

Installing and Using Software on CARC Systems

Cesar Sul

Research Computing Associate



USC

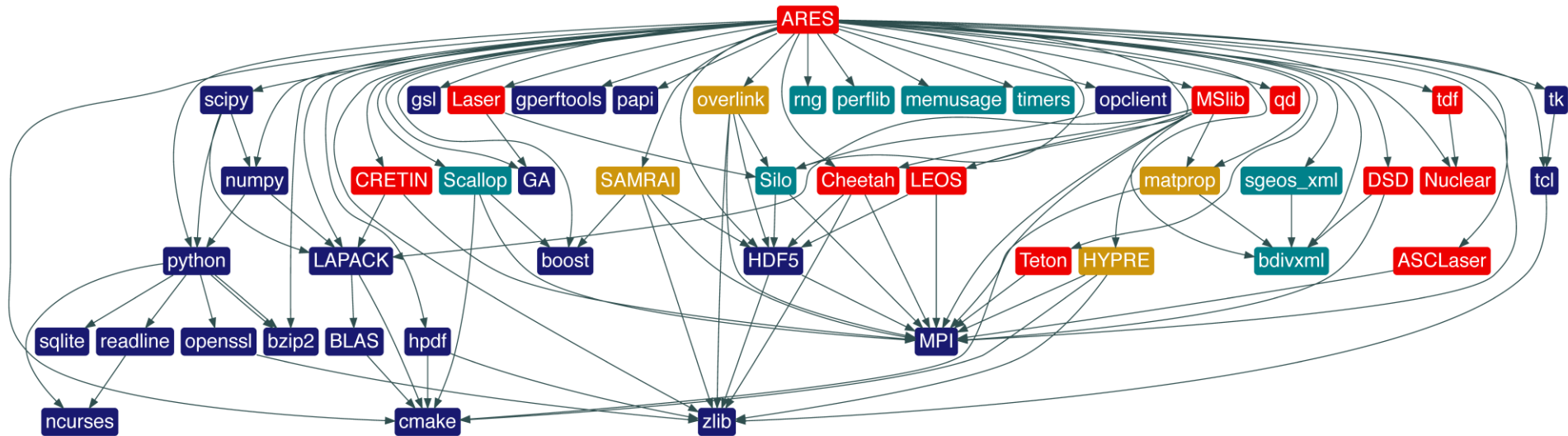
Advanced Research Computing
Enabling scientific breakthroughs at scale

Outline

- CARC managed software
 - Finding software
 - Using modules
 - Neat features
- Installing software
 - Precompiled binary
 - Conda
 - Python
 - R
 - Singularity
 - Building Source Code

Software is complex!

- Dependency graph for ARES



<https://computing.llnl.gov/projects/spack-hpc-package-manager>

What are software modules?

- Modules present installed software to users
- Set environment variables
 - PATH
 - PKG_CONFIG_PATH
 - LD_LIBRARY_PATH
 - <SOFTWARE>_ROOT
- Show how package was built
- Show where package was installed to
- Prevent loading incompatible software
- Written in Lua

What are software modules?

