# Deploying (community) codes

*Martin Čuma*
*Center for High Performance Computing*
*University of Utah*
*m.cuma@utah.edu*

# Outline

- What codes our users need
- Prerequisites
- Who installs what?
- Community codes
- Commercial codes and licensing
- Building for multiple architectures
- Automatic building
- Application management

# User program needs

- Community programs
  - Free (sort of), written by scientists and engineers
  - Varying level of support and stability
  - There may support on commercial basis
- Commercial programs
  - Sold as a product, have usage restrictions and/or licensing
  - Generally offer support and stability

# What programs are they?

- Community programs
  - Numerical libraries (OpenBLAS, FFTW)
  - Simulation programs (NAMD, NWChem, WRF, OpenFoam)
  - Visualization programs (VisIt, Paraview)
- Commercial programs
  - Numerical libraries (MKL, IMSL)
  - Numerical analysis (Matlab, IDL, Mathematica)
  - Chemistry/material science simulation (Gaussian, Schroedinger)
  - Engineering simulation/CAE (Ansys, Abaqus, COMSOL, StarCCM+)

# Prerequisites

- Supported OS
  - A necessity for binaries (even on Linux)
  - Less strict for builds from source but helpful
- Compilers
  - Most sources build with GNU, may get better performance with commercial compilers (Intel, PGI)
- Software prerequisites (libraries the given code depends on)
  - Additional system packages (e.g. rpms on RedHat/CentOS)
  - Hand built libraries (e.g. MPI, FFTW, …)

# Who installs what?

- Single user system
  - Often have root, install themselves (or use --prefix)
- Multi user system
  - Commonly used programs – user support installs
  - Uncommon or experimental programs – steer users to install themselves
- Special case – Python or R packages
  - Include common packages to the build (numpy, SciPy,…)
  - Instruct users to install themselves and use PYTHONPATH, RLIBS (in ~/.Renviron), etc.

# Installation location

- Local system
  - Some system path (standard /usr/…, /opt) or user's home
- Network file system
  - Applications file system (e.g. NFS) mounted on all servers
  - Need to use --prefix or other during installation
  - No need for root
  - Specific branch for each architecture (x86, power), and potentially OS version (CentOS6, 7)

# COMMUNITY CODES

# Deployment options

- Binaries
  - Many packages supply binaries for the given OS (CentOS), use them, especially if they use graphics
- Build from source
  - several configuration/build systems
    - GNU autoconf (configure/make)
    - CMAKE
    - Scons
  - Need to include dependencies if any

# Source build workflow

- Get the source
  - Ask the researcher, colleagues, or do web search
- Find out how to build it
  - Untar and look for `configure`, cmake files, etc
  - Read the documentation
  - Do web search
  - Beware of configuration options (`configure –help`)
- Decide what compiler and dependencies to use
  - GNU for basic builds, Intel for better optimizations

# COMMERCIAL CODES

# Commercial licensing

- Pay and use w/o license manager (but enforcing license)
  - VASP, Gaussian
- License manager
  - Flexera FlexNet (formerly FlexLM) – used by most
  - Extension to FlexNet (Ansys), other license tool (RLM, own provenience)
- License server setup
  - Best external server, running one license daemon per lmgrd server
  - Good candidate for VM as long as file system traffic is low
- External license servers
  - NAT to access cluster private network
  - Troubleshoot connectivity issues / firewall (`lmutil lmstat`, etc)

# Commercial installation

- Modify makefile and build
  - VASP, Gaussian
- Installers (text or GUI)
  - Mostly straightforward installation
  - Pay attention to where to enter license information
    - Enter license.dat or license server info in the installer
    - Copy license.dat to directory with the program
  - Most FlexNet licenses have environment variable to specify license info, e.g. `MLM_LICENSE_FILE=12345@mylicense.u.edu`
  - If use 3 redundant servers, license must be specified by env. var.

