Providing 3rd Party Software: Introducing EasyBuild & Singularity

•••

Jared D. Baker
Advanced Research Computing Center
University of Wyoming

ARCC UW/IT

RMACC
Rocky Mountain Advanced Computing Consortium

Providing 3rd Party Software in HPC / Research Computing

Why?

1. Because research is a moving target and often users are not necessarily programmers or systems admin savvy. They just need to work with package X.
2. For some reason, many software packages seem to not leverage standard utilities to configure, build, and install the software and can be difficult to understand system related issues.
3. We like our users to focus on getting science done, not fighting with the machine for which they have little control.

Traditional Methods on Linux - System Package Manager

1. System package managers (obviously many more)
    a. Yum (RPM based)
    b. Zypper (RPM based)
    c. APT (deb based)

These are great for running a base system: SSH servers, shells, filesystems, network connectivity.

Often, only single versions of software is available, compiled with very generic optimizations to reach wide audience, likely compiled with a single compiler of the GCC world and perhaps a very old one.

Supports multiple versions...well, from your own repositories pending naming scheme.

Large Binary Installers

Often distributed for licensed software:

Intel Parallel Studio, PGI Compilers

Gaussian, ADF, COMSOL, CMG, FLUENT, etc.

Often statically compiled or comes with a set of wrapper scripts to run the software.

Allows multiple versions by varying the installation prefix in some manner.

Problems: Large installation footprint (small issue)
comes with own set of libraries

Source Builds - Looking for Performance / Multiple Versions

Target a specific instruction set → SSE4.1, AVX, etc.

Build with a specific compiler → Intel, PGI, Clang, etc.

Enable/Disable extra package features

Problems:

1. ABI compatibility (name mangling in Fortran / C++) → require same compiler all the way through (or at least within a similar version)
2. Requires all dependency development files / packages as well
3. Understanding all different methods to compile different software.

University of Wyoming Story

Mount Moran → First “condo” HPC cluster in 2012

TCL environment modules with GCC/4.4 compiled software

Intel Compilers, PGI compilers requested by users

Attempt to write and maintain custom written scripts to build each package

Admin dependent, no coding styles, near 0 comments, no version control

~ 2014 transition to Lmod as replacement for TCL modules to leverage hierarchy

Users attempting to compile codes with different compiler-driven libraries.

We didn’t have well written modulefiles so this helped dramatically... for our users

Custom Written Build Scripts

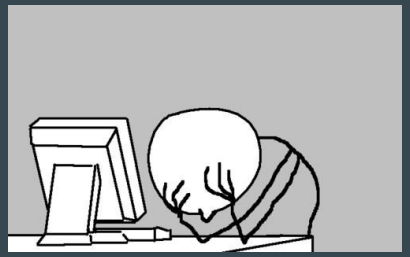
Not really a systematic approach to building software

Admins have different styles of writing scripts (no style guidelines)

Serves the basics of audit capability

Site errors:

- No use of version control (i.e., Git, Mercurial, etc.)
- Lack of comments (present me really hates past me at times)
- Lack of rules involving installation locations. Leads to manual modulefile writes for each individual install, not just package.



How can we improve?

Find a systematic way to build and install software:

- Develop our own framework
  - Looks fun and interesting
  - Already struggling with time commitments
- Look for a framework that already exists
  - A few solutions out there with dedicated teams and communities
  - May not fit our cluster personality per se

Build Systems

EasyBuild → https://github.com/easybuilders

Maali → https://github.com/Pawseyops/maali

Spack → https://github.com/LLNL/spack

Frameworks do exists. GREAT. Seem to have substantially large pieces. EasyBuild has 3 components separating functionality at the moment, Maali is shell based and has ~250 variables that can be set, Spack is intuitive and delivers a powerful domain specific language (DSL).

Administrator and user focused.

Heavy integration with a modules system (Lmod is a good choice)

REALLY vibrant community:

IRC, mailing list, Github

LOTS of packages in the repository ready to be built on your machine.