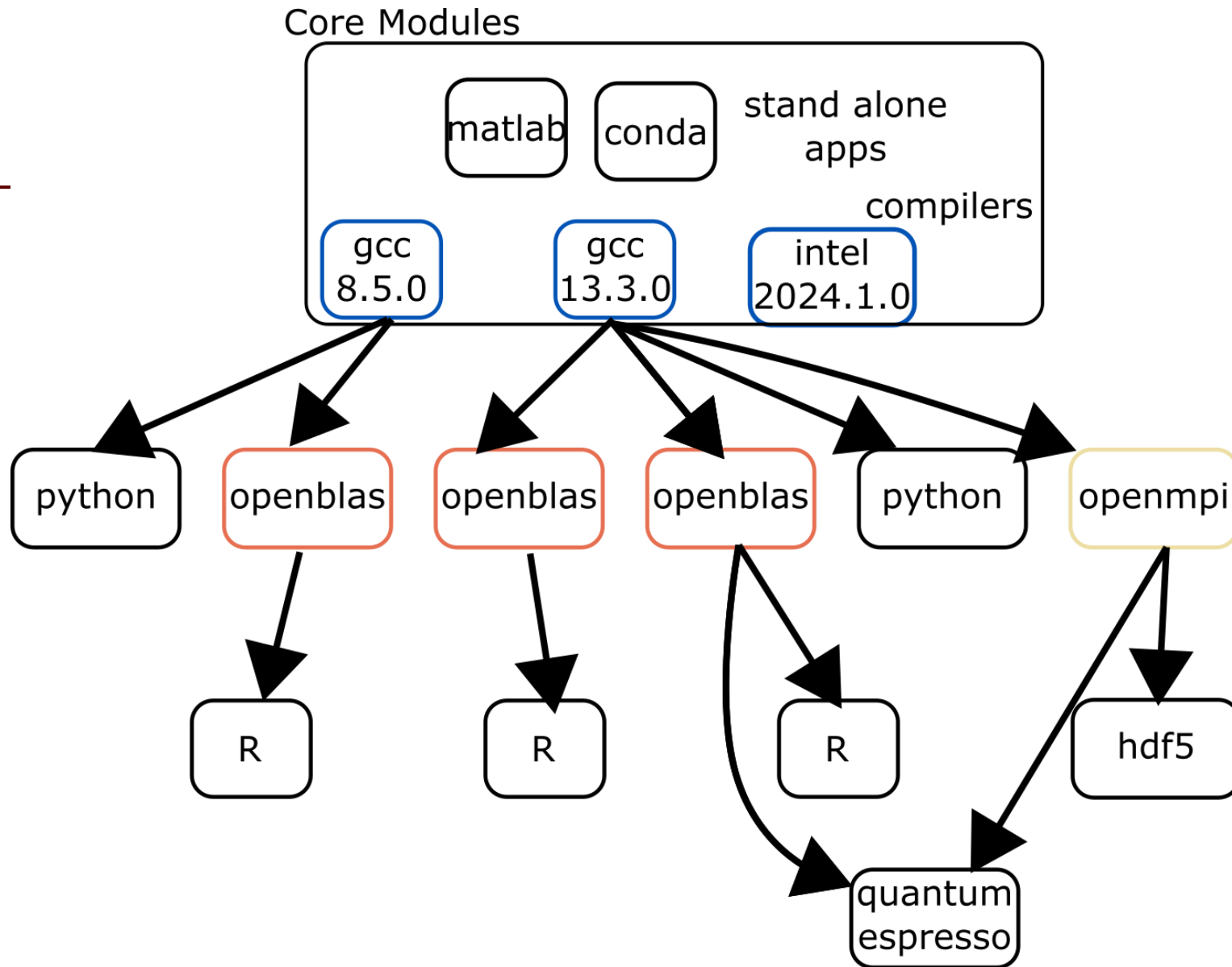
- There are 4 kinds of modules
 - Utilities
 - git, screen, tmux, wget,
 - Compiler (gcc, intel)
 - BLAS
 - openblas, mkl, amd-blis
 - Math Library
 - MPI
 - OpenMPI, mvapich2, IntelMPI
 - Multi node communication
 - Application
 - most common, bamtools, cmake, libpng...
 - Other
 - Matlab, Mathematica, julia,

What are software modules?

- By default you have the recommended "usc" module loaded

```
$ module list
Currently Loaded Modules:
  1) gcc/13.3.0          3) openblas/0.3.28    5) usc/13.3.0
  2) python/3.11.9      4) openmpi/5.0.5
```

- You can check what's available with `module avail`
- Depending on active modules, you will see different results from `module avail`



Example module avail listing

\$ module avail

[/spack/apps/lmod/linux-centos7-x86_64/openmpi/4.0.2-ipm3dnv/openblas/0.3.8-2no6mfz/gcc/8.3.0](#)

cantera/2.4

Applications built with gcc 8.3.0 compiler
AND openmpi 4.0.2 AND openblas 0.3.8

openblas-openmpi

hypre/2.18

[/spack/apps/lmod/linux-centos7-x86_64/openmpi/4.0.2-ipm3dnv/gcc/8.3.0](#)

hdf5/1.10.6-openmpi

Applications built with gcc 8.3.0 compiler AND openmpi 4.0.2

netcdf-c/2.29-openmpi

hmmer/3.3-openmpi

matio/1.5.13-openmpi

parmetis/4.0.3-openmpi

sundials/5.1.0-openmpi (D)

