Scientific Software Management in Real Life
Deployment of EasyBuild on a Large Scale System

HUST ’16, November 13, 2016 | D. Alvarez*¹ A. O’Cais¹ M. Geimer¹ K. Hoste² |
¹ Jülich Supercomputing Centre ²HPC-UGent
Scientific Software Management in Real Life
Part I: Introduction
Managing scientific software

- HPC systems typically used by different kind of users.
- Very different software requirements
  - Different compilers
  - Different libraries
    - Different versions of these libraries
  - Different levels of HPC expertise
  - Different tools
- Different time plans
Managing scientific software

- Burden for system administrators and user support teams.
  - May lead to relying on OS packages
    - Can only be updated during a maintenance window
    - Limited to the OS available packages
    - Increased size of OS images

**OS packages examples:**

- Software for general programming
  - Subversion, git, CMake, …

- Software to support components of the scientific software stack
  - X11, additional Python modules, …

- How to deal with different versions?
- How to keep them reasonably up to date?
Scientific software from a user view

- Software often provided via environment modules.
  - Shell-independent way to modify a user’s environment
  - Can be organized in various ways (flat, hierarchical, ...)
    - Though sometimes difficult to implement

- Creating and maintaining consistent module views is tedious and error-prone.

Solution:
Various tools exist to help with software installations and automatic module file creation.
EasyBuild & Lmod @ JURECA

- EasyBuild is a software installation framework
  - http://hpcugent.github.io/easybuild/
  - Already provides lots of useful functionality
  - Compartmentalized structure: framework, easyblocks, easyconfigs
    - Some features we require *were* still missing
- Lmod is a modern environment modules tools
  - https://github.com/TACC/Lmod
  - Powerful support for hierarchy of modules
- How do we use and *extend* these tools to support our users effectively and efficiently?
Scientific Software Management in Real Life
Part II: System details & requirements

HUST ’16, November 13, 2016 | D. Alvarez* A. O’Cais M. Geimer K. Hoste
JURECA system characteristics

- 1.8 + 0.44 PFlops, #57 in Top500 (June’16)
- 1872 compute nodes (Haswell)
- 75 compute nodes with NVIDIA K80 GPUs
- 12 visualization nodes, each with two NVIDIA K40 GPUs
- Mellanox EDR InfiniBand with fat tree topology

Any guess on user requirements?
User requirements

“I want it all, I want it all, I want it all, and I want it now”🎵🎵🎵🎵
User requirements

“I want it all, I want it all, I want it all, and I want it now”

- Intel compilers
User requirements

“I want it all, I want it all, I want it all, and I want it now”

- Intel compilers
- GNU compilers
User requirements

“I want it all, I want it all, I want it all, and I want it now”

- Intel compilers
- GNU compilers
- ParaStationMPI
User requirements

“I want it all, I want it all, I want it all, and I want it now”

- Intel compilers
- GNU compilers
- ParaStationMPI
- IntelMPI
User requirements

“I want it all, I want it all, I want it all, and I want it now”

- Intel compilers
- GNU compilers
- ParaStationMPI
- IntelMPI
- CUDA
User requirements

“I want it all, I want it all, I want it all, and I want it now”

- Intel compilers
- GNU compilers
- ParaStationMPI
- IntelMPI
- CUDA
- CUDA-aware MPI
User requirements

“I want it all, I want it all, I want it all, and I want it now”

- Intel compilers
- GNU compilers
- ParaStationMPI
- IntelMPI
- CUDA
- CUDA-aware MPI
- PGI compilers
User requirements

“I want it all, I want it all, I want it all, and I want it now”

- Intel compilers
- GNU compilers
- ParaStationMPI
- IntelMPI
- CUDA
- CUDA-aware MPI
- PGI compilers
- And of course:
User requirements

“I want it all, I want it all, I want it all, and I want it now”

- Intel compilers
- GNU compilers
- ParaStationMPI
- IntelMPI
- CUDA
- CUDA-aware MPI
- PGI compilers
- And of course:
  - Tons of libraries
User requirements

“I want it all, I want it all, I want it all, and I want it now”

- Intel compilers
- GNU compilers
- ParaStationMPI
- IntelMPI
- CUDA
- CUDA-aware MPI
- PGI compilers
- And of course:
  - Tons of libraries
  - Compatibility
User requirements

“I want it all, I want it all, I want it all, and I want it now”