Maali

Interesting project

+1 for being in BASH

Lack of a larger community

Integrates modules → TCL Environment Modules

Github presence is a small number of contributors, small number of forks

Spack

Project from Lawrence Livermore National Lab (LLNL)

Written in Python

Superior command line (IMHO)

```
$ spack install hdf5@1.10.1 %gcc@4.7.3 +szip
```

Very good DSL, leverages RPATH

Focus on power users initially and required built in activation

This has changed quite a bit since SC15.

EasyBuild: The Basic Installation

First time = bootstrap install → `$ python bootstrap_eb.py $PREFIX`

Future installs using EasyBuild → `$ eb --install-latest-eb-release`

Make sure you have Environment Modules (TCL) or Lmod installed first

Follow on screen instructions to add EasyBuild modules directory to your $MODULEPATH environment variable in the appropriate method.

```
$ module load EasyBuild
```

Now to use EasyBuild!!! Perhaps.

EasyBuild: Tuning Installation as SW provider

Changing the install path: (e.g., /apps/Core):

```
--installpath-software=<path>    $EASYBUILD_INSTALLPATH_SOFTARE
```

Changing the modules path (e.g., /apps/Core/lmod/mf/Core):

```
--installpath-modules=<path>$EASYBUILD_INSTALLPATH_MODULES
```

Setting modules tool [tcl|lmod]:

```
--modules-tool=<spec>            $EASYBUILD_MODULES_TOOL
```

EasyBuild: Module Naming Scheme

I personally like a hierarchy of modules showing ABI compatibility:

```
EASYBUILD_MODULE_NAMING_SCHEME=HierarchicalMNS
```

Others prefer the standard EasyBuild naming schemes or something other.

Use a VM to experiment to see which fits in your environment the best.

1. Reset the values for the `$EASYBUILD_INSTALLPATH_SOFTWARE`
2. Reset values for `$EASYBUILD_INSTALLPATH_MODULES`
3. Reset `$MODULEPATH` to exclude previous built software by EasyBuild
4. Set `$EASYBUILD_MODULE_NAMING_SCHEME` to desired value

EasyBuild: Toolchains → Basics of Building Software

EasyBuild utilizes a concept of a **toolchain**.

A **toolchain** is a collection of specific software defining the base for building software.

A **toolchain** utilizes more leading edge software that what usually comes with the operating system on the HPC system.

Examples:

1. foss → Free, Open Source Software (GCC, OpenMPI, OpenBLAS, LAPACK, FFTW,...)
2. Intel → Intel compiler based tools (Intel Compilers, Intel MPI, Intel MKL, ...)

EasyBuild: Installing software

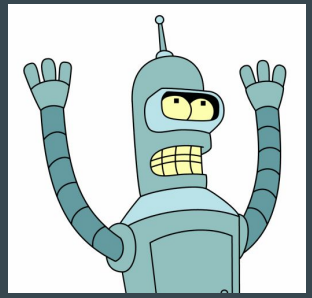
Installing GCC version 6.3.0

```
$ module load EasyBuild

$ eb -S ^gcc

$ eb --dry-run GCC-6.3.0-2.28.eb
```

"Oh, I get it, make the robot do all the work!"



You should get a dry run first to see what dependencies you have currently and which ones will be required. Required dependencies can be fulfilled by using the `--robot` option to the `eb` command.

```
$ eb GCC-6.3.0-2.28.eb --robot
```

Compilers are swell, but we need infrastructure libraries too!