[/spack/apps/lmod/linux-centos7-x86_64/openblas/0.3.8-2no6mfz/gcc/8.3.0](#)

r/3.4.4

Applications built with gcc 8.3.0 compiler AND openblas 0.3.8

adapterremoval/2.3.1

kbproto/1.0.7

pdt/3.25.1

anaconda3/2019.03

Applications built with gcc 8.3.0 compiler

er/2.173

argtable/2.13

ale/1.05

at-spi2-atk/2.26.2

lcms/2.9

perl-extutils-config/0.008

How to use modules

- Use **module avail** to see what's available
- Use **module load** to load the module

```
$ which python  
/usr/bin/python
```

```
$ module load python
```

```
$ which python  
/apps/spack/2406/apps/linux-rocky8-x86_64_v3/gcc-  
13.3.0/python-3.11.9-x74mtjf/bin/python
```

- Some modules 'unlock' more modules
 - Compiler
 - MPI
 - BLAS

Finding modules

- Use **module spider** to search for software that's not available
 - Might be hidden due to prerequisites

```
$ module spider r/4.4.2
```

```
-----  
r: r/4.4.2  
-----
```

You will need to load all module(s) on any one of the lines below before the "r/4.4.2" module is available to load.

```
gcc/13.3.0
```

Saving sets of modules

- If you find yourself loading a set of modules frequently

<code>module save</code>	Save current modules to default collection
<code>module save <name></code>	Save current modules as <code><name></code> collection
<code>module restore</code>	Load modules in default collection
<code>module restore <name></code>	Load modules in <code><name></code> collection
<code>module describe <name></code>	Show which modules are in <code><name></code> collection
<code>module savelist</code>	Show names of all collections

Creating your own modules

- Check our Discourse page for more details
- [How to: Create our own modules](https://hpc-discourse.usc.edu/how-to-create-your-own-modules/149)

The screenshot shows a web browser displaying a Discourse forum post. The browser's address bar shows the URL `https://hpc-discourse.usc.edu/how-to-create-your-own-modules/149`. The page header includes the USC logo and the text 'Advanced Research Computing' and 'Enabling scientific breakthroughs at scale'. A blue banner at the top of the post area contains the text: 'To make launching your new site easier, you are in bootstrap mode. All new users will be granted trust level 1 and have daily email summary emails enabled. This will be automatically turned off when 50 users have joined.'

The main heading of the post is 'How to: Create your own modules' with a pencil icon. Below it, the breadcrumb trail reads 'HPC Cluster: Discovery > Software Modules'. The post is by a user named 'csul' and was posted '4d' (4 days) ago. The post content begins with: 'After building and installing your own software, you might find it convenient to add it to your personal module tree. There are three basic steps: 1. Create module directory tree 2. Add module directory tree to \$MODULEPATH 3. Create module file from template'. It then says 'Read below for more details on each step.' and proceeds to '1. Create module directory tree'. The text explains that a directory tree should be set up where all module files are saved, with the root being `$MODULE_ROOT`. It provides an example path: `/spack/apps/mod/linux-centos7-x86_64/`. It also mentions that CARC manages many applications with separate module trees for compiler, MPI, and BLAS libraries.

A diagram illustrates the directory structure. At the top is a yellow box labeled `$MODULE_ROOT`. Below it are two red boxes labeled `software1/` and `software2/`. Under `software1/` is a blue box labeled `ver-1.lua`. Under `software2/` are two blue boxes labeled `ver-1.lua` and `ver-2.lua`.

The text continues: 'In this example there are 2 directories under `$MODULE_ROOT`, `software1` and `software2`. Each of these software directories has at least 1 module file written in Lua (don't worry, there's a template you can use below). 2. Add module directory tree to \$MODULEPATH'. It concludes with: 'After creating the directory tree you can add it to your `$MODULEPATH` by running the command'.

Installing Your Own Software



USC

| Advanced Research Computing
Enabling scientific breakthroughs at scale

Installing software

- Installing software can be quick and painless
 - With precompiled binaries for your specific operating system, it can be as easy as unzipping a file
- ... Or neither quick nor painless
 - If you have to compile the software yourself using compilers, linkers, Makefiles, external libraries, etc.
- (or even worse...)
 - If it's from an academic lab from 1999 and requires old versions of multiple libraries which have multiple dependencies!