- Intel compilers
- GNU compilers
- ParaStationMPI
- IntelMPI
- CUDA
- CUDA-aware MPI
- PGI compilers
- And of course:
  - Tons of libraries
  - Compatibility
  - A simple user view
Scientific Software Management in Real Life
Part III: Designing the User View
Designing the User View: Module Hierarchy

- Different ways to present modules to the users:
  - Flat
    - More than 800 packages (compilers × MPI runtimes × software packages) at once
    - Not all visible software is compatible
Designing the User View: Module Hierarchy

- Different ways to present modules to the users:
  - Flat
    - More than 800 packages (compilers × MPI runtimes × software packages) at once
    - Not all visible software is compatible
  - Toolchain based
    - Have to choose particular compiler and MPI combinations before seeing any other package
    - Have to choose between weird or fairly long names (pmvmklc vs PGI_MVAPICH2_MKL_CUDA) for the toolchain modules
    - Visible software is compatible
Designing the User View: Module Hierarchy

- Different ways to present modules to the users:
  - Flat
    - More than 800 packages (compilers × MPI runtimes × software packages) at once
    - Not all visible software is compatible
  - Toolchain based
    - Have to choose particular compiler and MPI combinations before seeing any other package
    - Have to choose between weird or fairly long names (pmvmlkc vs PGI_MVAPICH2_MKL_CUDA) for the toolchain modules
    - Visible software is compatible
  - Hierarchy of compilers and MPI runtimes
    - Modules available are shown in a staged fashion
    - Intuitive
    - Visible software is compatible
Designing the User View: Lmod as modules tool

- Lmod was designed with module hierarchies in mind
  - module spider and module key
  - Module families (family("compiler") or family("mpi"))

- Lmod also has other interesting features
  - Good support for hidden modules (--show-hidden)
  - Cache
  - Properties
Scientific Software Management in Real Life
Part IV: EasyBuild

HUST ’16, November 13, 2016 | D. Alvarez* A. O’Cais M. Geimer K. Hoste
Why EasyBuild in JURECA?

- Because the universal install script doesn’t work reliably
Why EasyBuild in JURECA?

- Because the universal install script doesn’t work reliably

```
#!/bin/bash

pip install "$1" &
easy_install "$1" &
brew install "$1" &
npm install "$1" &
yum install "$1" &
dnf install "$1" &
docker run "$1" &
pkg install "$1" &
apt-get install "$1" &
sudo apt-get install "$1" &
steamcmd +app_update "$1" validate &
git clone https://github.com/"$1"/"$1" &
cd "$1";/configure;make;make install &
curl "$1" | bash &
```

Source: http://xkcd.com/1654/
Why EasyBuild in JURECA?

- Designed exactly for this use case
- Production ready
- Easily configurable
- Nice integration with Lmod and different Module Naming Schemes
- Active and dynamic project
- Support for over 1000 packages
Shortcomings

1. Was based on *monolithic* toolchains
   - Unnecessary redundancy in package builds.
     - E.g., CMake built with many different toolchains

2. Each of the X11 libraries (and other auxiliary libraries) had its own module
   - Swamps default module view with many libraries and their dependencies

3. Software that only compiles with GCC couldn’t be visible in non-GCC toolchains

4. Cryptic toolchain names led to confusion and support issues.
Implemented, user-driven enhancements I

- Enhanced dependency resolution
  - Minimal toolchains
  - Software built with compiler $x$ version $y$ and MPI $z$ version $w$ can use libraries built just with a toolchain containing compiler $x$ version $y$.
  - Toolchain hierarchy: dummy $\Rightarrow$ compiler $\Rightarrow$ MPI $\Rightarrow$ Math libraries

- Common base compiler ($\text{GCCcore}$) for toolchains
  - Enables base layer for compilers, tools and auxiliary libraries
  - Toolchain hierarchy: dummy $\Rightarrow$ $\text{GCCcore}$ $\Rightarrow$ compiler $\Rightarrow$ MPI $\Rightarrow$ Math libraries
Implemented, user-driven enhancements II

- Support for hidden modules
  - Eliminates clutter
  - Supported in various ways (command line options, environment variables, easyconfig parameters)
  - Can hide GCCcore
- Custom module naming schemes
  - Flat
  - Hierarchical
  - Toolchain based
- Naming scheme-independent software installation directories
- Performance improvements
- Refactoring of support for MPICH-based MPI libraries
Scientific Software Management in Real Life
Part V: Current state in JURECA

HUST ’16, November 13, 2016  |  D. Alvarez*  A. O’Cais  M. Geimer  K. Hoste
[Old] current state (Stage 2016a)
User View and Hidden Modules

- Initial user view:
  - Compilers (GCC, Intel, PGI)
  - Binary tools (VTune, Advisor, TotalView, ...)