Toolchain: foss-2017a

```
[vagrant@localhost ~]$ eb -D foss-2017a.eb -r
== temporary log file in case of crash /tmp/eb-uRljAt/easybuild-GOXAm8.log
Dry run: printing build status of easyconfigs and dependencies
CFGS=/opt/Core/eb/Core/EasyBuild/3.3.1/lib/python2.7/site-packages/easybuild_easyconfigs-3.3.1
-py2.7.egg/easybuild/easyconfigs
 * [ ] $CFGS/m/M4/M4-1.4.17.eb (module: Core | M4/1.4.17)
 * [ ] $CFGS/b/Bison/Bison-3.0.4.eb (module: Core | Bison/3.0.4)
 * [ ] $CFGS/f/flex/flex-2.6.0.eb (module: Core | flex/2.6.0)
 * [ ] $CFGS/z/zlib/zlib-1.2.8.eb (module: Core | zlib/1.2.8)
 * [ ] $CFGS/b/binutils/binutils-2.27.eb (module: Core | binutils/2.27)
 * [ ] $CFGS/g/GCCcore/GCCcore-6.3.0.eb (module: Core | GCCcore/6.3.0)
 * [ ] $CFGS/m/M4/M4-1.4.18-GCCcore-6.3.0.eb (module: Compiler/GCCcore/6.3.0 | M4/1.4.18)
 * [ ] $CFGS/z/zlib/zlib-1.2.11-GCCcore-6.3.0.eb (module: Compiler/GCCcore/6.3.0 |
zlib/1.2.11)
 * [ ] $CFGS/h/help2man/help2man-1.47.4-GCCcore-6.3.0.eb (module: Compiler/GCCcore/6.3.0 |
help2man/1.47.4)
 * [ ] $CFGS/b/Bison/Bison-3.0.4-GCCcore-6.3.0.eb (module: Compiler/GCCcore/6.3.0 |
Bison/3.0.4)
 * [ ] $CFGS/f/flex/flex-2.6.3-GCCcore-6.3.0.eb (module: Compiler/GCCcore/6.3.0 | flex/2.6.3)
 * [ ] $CFGS/b/binutils/binutils-2.27-GCCcore-6.3.0.eb (module: Compiler/GCCcore/6.3.0 |
binutils/2.27)
 * [ ] $CFGS/g/GCC/GCC-6.3.0-2.27.eb (module: Core | GCC/6.3.0-2.27)
```

Okay, we have infrastructure ...

... But we have users who don't care about compilers, MPI, etc. I want to run my scientific application XXXX. Say GROMACS MD package.

```
$ eb GROMACS-2016.3-foss-2017a.eb -r
```

This will attempt to install GROMACS utilizing a formula specific to the foss-2017a toolchain and install any required dependencies by using the robot!

Search for your users' software using `eb -S regex`

Improving build times for large packages

Default “build” directory is `~/.local/easybuild/build`.

- My cluster `$HOME` is on GPFS system that is ready for refresh and struggles with open/close operations like compiling software.
- Use `/tmp` if SSD enabled
- Use `/dev/shm` if machine has decent memory. Be aware of others building software on the node too.

```
$ Export EASYBUILD_BUILDPATH=/dev/shm
```

EasyBuild: `eb --show-config`

See your configuration with the command:

```
$ eb --show-config     # eb --show-full-config
```

You can use this to try different things out and whatnot. When looking at the following:

1. Repositorypath (easyconfig repos)
2. robot-paths (searching for easyconfigs and source archives by robot)
3. sourcepath (source archives or binary archives)

You can prepend/append using the : character just as other PATH like variables, but try not to remove the default unless you're attempting a fresh installation of EasyBuild.

EasyBuild: Intel Compilers & Tools

Tips:

1. Download the individual component packages, not the comprehensive toolkit.
2. Place the archives in appropriate source directories:
   - Default is `~/.local/easybuild/sources`
   - Part of a team? Put this somewhere accessible by your team (globally accessible perhaps)
   - E.g., Intel MKL
     → `l_mkl_2017.1.132.tgz`
     → `~/.local/easybuild/i/imkl/l_mkl_2017.1.132.tgz`
3. Set the `INTEL_LICENSE_FILE` when installing via EasyBuild (port@host)
4. Alternatively, prepend/append path to `EASYBUILD_ROBOT_PATHS/EASYBUILD_SOURCEPATH`
   - `$ export EASYBUILD_SOURCEPATH=:/archives/intel/2017`
5. You can also use `--robot-paths` to look, but also searches for easyconfig files

Notes

When setting INTEL_LICENSE_FILE, if you load an Intel based module and then remove it, Lmod will unset the variable in your shell unless you allow duplicates in your Lmod installation. Hint: `LMOD_DUPLICATE_PATHS=yes`, but use at your own risk WRT other modules.