Installing software

- Generally speaking, software can be installed globally or locally
 - On your laptop, you are the system administrator
 - On HPC, you are not the system administrator
- Globally means system-wide
 - Software is installed to system locations like [/usr/bin](#) or [/usr/local](#)
 - Global installs require root privileges
- System-wide installations will not work on HPC
 - Only systems administrators have root privileges on HPC
 - E.g., “yum install” and “apt install” will not work



Installing software

- CARC users must perform local, or “user”, installs
 - Software installed to 'local' folders
 - `/project/<pi_id>/software`
 - Requires write privileges, which you have in your own directories
 - Software will be accessible by you, even on compute nodes
- It is not always obvious how to perform a user install
 - Depends on software
 - You may have to check documentation

Precompiled binary

- Simplest case
- Just download and extract
- Not always available

```
$ cd /project/ttroj_412/software
```

```
#Copy tarball
```

```
$ wget https://example.com/sample.tar.gz
```

```
#Extract files
```

```
$ tar xvf sample.tar.gz
```

```
#Set your environment (adds a new location to your path)
```

```
$ export PATH=/project/ttroj_412/software/sample/bin:${PATH}
```

```
#Test installation
```

```
$ binary_name
```

Conda Environments



USC

Advanced Research Computing
Enabling scientific breakthroughs at scale

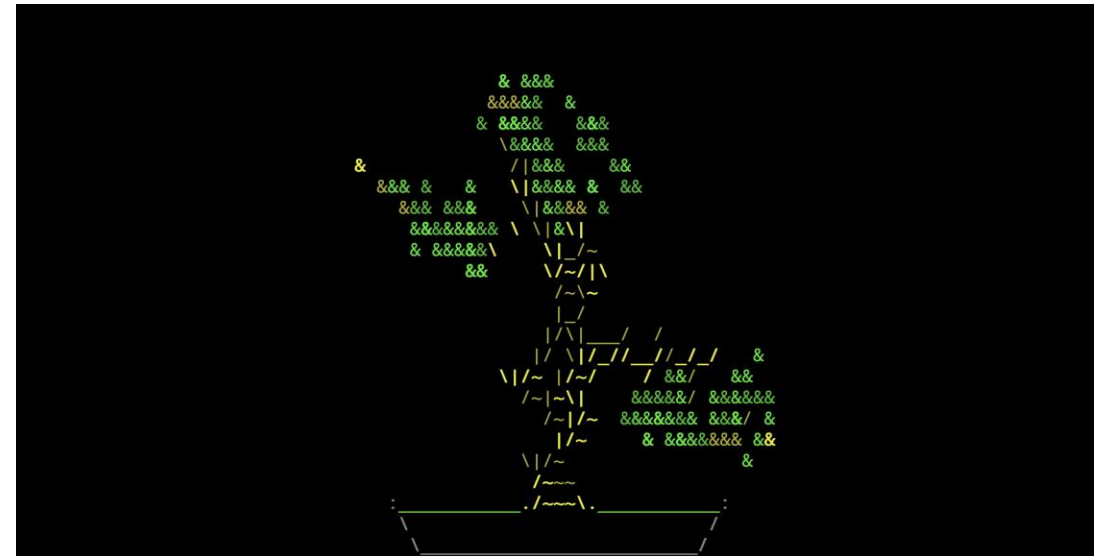
Conda Environments

- Some developers package their software as conda environments
- Create a collection of software packages
- Dependencies are managed for you
- `mamba env create -f environment.yml`
- Full documentation [here](#)

```
name: mustache
channels:
  - conda-forge
  - defaults
dependencies:
  - pip
  - h5py
  - hdf5
  - numpy
  - python=3.8
  - pip:
    - cooler
    - hic-straw
    - pandas
    - scipy
    - statsmodels
```

Our own environment

- We can also create our own custom environments
- Let's use "cbonsai" example
- Prints ascii art of bonsai tree



Our own environment

- Set up conda

```
$ module load conda
```

```
$ mamba init bash
```

- Create and name environment

```
$ mamba create -n cbonsai
```

```
# Enter environment
```

```
$ mamba activate cbonsai
```

- Install packages

```
$ mamba install cbonsai
```

```
$ mamba install cbonsai
```

```
.  
.br/.
```

Package	Version	Build	Channel	Size
---------	---------	-------	---------	------

Install:

+ _libgcc_mutex	0.1	conda_forge	conda-forge	3kB
+ libgomp	14.2.0	h77fa898_1	conda-forge	461kB
+ _openmp_mutex	4.5	2_gnu	conda-forge	24kB
+ libgcc	14.2.0	h77fa898_1	conda-forge	849kB
+ ncurses	6.5	h2d0b736_3	conda-forge	892kB
+ cbonsai	1.3.1	haa1a288_0	conda-forge	30kB

