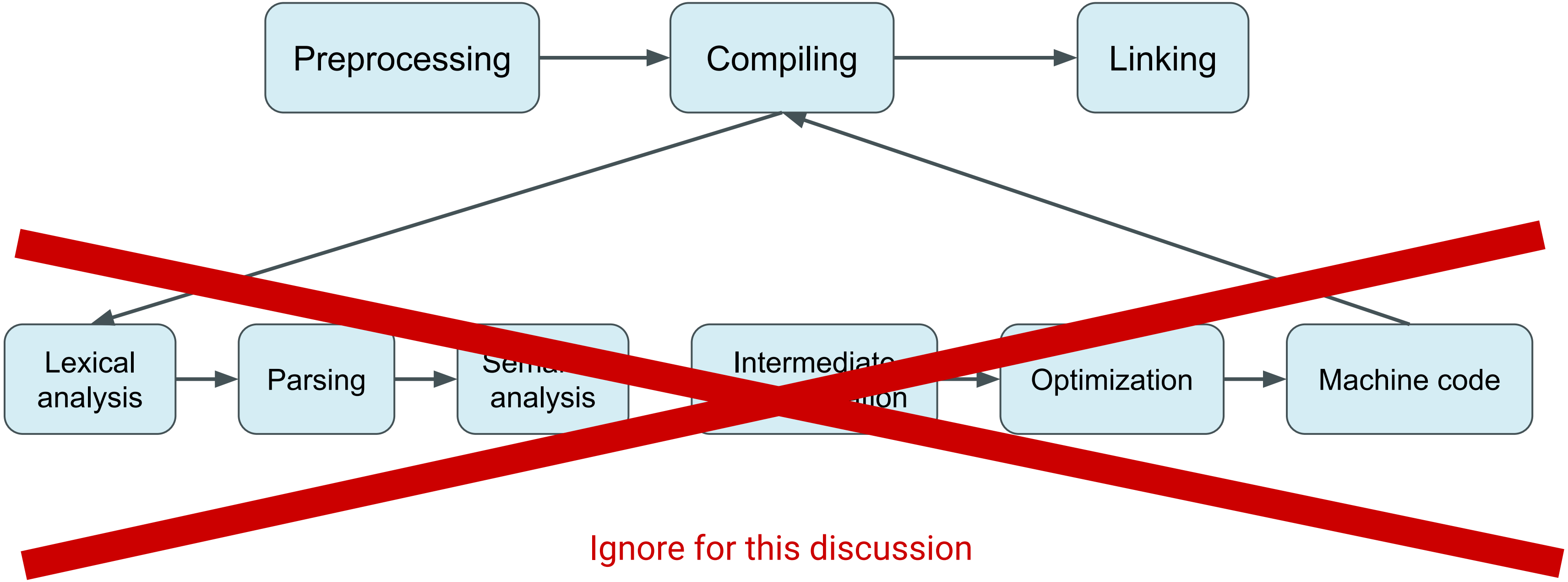
Steps of Compilation



Steps of Compilation



Steps of Compilation



Ignore for this discussion

Where errors are most likely to occur

- Preprocessing
  - Needed files are not installed
  - Preprocessor can’t find files
- Linking
  - Needed libraries are not installed
  - Compile-time linker can’t find libraries needed for linking
- Fixes
  - Install the missing packages
  - Specify the paths to the header files and libraries as part of the build process

Where errors are most likely to occur

- Preprocessing
  - Needed files are not installed
  - Preprocessor can’t find files
- Linking
  - Needed libraries are not installed
  - Compile-time linker can’t find libraries needed for linking
- Fixes
  - Install the missing packages
  - Specify the paths to the header files and libraries as part of the build process

Compile-time linking and run-time linking are two separate things!