Heterogeneous systems may want to think carefully about the use of the EasyBuild option `--optarch` to control the optimization. By default, EasyBuild will attempt to target the hosting build machine. May need different things for different compute systems or compilers. i.e., GCC lowest instruction set, Intel “Fat” binaries?

EasyBuild: How

3 Components: EasyBuild Framework, Easyblocks, Easyconfigs

EasyBuild Framework

→The core code and flow of configuring, building, and installing software

Easyblocks

→ The build scenario template (configure, make, make install, ...)

Easyconfigs

→ The build recipe to build and install the software.

→ Requires specification of an Easyblock.

EasyBuild: Other Admin-ish Stuff of Interest

- Configuring EasyBuild to submit the build request as job on your HPC resource
- Installation tweaks (umask, setgid, hiding toolchains)
- Packaging the software using FPM to install all packages locally
- Making better use of Modules by EasyBuild
- Improving the users' environment WRT modules generated by EasyBuild
  - Setting `CC`, `CXX`, `CPP`, `CPPFLAGS`, `LD`, `LDFLAGS`, `LIBS`, etc. or loosely related cousins suitable for compiler wrappers for those users wanting to compile their own software, but utilizing EasyBuild provided modules.
  - Adding extra variables (`I_MPI_CC`) or similar
- Giving back by enabling Github integration

EasyBuild Personal Issues

1. No prefix specific config file w/o `EASYBUILD_CONFIGFILES`
2. I don't care for CaMeLCase modules. I prefer lowercase when all possible.
    - This can potentially be fixed by using a custom module naming scheme, but I still need to experiment with it.
3. Starting to many, many options (output of `--help`)
    - I'd like to see some sub commands similar to `yum`, `git`, `spack`

EasyBuild: Conclusions

1. EasyBuild is a great tool for managing HPC specific software. It really let's the smaller HPC centers really take control of their sciences by working on the packages that aren't available from the repos
2. There is a really vibrant community surrounding EasyBuild and has tons of talent!
3. There are alternatives depending on your style or what you feel. I really like Spack as well and I hope to look at comparing the two, hopefully before SC17 in Denver if anybody wants to chat about it then.

Comments or Questions

Introduction to Singularity: Containerization for HPC

Containerization: Historical Overview

Containerization: Advanced `chroot` on UNIX-like operating systems

chroot syscall in 1979 in version 7 UNIX

Added to BSD in 1982

Strengthened concept in 2000 with “jail” command

2005 → Solaris containers

2008 → LXC

... Docker, Singularity, Shifter, ...

Why use containers?

Software segregation/isolation

Reproducibility of the results and the environment performed in.

Minimize overhead of a virtualized environment / machine

Packaging of all dependencies in a single source image

End user chooses operating system (Ubuntu, Debian, Fedora, Arch, CentOS, ...)

Why Singularity?

Seems to be the most prominent in the HPC space currently.

Designed around HPC environments to run on **shared** resources.

→ Minimize root escalation

Can execute Docker images (within limits) which end users may prefer on their local workstations, desktops, laptops. Why? Docker works with Windows (sort of) & Mac too!

Fairly straightforward usage.

Singularity: Installation

```
VERSION=2.3.1
wget https://github.com/singularityware/\
singularity/releases/download/$VERSION/\
singularity-$VERSION.tar.gz

tar xvf singularity-$VERSION.tar.gz

cd singularity-$VERSION

./configure --prefix=/usr/local

make

sudo make install
```

← Basic “configure, make, make install”

Alternatively, you can utilize EasyBuild.

```
$ eb \
     Singularity-2.2.1-GCC-6.3.0-2.27.eb \
     --robot
```

Singularity: Considerations

Reduced escalation capability means limit functionality in some cases:

1. Port forwarding for cases like RStudio Server, Jupyter Notebooks, etc.
2. Certain bind mounts on older Linux systems (RHEL 6 & family)

Host drivers within the containers for GPUs and HCAs

There is still some SETUID portions in the code

Waiting for good overlay file system support still, perhaps in RHEL 7.4?