Summary:

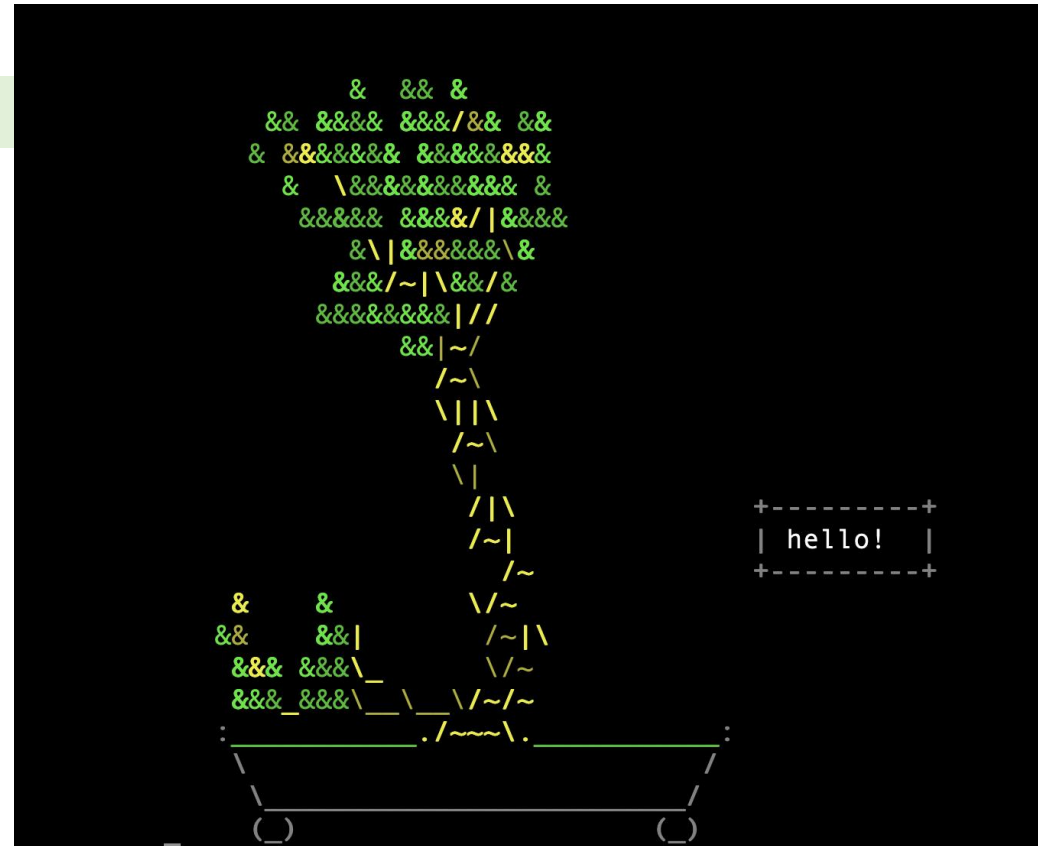
Install: 6 packages

Total download: 2MB

Our own environment

- Run cbonsai

```
$ cbonsai -m hello! -p
```



Installing Python Packages



USC

| Advanced Research Computing
Enabling scientific breakthroughs at scale

Installing Python packages

- Don't forget to load the version of python you want to use

```
$ module load python/3.7.6
```

- To check what packages are available use the command

```
$ pip list
```

- Install package (bash shell)

```
$ pip install <package_name> --user
```

- Sometimes you'll need to install the latest version of a package that is already installed

```
$ pip freeze  
$ pip install <package_name> --upgrade --user
```

Dependencies for Python packages

- Some packages are Python wrappers for C/C++ libraries
- The installer needs to know where these libraries are
- The [h5py](#) package is one example

```
$ HDF5_DIR=/path/to/hdf5  
$ HDF5_VERSION=X.Y.Z  
$ CC="mpicc"  
$ pip install h5py --user
```

- You might have to download the package tarball and edit some files like setup.py

Installing R Packages



USC

| Advanced Research Computing
Enabling scientific breakthroughs at scale

Installing R packags

- Source the version of R you want to use and start R

```
$ module load r  
$ R
```

- Install package syntax (you may have to specify a path)

```
> install.packages('<package_name>')  
> install.packages('<package_name>', lib="/path/to/packages")
```