Where errors are most likely to occur

- Preprocessing
  - Needed files are not installed
  - Preprocessor can’t find files
- Linking
  - Needed libraries are not installed
  - Compile-time linker can’t find libraries needed for linking
- Fixes
  - Install the missing packages
  - Specify the paths to the header files and libraries as part of the build process

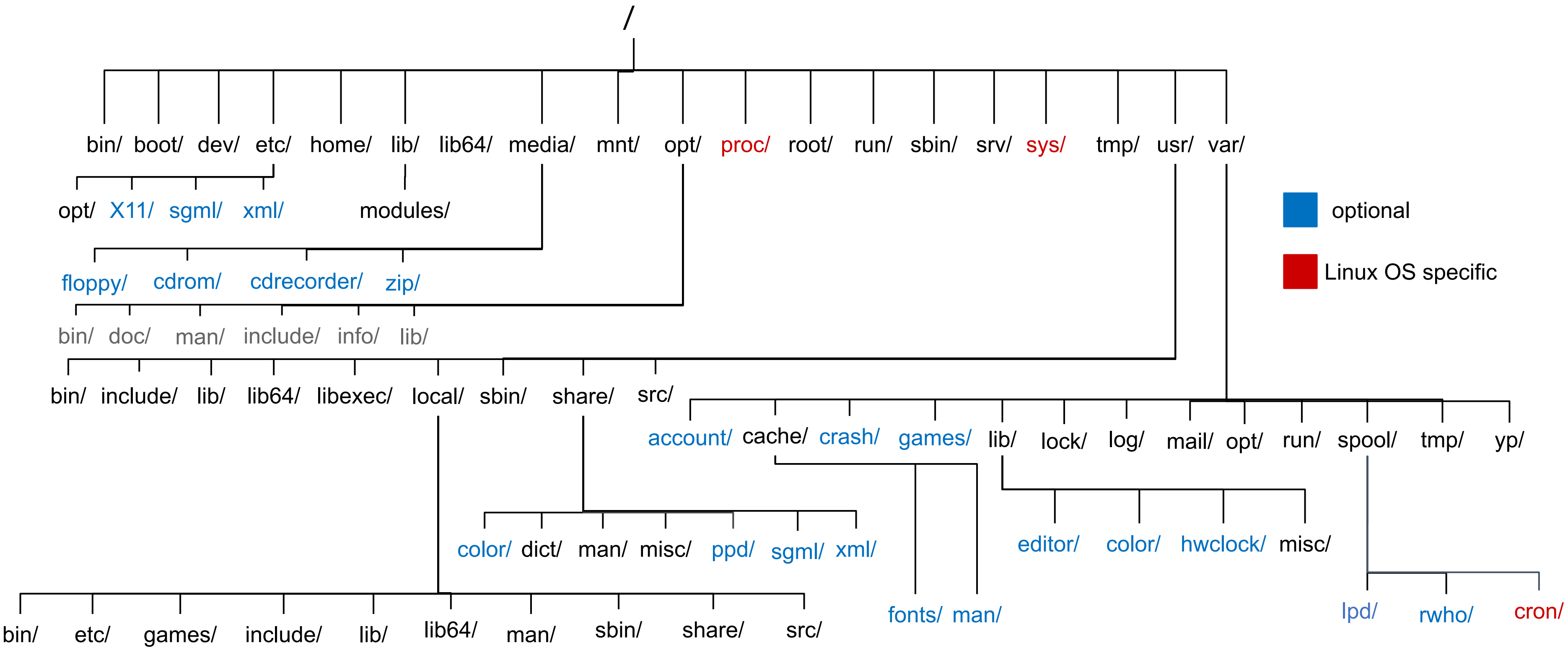
Compile-time linking and run-time linking are two separate things!