- After loading a compiler:
  - MPI runtimes (ParaStationMPI, MVAPICH2, IntelMPI)

- Packages built with GCCcore

- Packages compiled with the chosen compiler

- Not all packages available for a given combination are visible:
  - There are almost 400 hidden packages in total!
User View and Hidden Modules

- Initial user view:
  - Compilers (GCC, Intel, PGI)
  - Binary tools (VTune, Advisor, TotalView, . . .)

- After loading a compiler:
  - MPI runtimes (ParaStationMPI, MVAPICH2, IntelMPI)
  - Packages built with GCCcore
  - Packages compiled with the chosen compiler

Not all packages available for a given combination are visible: there are almost 400 hidden packages in total!
User View and Hidden Modules

- Initial user view:
  - Compilers (GCC, Intel, PGI)
  - Binary tools (VTune, Advisor, TotalView, ...)

- After loading a compiler:
  - MPI runtimes (ParaStationMPI, MVAPICH2, IntelMPI)
  - Packages built with GCCcore
  - Packages compiled with the chosen compiler

- After loading an MPI runtime:
  - Packages compiled with the chosen compiler and MPI runtime

Not all packages available for a given combination are visible: There are almost 400 hidden packages in total!
User View and Hidden Modules

- Initial user view:
  - Compilers (GCC, Intel, PGI)
  - Binary tools (VTune, Advisor, TotalView, ...)

- After loading a compiler:
  - MPI runtimes (ParaStationMPI, MVAPICH2, IntelMPI)
  - Packages built with GCCcore
  - Packages compiled with the chosen compiler

- After loading an MPI runtime:
  - Packages compiled with the chosen compiler and MPI runtime

- Not all packages available for a given combination are visible:
  - There are almost 400 hidden packages in total!
Bundling Extensions

- Python, R and Perl have “extensions”
- 1 module per extension is excessive
  ⇒ Bundles
Bundling Extensions

- Python, R and Perl have “extensions”
- 1 module per extension is excessive
  ⇒ Bundles

- Python
  - Python (30 extensions)
  - SciPy-Stack (22 extensions)
  - PyCUDA (6 extensions)
  - numba (2 extensions)
Bundling Extensions

- Python, R and Perl have “extensions”
- 1 module per extension is excessive
  ⇒ Bundles

- Python
  - Python (30 extensions)
  - SciPy-Stack (22 extensions)
  - PyCUDA (6 extensions)
  - numba (2 extensions)

- R (365 extensions)
Bundling Extensions

- Python, R and Perl have “extensions”
- 1 module per extension is excessive
  ⇒ Bundles

- Python
  - Python (30 extensions)
  - SciPy-Stack (22 extensions)
  - PyCUDA (6 extensions)
  - numba (2 extensions)

- R (365 extensions)

- Perl (217 extensions)
Bundling Extensions

- Python, R and Perl have “extensions”
- 1 module per extension is excessive
  ⇒ Bundles

- Python
  - Python (30 extensions)
  - SciPy-Stack (22 extensions)
  - PyCUDA (6 extensions)
  - numba (2 extensions)

- R (365 extensions)
- Perl (217 extensions)
- X.Org (229 extensions)
Finding Software

- 1 important option and 3 commands:
  - module [--show-hidden] available
    - Shows software immediately available
  - module key something
    - Crawls the module tree looking for modules with something on their name
    - Tells what it finds and how to get to it
  - Might need to use spider afterwards to find how to get them
  - Useful for looking for the contents of a bundle (e.g., numpy)
Finding Software

- 1 important option and 3 commands:
  - `module [--show-hidden] available`
    - Shows software immediately available
  - `module [--show-hidden] spider something[/version]`
    - Crawls the module tree looking for modules with `something` on their name
    - Tells what it finds and how to get to it
  - `module something`
    - Crawls the module tree looking for modules with `something` in their description
    - Might need to use `spider` afterwards to find how to get them
    - Useful for looking for the contents of a bundle (e.g., `numpy`)
Finding Software