- Then load the library when you want to use

```
> library('<package_name>')  
> library('<package_name>', lib.loc="/path/to/packages")
```

Dependencies for R packages

- Some packages are R wrappers for C/C++ libraries
 - The installer needs to know where these libraries are
 - You might have to download the package tarball and edit some files
- You can set compilation environment variables like `LD_FLAGS` in the file ``${HOME}/.R/Makevars`

Singularity/Containers



USC

| Advanced Research Computing
Enabling scientific breakthroughs at scale

Apptainer



-
- Formerly Singularity
 - For difficult installations
 - Apptainer provides packaged "computing environments"
 - Works best with complex dependency chains
 - Compatible with Docker

Apptainer

Example: [lolcow](#) (date | cowsay | lolcat)

```
#Download container image
$ apptainer pull docker://ghcr.io/apptainer/lolcow

#Test
$ singularity run lolcow_latest.sif
```

See this page for more ways to interact with a container:
https://apptainer.org/docs/user/latest/quick_start.html#interacting-with-images

```
[ttroj@discovery playground]$ apptainer run lolcow_latest.sif
```

```
< Fri Feb 7 09:26:29 PST 2025 >
```

```
-----
 \  ^__^
  \ (oo)\_______
    (__)\       )\/\
       ||----w |
       ||     ||
```


Building Source Code

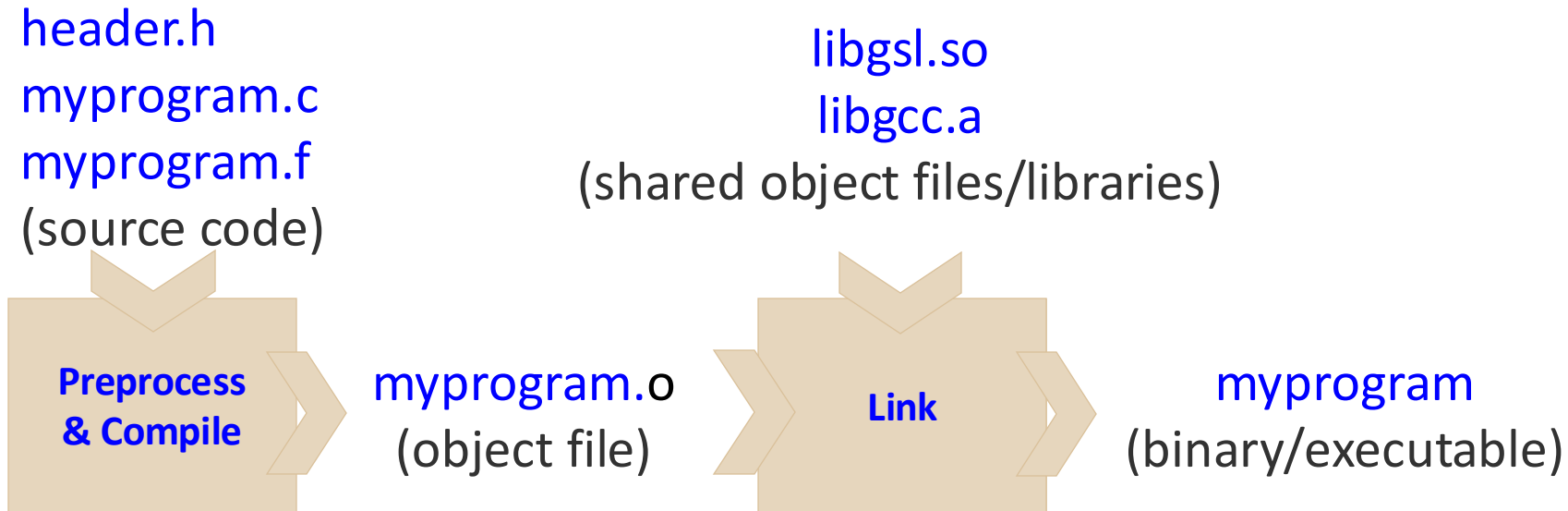


USC

| Advanced Research Computing
Enabling scientific breakthroughs at scale

Compiling to source code

- C/C++ and Fortran programs are compiled and assembled



Compilers

- A typical compile command for C code

```
$ gcc ${CCFLAGS} source.c ${CPPFLAGS} ${LDFLAGS} -o myprogram
```