Errors rarely occur during the compilation phase. When they do it’s because of a bug in the compiler, or the code is written in a version of the language not supported by the compiler (Fortran 66 or Fortran 2023, or compiler-specific extensions, for example)

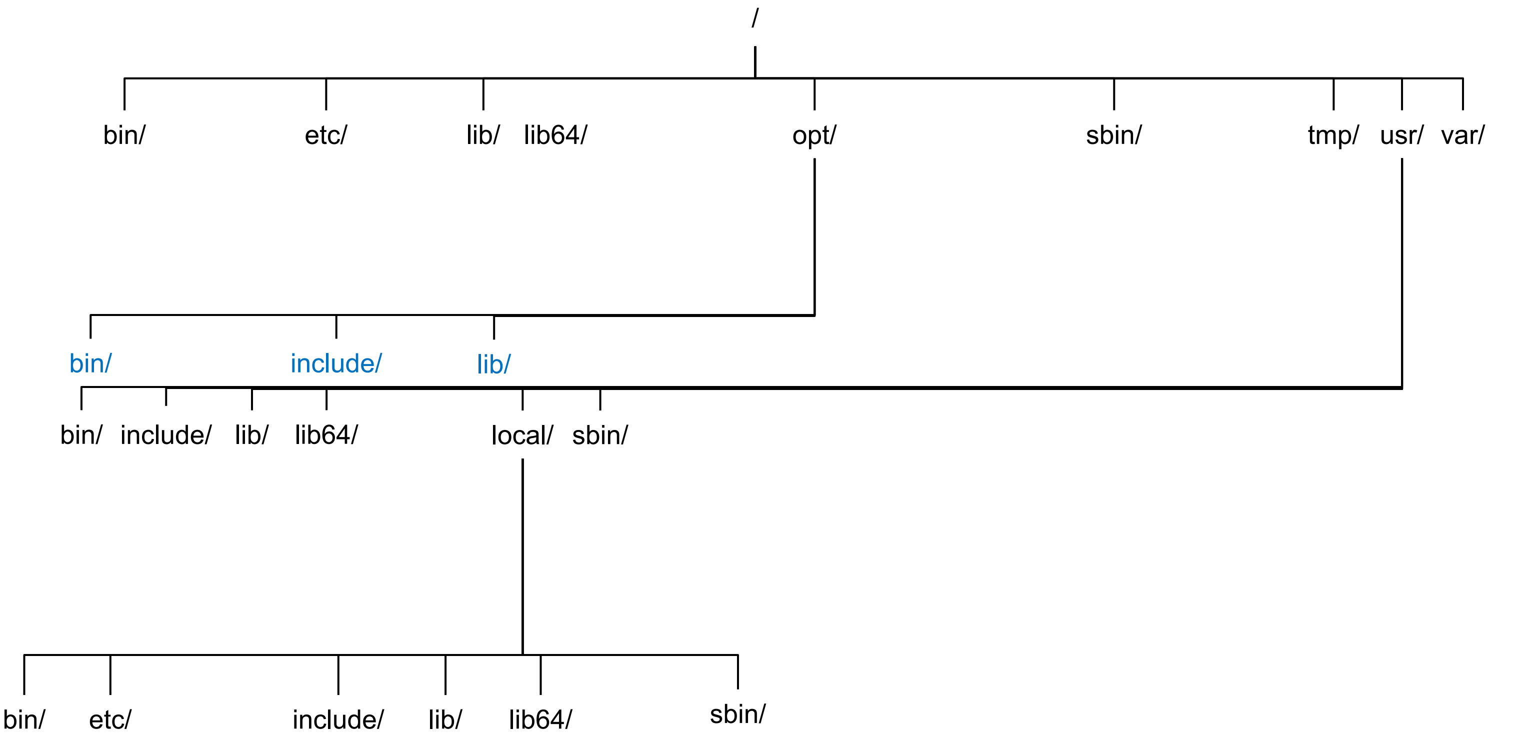
The Filesystem Hierarchy Standard (FHS)

- The Filesystem Hierarchy Standard (FHS) is an industry standard most Linux distros adhere to.
- https://refspecs.linuxfoundation.org/fhs.shtml
- "Local placement of local files is a local issue, so FHS does not attempt to usurp system administrators."
    - For example, all locally install software at PPPL is installed in /usr/pppl
    - Some sites install to /usr/local
    - Some commercial software prefers to install in /opt