# BUILDING FOR MULTIPLE ARCHITECTURES

# CPU and network considerations

- Most institutions run several generations of CPU and network
  - May have significant performance implications
    - E.g. CPU vectorization instructions can quadruple FLOPS going from SSE4.2 to AVX2 CPUs (3 tic-toc CPU architecture generations)
- What to do about it?
  - Build for lowest common denominator
    - Potentially significant performance implications
  - Build separate optimized executable for each architecture
    - Need to keep track of what executable to run where
  - Build single executable using multi-architecture options

# How can this be done?



- Some compilers allow to build multiple versions of objects (functions) into a single executable
  - Intel calls this “automatic CPU dispatch”
    - Compiler flag –axCORE-AVX2,AVX,SSE4.2
  - PGI calls this “unified binary”
    - Compiler flag --tp=nehalem,sandybridge,haswell
- For multiple network types – use MPI that support multiple network channels
  - Most MPIs these days do – MPICH, OpenMPI, Intel MPI
  - Network interface selected at runtime, usually via environment var.

# Multi-arch executable strategies

- Link with optimized libraries
  - Some vendors (Intel MKL) provide these
  - Build yourself
- Build your application with the appropriate compiler flags/MPIs
- For details see https://www.chpc.utah.edu/documentation/software/single-executable.php

# AUTOMATIC BUILDING

# Deployment strategies

- Occasional builds can be done manually
  - keep old configure files/scripts
- Repetitive builds can be scripted
  - MPIs, file libraries (NetCDF, HDF), FFTW
- Use build automation tools
  - Some localized to a HPC center (Maali, Smithy, HeLMOD)
  - Wider community – EasyBuild, Spack

# EasyBuild and Spack

- EasyBuild seems to be more widely used and flexible
  - Fairly easy to start and deploy if your cluster is “standard” and you don’t care where the builds are stored
  - Customization needs some learning curve, with flexibility comes complexity
  - Implementing over existing stack best done incrementally
  - Good modules support
- Spack seems to be simpler to use but lacks hierarchical module support (LMOD)

# EasyBuild

- Automatic build and installation of (scientific) programs
- Flexible and configurable (build recipes)
- Automatic dependency resolution
- Module file generation, logging, archiving
- Good documentation, increasing community acceptance
- Relatively simple to set up and use when using defaults
- Due to its flexibility, more complicated to customize
- Probably best deployed as a fresh build-out

# EasyBuild basic terminology

The EasyBuild *framework* leverages *easyblocks* to automatically build and install (scientific) software using a particular compiler *toolchain*, as specified by one or more *easyconfig* files.

- EasyBuild framework
  - Core functionality for EasyBuild
- Easyblock
  - Python module, ‘plugin’ into EasyBuild framework
  - implementation of software build and install procedure (generic or specific)
- Easyconfig (`*.eb`)
  - specification file for building/installing a given package version
- Toolchain
  - combination of compiler and additional packages that are needed to build programs (compiler-MPI-numerical libraries)

# EasyBuild basic functionality

- `eb –list-easyblocks` – lists available easyblocks
  - ```
    |-- ConfigureMake
    |   |-- CMakeMake
    |   |-- EB_GROMACS
    ```
- `eb –list-toolchains` – lists available toolchains
  - `goolf: BLACS, FFTW, GCC, OpenBLAS, OpenMPI, ScaLAPACK`
- `eb –S pkgname` – search for package easyconfig
  - ```
    eb -S GROMACS
    * $CFGS1/GROMACS-4.6.5-goolf-1.4.10-hybrid.eb
    ```
- `eb pkgname -r` –install package with dependencies (`-r`)