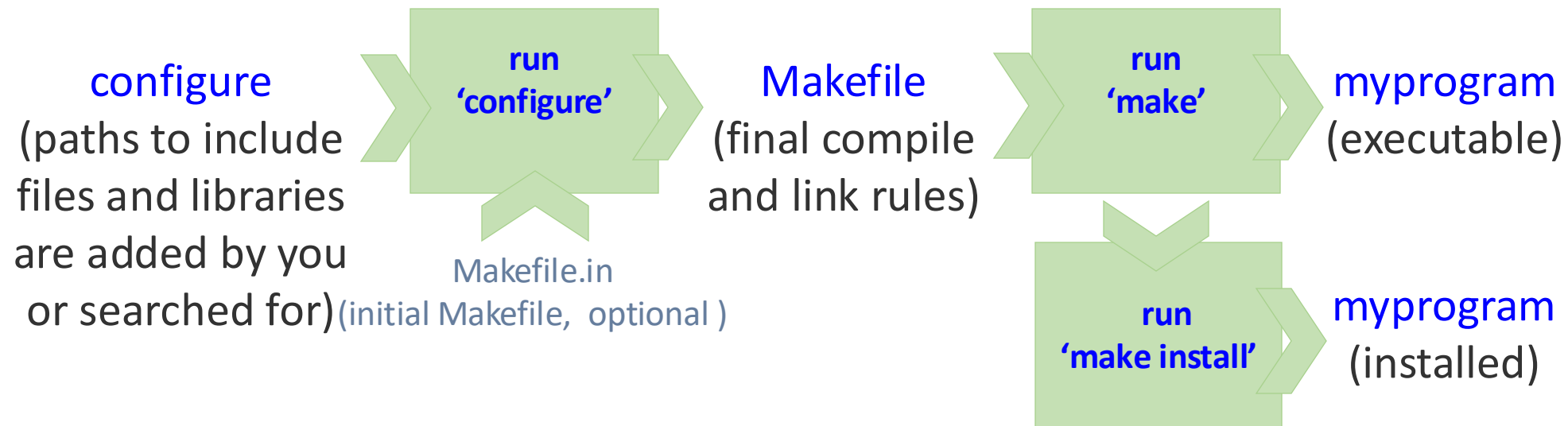
- Where these environment variables were pre-defined

```
$ CCFLAGS='-Wall -O3'  
$ CPPFLAGS='-I/path/to/include'  
$ LDFLAGS='-L/path/to/lib -lgsl -lgslcblas -lm'
```

CCFLAGS	Flags to pass the C compiler
CPPFLAGS	Where the C preprocessor can find include (.h) files
LDFLAGS	Which libraries (.so, .a files) to use and where the linker can find them

With configure/make

- Manually typing compile and link commands is not feasible
- Software build utilities like autotools, cmake handle this



Building software with modules

- Most modules modify `$PKG_CONFIG_PATH`
- Many installer scripts check here for prerequisite software

```
$ module load ncurses
$ ./configure <options>
...
checking curses.h usability... yes
checking curses.h presence... yes
checking for curses.h... yes
checking ncurses.h usability... yes
checking ncurses.h presence... yes
checking for ncurses.h... yes
...
```

- If we're lucky it's that easy

Building software with modules

- If unlucky, specify with script options

```
module load ncurses
./configure --ncurses-root=${NCURSES_ROOT} <other options>
...
checking curses.h usability... yes
checking curses.h presence... yes
checking for curses.h... yes
checking ncurses.h usability... yes
checking ncurses.h presence... yes
checking for ncurses.h... yes
...
```

- Our modules set environment variable `<software>_ROOT` for just this occasion

Building software with modules

- If very unlucky, modify Makefile

```
...  
override LDFLAGS += -L./nicksrc -L$(GSL_ROOT)/lib  
-L$(OPENBLAS_ROOT)/lib  
  
override CFLAGS += -c -g -p -Wimplicit -I./ -I./nicksrc  
-I$(GSL_ROOT)/include -I$(OPENBLAS_ROOT)/include  
...  
  
$ module load ncurses  
$ make
```

Compiling source code



Example: vim

#Download tarball

```
$ git clone https://github.com/vim/vim.git
```

```
$ cd vim
```

#Run configure script

```
$ ./bootstrap
```

```
$ ./configure --prefix=/project/<pi_id>/<username>/vim [other options]
```