The Filesystem Hierarchy Standard (FHS)


/
bin/ boot/ dev/ etc/ home/ lib/ lib64/ media/ mnt/ opt/ proc/ root/ run/ sbin/ srv/ sys/ tmp/ usr/ var/
opt/ X11/ sgml/ xml/
modules/
floppy/ cdrom/ cdrecorder/ zip/
bin/ doc/ man/ include/ info/ lib/
bin/ include/ lib/ lib64/ libexec/ local/ sbin/ share/ src/
account/ cache/ crash/ games/ lib/ lock/ log/ mail/ opt/ run/ spool/ tmp/ yp/
color/ dict/ man/ misc/ ppd/ sgml/ xml/
editor/ color/ hwclock/ misc/
bin/ etc/ games/ include/ lib/ lib64/ man/ sbin/ share/ src/
fonts/ man/
lpd/ rwho/ cron/
optional
Linux OS specific

The Filesystem Hierarchy Standard - Simplified



/

bin/ etc/ lib/ lib64/ opt/ sbin/ tmp/ usr/ var/

bin/ include/ lib/

bin/ include/ lib/ lib64/ local/ sbin/

bin/ etc/ include/ lib/ lib64/ sbin/

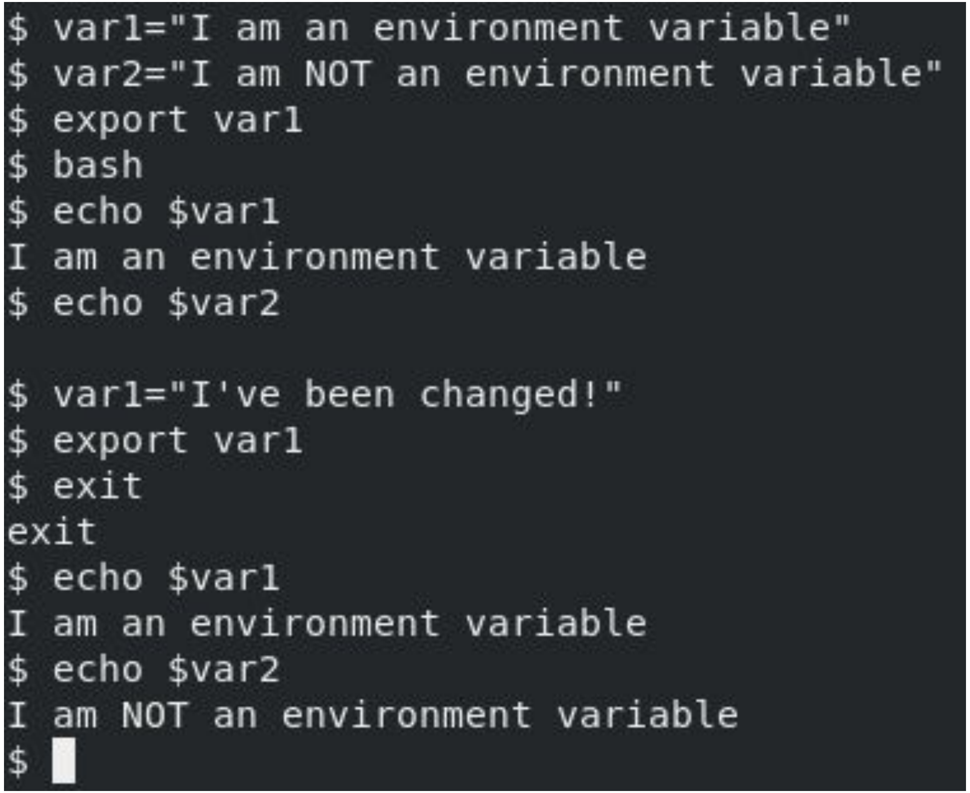
🟦 optional

🟥 Linux OS specific

Environment Variables

- Variables whose values are available to children of the process where they are defined
- Must use a shell-specific command to define an environment variable
  - Bash → export
  - csh → setenv
- Customary to use ALL CAPS for variable names, but not required
- Using ALL CAPS for variable name doesn’t make it an environment variable
- Programs can check the values of environment variables determine how they should act:

Environment Variable example



```
$ var1="I am an environment variable"
$ var2="I am NOT an environment variable"
$ export var1
$ bash
$ echo $var1
I am an environment variable
$ echo $var2

$ var1="I've been changed!"
$ export var1
$ exit
exit
$ echo $var1
I am an environment variable
$ echo $var2
I am NOT an environment variable
$ 
```

Common Environment Variables

- PATH
- HOME
- SHELL
- LIBRARY_PATH (GCC)
- LD_LIBRARY_PATH (ld.so - the Linux runtime linker)
- PYTHONPATH (Python)

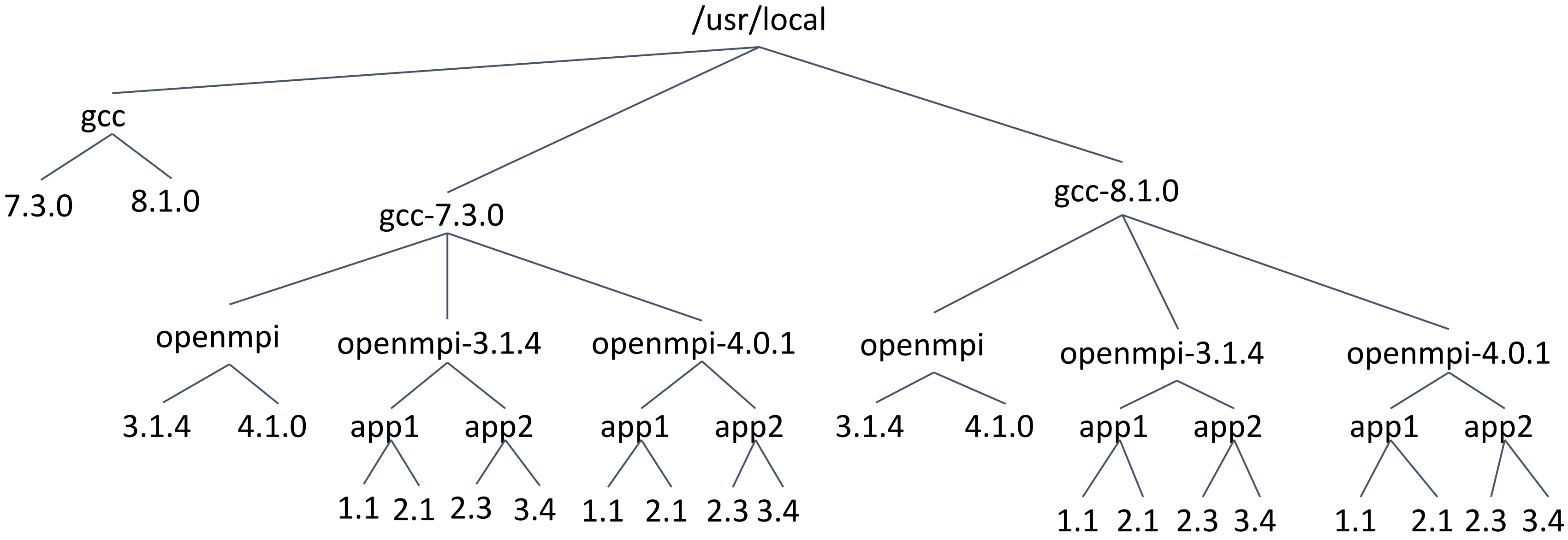
Environment Variables - the bottom line

- When building/installing software, we may need to use environment variables to tell the build process where to look for files needed by build process
- After software is installed, we almost always need to define environment variables so
  - We can find and execute the software (PATH, LD_LIBRARY_PATH)
  - Tell the software about its environment so it works properly (whether to provide verbose output, etc.)