Singularity: Workflow

1. Create an image file
2. Bootstrap the image file with the appropriate operating system
3. Modify the image as necessary
    - Warning: how does this affect your reproducibility.
    - Can you regenerate this image fairly regularly.
    - Works as you're tweaking your environment.
    - Add any necessary mount points to your image.
4. Execute your image (using "run" or "exec")
    - `$ singularity run ...` → execute a predetermined command/script defined in the image definition.
    - `$ singularity exec ...` → execute a command from the command line of the user's choosing.

Singularity: Creating an Image

```
# singularity create container.img
```

```
# singularity create --size X_in_MB container.img
```

Then when you need to expand the size of the image file:

```
# singularity expand container.img
```

This will expand the container by a default of 768 MiB (in 2.3.1) at a time. It was 512 MiB in 2.2.1. You can change the growth increment using `--size` option.

Singularity utilizes Ext3 filesystems inside.

Singularity: Inspect image file by using mount

```
# singularity mount container.img
```

This can be used to do a read only inspection treating the image file as a block device utilizing a loop device and allow you to copy files between image

This spawns a `/bin/sh` session as read-only by default, but you can use `--writable` option to make it a writable image.

*The online documentation states you can specify a mount path. I don't find this working.

Singularity: Bootstrap Image

Create a spec file, generally `Singularity`, where you'll use similar syntax to RPM spec files to define the image.

```
# singularity bootstrap container.img Singularity
```

Bootstrapping can utilize Docker, Yum, Debootstrap

Spec file sections include %setup, %post, %test, %labels, %files, %environment, %runscript

The spec file is where best practices come in for creating a reproducible image. However, I'd like to see more of a comprehensive configuration management solution like Ansible capability within as an **option**.

Singularity: Running & Execution

Run scriptlet which is defined in the %runscript section of the spec file

```
$ singularity run container.img
```

Execute a specific command

```
$ singularity exec container.img /bin/bash
```

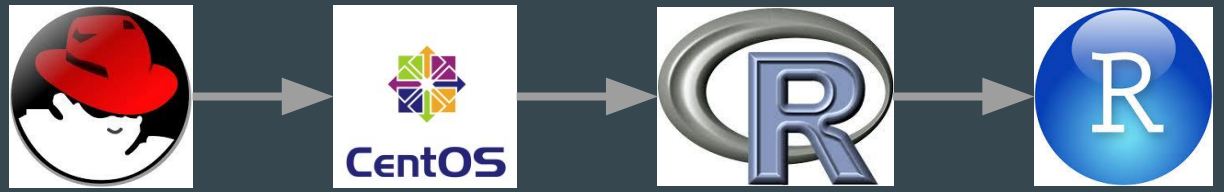
Launch a basic shell within the Singularity container

```
$ singularity shell container.img
```

Singularity: Use Case → RStudio on RHEL 6.x

Our HPC resource is currently still RHEL 6.8 series. We'll be migrating to 7.x series during our anticipated refresh cycle where we can utilize a rolling upgrade pattern. However, a firm date on this is not known.

Want to run RStudio on the HPC resource to do interactive computations, however, RStudio states it only works on RHEL 7.x or later at this point. So let's create a CentOS 7 image to run on RHEL 6.8 system to get around this limitation.



Singularity: Use Case → RStudio

I'm using a local workstation with singularity installed. First thing is to create the image file necessary.

```
$ sudo singularity create --size 8192 Cent7-RStudio.img
```

Create a Singularity bootstrap spec file to use a Yum bootstrap flavor. Pick a basic bootstrap file off the web to minimize your syntax typos. We can use this gist.

```
$ wget
https://gist.githubusercontent.com/jbaksta/5dc4307afda3bfa45cc807c36283e403/raw/d9dc5
777050ff7b04f973343bb43a8dc2e676707/Cent7-RStudio.spec
```

Singularity: Use Case → RStudio

Now, let's actually bootstrap the image

```
$ sudo singularity bootstrap ./Cent7-RStudio.img Cent7-RStudio.spec
```

Watch the bootstrap go through the basic installation of the container OS and install the necessary RPMs. This can take a few minutes. Go grab coffee or something.