#Run makefile

```
$ make
```

```
$ make install
```


Compiling source code

- Configure script can't find libtinfo.so (provided by ncurses)

```
checking --with-tlib argument... empty: automatic terminal library selection
checking for tgetent in -ltinfo... no
checking for tgetent in -lncurses... no
checking for tgetent in -ltermcap... no
checking for tgetent in -ltermcap... no
checking for tgetent in -lncurses... no
no terminal library found
checking for tgetent()... configure: error: NOT FOUND!
```

Compiling source code

- In this case, pkg-config is not used
- Manually override with pkg-config, \$LDFLAGS, and \$CXXFLAGS

```
$ pkg-config --cflags-only-I ncurses
```

```
-D_GNU_SOURCE -I/spack/apps/linux-centos7-x86_64/gcc-8.3.0/ncurses-6.1-  
akiyo4qrgzlxw3hggkc42nvv7hz2evj/include
```

```
$ pkg-config --libs-only-L ncurses
```

```
-L/spack/apps/linux-centos7-x86_64/gcc-8.3.0/ncurses-6.1-akiyo4qrgzlxw3hggkc42nvv7hz2evj/lib
```

Compiling source code



Example: vim

```
#Download tarball
$ git clone https://github.com/vim/vim.git
$ cd vim

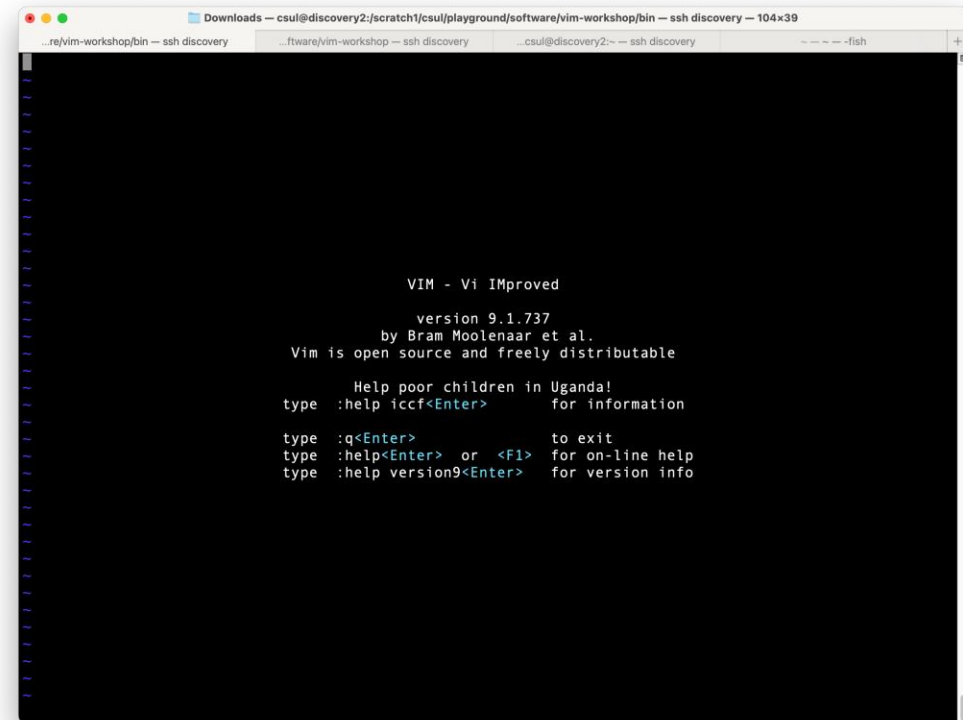
#Run configure script
$ CPPFLAGS=$(pkg-config --cflags ncurses) \
  LDFLAGS=$(pkg-config --libs-only-L ncurses) \
  ./configure \
  --prefix=/project/<pi_id>/<username>/vim \
  --with-tlib=tinfo

#Run makefile
$ make
$ make install
```

Compiling source code

Test installation

```
$ ./vim # :q! to quit
```



Getting Help

- Request assistance
 - Email carc-support@usc.edu
 - Office Hours (drop-in)
 - Every Tuesday@2:30pm (Zoom)
- Learn more!
 - Visit carc.usc.edu
 - Request a consultation (anytime)
 - Attend a Workshop (when scheduled)
 - [Visit our Discourse page!](#)



Thank you for attending!

Questions?

carc-support@USC.EDU



USC

Advanced Research Computing
Enabling scientific breakthroughs at scale