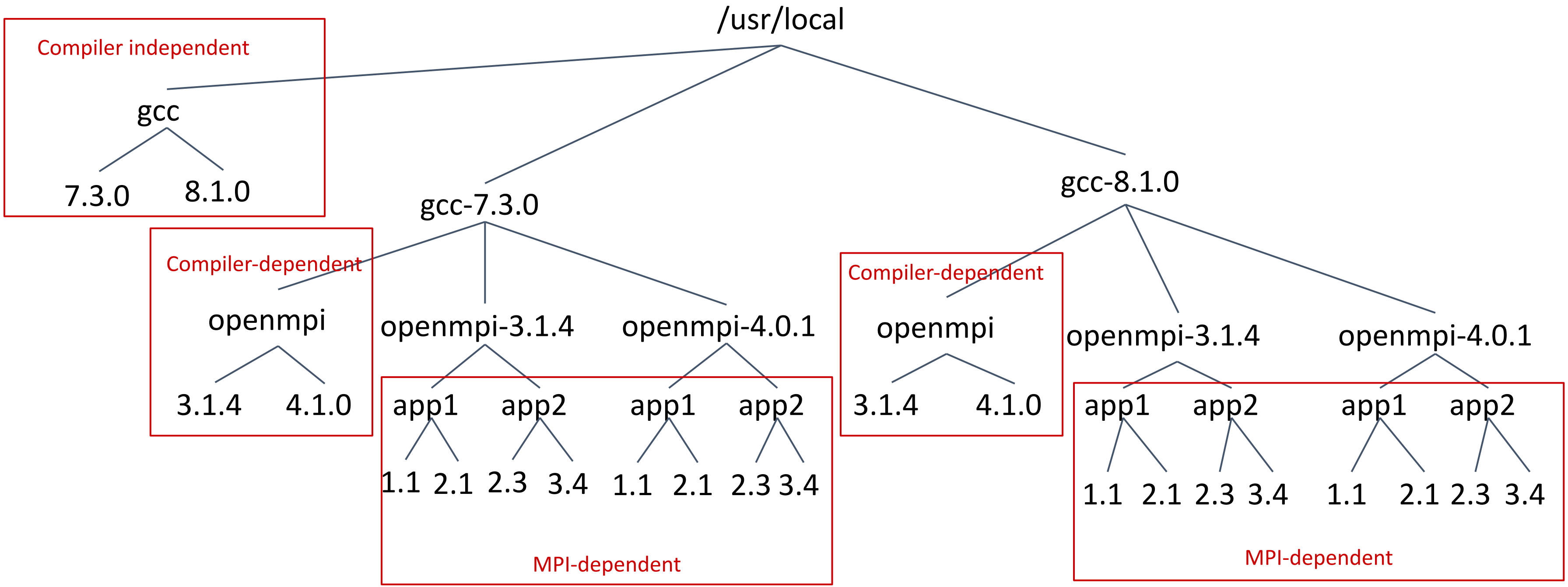
An example software installation layout

- Software will be installed in /usr/local
- Each application will be installed in its own directory under /usr/local
- Each application will be installed in /usr/local/<application name>/<version>
- Examples:
  - /usr/local/gcc/8.1.0
  - /usr/local/fftw/3.1.5
- The install directory is also known as the prefix, which we will need to specify when compiling the software
- We will build with multiple versions of OpenMPI

Hierarchical Software Tree



Hierarchical Software Tree



Installation Prefix

- An installation prefix is the full path to the directory where the software will be installed.
- Most configuration mechanisms allow you to specify a prefix
- Some examples of prefixes from the previous slide:
  - /usr/local/gcc/7.3.0
  - /usr/local/gcc-7.3.0/openmpi/3.1.4
  - /usr/local/gcc-8.1.0/openmpi-4.0.1/app2/2.3

Autoconfiguration tools

- GNU Autoconf and CMake are the most popular
- Try to preprocess, compile, link and/or execute small code snippets to determine
  - Paths to header files or their names
  - Determine what functions a library provides
  - Syntax of those functions
  - Whether they can execute the code they are producing
  - Output format of executables (ELF, etc.)
  - If they can find and link to the libraries needed by the application