When the bootstrap is finished, we'll be ready to run our container!!!

Singularity: Use Case → RStudio

Let's run our container at this point!

```
$ singularity run ./Cent7-RStudio.img
```

This should give us an interactive R CLI session. This command matches the commands given in the %runscript section of the spec file

Let's actually run R Studio though. We'll invoke this by using the `exec` subcommand of `singularity`.

```
$ singularity exec ./Cent7-RStudio.img rstudio
```

NOTE: if you get an error with QT_XKB_CONFIG_ROOT, you may need to go install the package `xkeyboard-config`

Singularity: Use Case → RStudio

Thoughts:

1. On a RHEL 7.3 host workstation, QT environment positions the cursor very weird in the R Studio editor and interpreter. The text ends up in the correct place, but quite confusing.
2. Need to be careful about where R stores the installed packages from additional repos as not to be confused by our R installation done with Intel compilers on the host system. This could lead to very strange weirdness. Perhaps set R_LIBS in the %environment section of the spec file to be independent. Protecting the user may be required.

Singularity: Use Case → RStudio

Where do we take this concept from here?

- We have a large group focused on spatial sciences that wants to evolve their workloads onto HPC-like resources. We'll probably create an environment where we can install a comprehensive set of tools for them in a single image. Packages like qgis, grass, etc.
- Bioinformatics and Genomics workflows where there are too many Python/R/Perl dependencies and specific to keep track of.
- Branch out to others who want to create their own environments.

Singularity: back to the admin stuff!

We've looked at a brief use case. That's great, we can spin up our users with a basic image or highly customized environment. But what else as an administrator should we know about. Take a look at the $PREFIX/etc/singularity.conf file.

The file contains a global set of options set by the administrator that tweak the default Singularity runtime.

Singularity: Admin Stuff

PID Namespace:

- Yes, by default, but may not work well with resource managers and have adverse effect on MPI shared memory. Going under deeper investigation on our end. The Slurm `scancel` functionality seems to work find for us right now.

Bind Paths:

- Local filesystems (we have `/lscratch`) which should be accessible to the users.
- Global filesystems (`/gscratch` and `/project`) necessary for users to hold data.

Don't forget to mount `/tmp` or bind it to an alternative location. Certain applications behave very strangely if `/tmp` doesn't exist.

Singularity: Admin Stuff

Root's $HOME directory within the container:

When tweaking an image environment (like using CPAN, or such), invoking the option -c/--contain is useful to make sure you're not mucking with your host root's $HOME directory.

When you open a container as writable, generally you cannot open a second instance in read-only mode. However, if you have read-only mode already going, you can open in writable mode.

Singularity: Admin Curiosities

We are primarily concentrating on the serial or shared memory parallel applications right now. We've not strayed into additional HPC areas like MPI or GPUs yet.

Anybody worked with Singularity and Lustre or GPFS MPI-IO applications?

NVIDIA users space drivers and verification (see docs)

Mellanox OFED drivers matching the hosts (see docs)

OpenMPI BTLs (see docs)

Singularity: User Curiosities

We use Slurm as the resource manager on almost all our systems.

Using Singularity images on a heterogeneous allocation to run a parallel application, and an I/O server.

Comparisons of bare metal vs different container platforms.

Integration with PGI/Intel compilers. Portable doesn't mean sacrifice capability or performance.

Singularity: Conclusion

1. Singularity is going to provide us with a awesome mechanism to enable our users to bring their environment to our resources
2. Less fear of exploitation of Docker workflows on shared resources. Parts of our filesystem have permissions and ACL's for a certain reason. We don't need new users who figured out shell globbing to destroy our filesystems
3. Seems to be undergoing changes fast at times. Differences between minor versions and documentation is perhaps falling behind a little (i.e., see mount command).
4. Singularity is about providing reproducible environments while also providing adequate security on a shared resource.
5. The ability to leverage Docker images is great.

Containerization Platforms

NCAR's Ben Matthews was also working on a containerized (or similar) framework.

Anybody working with Shifter (NERSC) over Singularity?

Additional comments on Containerization?