# EasyBuild basic functionality

- `eb pkgname -Dr` – get an overview of a planned install

```
eb GROMACS-4.6.5-goolf-1.4.10-hybrid.eb –Dr
* [ ] $CFGS/g/GCC/GCC-4.7.2.eb (module: GCC/4.7.2)
* [ ] $CFGS/h/hwloc/hwloc-1.6.2-GCC-4.7.2.eb (module: hwloc/1.6.2-GCC-4.7.2)
* [ ] $CFGS/o/OpenMPI/OpenMPI-1.6.4-GCC-4.7.2.eb (module: OpenMPI/1.6.4-GCC-4.7.2)
* [ ] $CFGS/g/gompi/gompi-1.4.10.eb (module: gompi/1.4.10)
* [ ] $CFGS/o/OpenBLAS/OpenBLAS-0.2.6-gompi-1.4.10-LAPACK-3.4.2.eb (module: OpenBLAS/0.2.6-gompi-1.4.10-LAPACK-3.4.2)
* [ ] $CFGS/f/FFTW/FFTW-3.3.3-gompi-1.4.10.eb (module: FFTW/3.3.3-gompi-1.4.10)
* [ ] $CFGS/s/ScaLAPACK/ScaLAPACK-2.0.2-gompi-1.4.10-OpenBLAS-0.2.6-LAPACK-3.4.2.eb (module: ScaLAPACK/2.0.2-gompi-1.4.10-OpenBLAS-0.2.6-LAPACK-3.4.2)
* [ ] $CFGS/g/goolf/goolf-1.4.10.eb (module: goolf/1.4.10)
* [ ] $CFGS/n/ncurses/ncurses-5.9-goolf-1.4.10.eb (module: ncurses/5.9-goolf-1.4.10)
* [ ] $CFGS/c/CMake/CMake-2.8.12-goolf-1.4.10.eb (module: CMake/2.8.12-goolf-1.4.10)
* [ ] $CFGS/g/GROMACS/GROMACS-4.6.5-goolf-1.4.10-hybrid.eb (module: GROMACS/4.6.5-goolf-1.4.10-hybrid)
```

# APPLICATION MANAGEMENT

# What is application management

- Location of the programs
  - Usually mounted file server
  - Every site has different directory structure
- Presenting programs to the users
  - Shell init scripts
    - Not flexible, need to log out to reset environment
  - Environment modules
  - Other environment management, e.g. Spack

# Programs locations

- Things to keep in mind when designing directory structure
  - Hierarchy/dependence of applications (Compiler – MPI)
  - Source, build and installation preferably in unique location
- Some sites choose hierarchical structure
  - Can lead to deep directory structure with a lot of empty/non-existing directories
- EasyBuild uses mix of hierarchy and name/version
  - Sources stored as name version (letter/name/version)
  - Builds and installs stored hierarchically under toolchain

# CHPC's apps structure

- Separate directories for source, build, installation
  - srcdir, builddir, installdir
  - Only pristine source in srcdir – allows for reuse when building with different compilers, MPIs, configure options, etc
- Subdirectories as package/version
  - E.g. srcdir/mpich/3.2
- Hierarchy denoted with extensions to directory names
  - E.g. built with PGI compilers, installdir/mpich/3.2p
- We generally don't worry too much about compiler/MPI version as they tend to be fairly backwards compatible
  - Exceptions treated via module dependencies and specific directory names

# Environment modules

- Allow user to load and unload program settings
  - TCL based modules part of CentOS distro
  - LMOD from TACC
- LMOD advantages
  - 3 level hierarchy of modules (compiler – MPI – application)
  - Usability enhancements (`ml, +/-, save`)
  - Site customization options
    - E.g. implementation to limit module loading to certain groups (licensees)
  - Companion XALT package tracks module usage

# DEMO

# Source build example

- MIT Photonic Bands (MPB)
  - Program to study photonic crystals
  - http://ab-initio.mit.edu/wiki/index.php/MIT_Photonic_Bands
  - Has a nice set of dependencies (BLAS, LAPACK, MPI, FFTW)
- Download the source
  - `wget http://ab-initio.mit.edu/mpb/mpb-1.5.tar.gz`
- Decide how to build
  - We want to optimize for highest performance – use Intel compilers and libraries (`module load intel impi`)

# Source build example

- Build in `/uufs/chpc.utah.edu/sys/builddir/mpb/1.5i`
- Run `configure –help` to see the options
- Set up configure script – `vi config.line`
  - I prefer to create a script with all the environment variables and configure options
- Run configure script - `./config.line`
- Run `make`
- Run `make install`
- There is no `make test`, so run own
  - `cd test2; ../mpb/mpb-mpi diamond.ctl`

# EasyBuild examples

- `ml use /home/mcuma/temp/easybuild/modules/all`
  `ml EasyBuild`
- `eb –list-easyblocks` – lists available easyblocks
- `eb –list-toolchains` – lists available toolchains
- `eb –S pkgname` – search for package easyconfig
- `eb pkgname -Dr` – get an overview of a planned install

# LMOD demo

- `ml intel mpich2`
- `ml`
- `ml –intel pgi`
- `ml av`
- `ml spider`
- `ml show`