GNU Autoconf Configure Script

- Generated with GNU AutoConf tools
- Created by the software's author(s)
- Checks environment for prerequisites, and determines proper settings for the build environment
- Creates makefiles with the correct settings
- Allows the user to specify various options for building the software:
  - Installation location (prefix)
  - Enable/disable certain features
  - Which compiler(s) to use
- `Configure --help` will display all available options for a package

The Open Source “5-step”

```
1.  tar xvf app-1.2.3.tgz
2.  cd app-1.2.3
3.  ./configure
4.  make
5.  make install
```

Issues with this:

- Doesn’t specify a prefix
- Doesn’t specify any other configuration options
- Configure/make should be in a separate build directory
- No ‘make check’
- Only ‘make install’ needs to be done with root privileges

Improved Open-source “5-step”

1. tar xvf app-1.2.3.tgz
2. cd app-1.2.3
3. mkdir build
4. cd build
5. ../configure\
6. --prefix=/usr/local/app/1.2.3\
   CC=gcc 2>&1 | tee configure.log
7. make 2>&1 | tee make.log
8. make check 2>&1 | tee check.log
9. sudo make install 2>&1 | install.log

Improvements:

- Separate build directory
- Specifies install prefix
- Specifies compiler with CC=...
- Creates logs for each step
- Runs ‘make check’ to verify software works
- Uses sudo to do only ‘make install’ as root.

An example of a configure command

```
../configure \
  --prefix=/usr/local/app/1.2.3 \
  --disable-silent-rules \
  --enable-shared \
  --enable-static \
  --with-foo=/path/to/foo \
  CC=gcc \
  CXX=g++ \
  FC=gfortran \
  CFLAGS=-I/path/to/include \
  LDFLAGS=-L/path/to/lib \
```

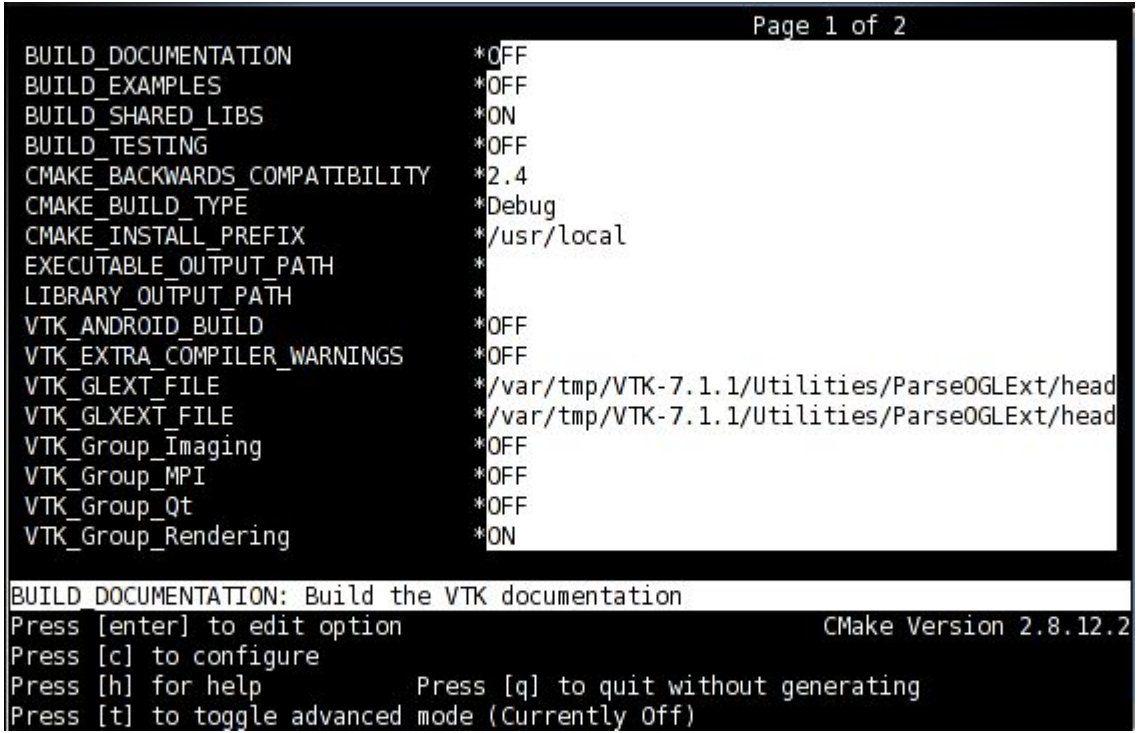
Notes:

- Replace gcc, g++, gfortran with mpicc, mpicxx, and mpif90 if compiling MPI application
- Every application will have different options
- Always use ../configure --help to see which options are available for a specific package

CMake - Another configuration method

- Not as popular as GNU AutoConf configure script
- Also needs to be setup by developer
- Same functionality as GNU AutoConf configure script
- Creates makefiles with the correct settings
- Requires a separate build directory
- ccmake command provides a TUI for specifying options

CMake's TUI



```
                                                    Page 1 of 2
 BUILD_DOCUMENTATION              *OFF
 BUILD_EXAMPLES                   *OFF
 BUILD_SHARED_LIBS                *ON
 BUILD_TESTING                    *OFF
 CMAKE_BACKWARDS_COMPATIBILITY    *2.4
 CMAKE_BUILD_TYPE                 *Debug
 CMAKE_INSTALL_PREFIX             */usr/local
 EXECUTABLE_OUTPUT_PATH           *
 LIBRARY_OUTPUT_PATH              *
 VTK_ANDROID_BUILD                *OFF
 VTK_EXTRA_COMPILER_WARNINGS      *OFF
 VTK_GLEXT_FILE                   */var/tmp/VTK-7.1.1/Utilities/ParseOGLExt/head
 VTK_GLXEXT_FILE                  */var/tmp/VTK-7.1.1/Utilities/ParseOGLExt/head
 VTK_Group_Imaging                *OFF
 VTK_Group_MPI                    *OFF
 VTK_Group_Qt                     *OFF
 VTK_Group_Rendering              *ON

BUILD_DOCUMENTATION: Build the VTK documentation
Press [enter] to edit option                          CMake Version 2.8.12.2
Press [c] to configure
Press [h] for help           Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

How to build an application with CMake

1. tar xvf app-1.2.3.tgz
2. cd app-1.2.3
3. mkdir build
4. cd build
5. `ccmake ../`   (This brings up the TUI)
6. Press “C” to run the initial configure step (configuration options appear)
7. Use arrow keys to move back and forth between options and change them
8. Press “C” again to reconfigure the settings
9. Press “G” to generate new files and exit
10. make
11. make check
12. sudo make install

Post-installation - defining environment variables

In users “rc” files:

- Have to communicate to users what to set
- Everytime you upgrade, you either need to install to the same location, or need to tell the users to update their “rc” scripts.
- If new version is installed to new location, users can continue using old version
- Only allows one version to be used easily.
- Takes effect on login

In /etc/profile.d:

- Don’t need to communicate environment change to users
- Need to make changes in only one place
- Need to create separate files for Bourne and Csh shells (Zsh?)
- Changes can catch users by surprise
- Only allows one version to be used.
- Takes effect on login

Environment modules - an easier way

- Allows users to quickly and easily modify their environment to switch between different versions of an application
- Shell-independent - one module file will affect all shells (Bourne, Csh, etc.)
- Can prevent mutually-exclusive settings from being made
- Two main environment module applications
    - Environment Modules
    - Lmod

Automating Combinatorial Builds

- Old way - take good notes of your build process for one build so that you could easily make changes to and cut-and-paste your previous commands for the next build, or script
- New ways:
  - Spack
  - EasyBuild

End of Part 1

Intermission