- 1 important option and 3 commands:
  - `module [--show-hidden] available`
    - Shows software immediately available
  - `module [--show-hidden] spider something[/version]`
    - Crawls the module tree looking for modules with `something` on their name
    - Tells what it finds and how to get to it
  - `module key something`
    - Crawls the module tree looking for modules with `<something>` on their description
    - Tells which modules have been found
    - Might need to use `spider` afterwards to find how to get them
    - Useful for looking for the contents of a bundle (ie: `numpy`)
Upgrading and Retiring Software

- **Stage concept:**
  - Software deployment area for a given timeframe
  - A simple directory
  - Default stage upgraded every 6 months
  - There is a development stage to test software
  - Tested software is added to our *Golden* repository, and then added to the current production stage
  - Close to seamless transitions between stages during maintenance windows
  - Development and old stages are available but not visible by default
Ensuring Consistency and Quality

- **Software team**
  - Allowed to install software in the development stage
  - Can test different compilation options, dependencies, functionality, etc
  - Anybody in the team can modify any other installation

- **Software manager**
  - Only account allowed to install software in the production stages
  - Supervises quality standards on easyconfigs before adding them to the *Golden* repository
    - Correct dependencies for the production stage
    - Proper programming in easyconfigs and patches (lack of hardcoded paths, use of EB provided variables)
  - Manages the whole infrastructure
Divergence from Upstream EasyBuild I

- Divergence motivated by
  - Use of latest versions available at deployment time
  - Re-positioning of packages in the toolchain hierarchy

- Most differences are minimal:
  - Different versions of software
  - Different versions of dependencies
  - Different toolchains

EasyConfigs used in JURECA

<table>
<thead>
<tr>
<th>EB upstream EasyConfigs</th>
<th>47</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSC EasyConfigs</td>
<td>777</td>
</tr>
</tbody>
</table>
Divergence from Upstream EasyBuild II

- Toolchains divergence

<table>
<thead>
<tr>
<th></th>
<th>EB upstream TCs</th>
<th>JSC TCs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp.</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Comp.+MPI</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Comp.+MPI+Math</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
Divergence from Upstream EasyBuild III

- EasyBlocks divergence

<table>
<thead>
<tr>
<th>EasyBlocks used in JURECA.</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>EB upstream EasyBlocks</td>
<td>±65</td>
</tr>
<tr>
<td>JSC tweaked EasyBlocks</td>
<td>5</td>
</tr>
<tr>
<td>JSC merged EasyBlocks</td>
<td>5</td>
</tr>
<tr>
<td>JSC private EasyBlocks</td>
<td>4</td>
</tr>
</tbody>
</table>
Demo
[New] current state (Stage 2016b)
Porting to Other Clusters

- Besides JURECA, JSC also has JUROPA3 and JUAMS
  - Similarities with JURECA: x86_64, InfiniBand, Red Hat based OS
  - Differences: Different microarchitecture, different OSes, mix of Xeon Phi and GPUs
- Minimal changes needed to reuse JURECA’s setup:
  - Fix erroneous easyconfigs
  - Provide new versions in EasyBuild of obsolete OS packages

<table>
<thead>
<tr>
<th>Software in JUAMS and JUROPA3.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total packages in JUAMS</td>
</tr>
<tr>
<td>Total packages in JUROPA</td>
</tr>
<tr>
<td>Ad-hoc packages in both</td>
</tr>
</tbody>
</table>
Scientific Software Management in Real Life
Part VII: Future Work

HUST ’16, November 13, 2016 | D. Alvarez*   A. O’Cais   M. Geimer   K. Hoste
Future

- Automatic upgrades
  - Of dependency versions
  - Of software versions
- Default module sets
  - Preselected packages for users that don’t care about compilers and MPI runtimes
- Linking with \texttt{-rpath} (experimental in EasyBuild 3.0)
- Tracking module usage with XALT
- Reshuffling packages
- “Fat” easyconfigs
Scientific Software Management in Real Life
Part VIII: Conclusions

HUST ’16, November 13, 2016 | D. Alvarez* A. O’Cais M. Geimer K. Hoste
Conclusions

- EasyBuild enables to deploy and manage a tremendous amount of software, using a small team
- Active project that grows everyday
- Effort needed to
  - Minimize SW replication
  - Provide latests and greatest (mismatch between our stage switch and EasyBuild releases)
  - Provide a meaningful user view
- EasyBuild enables easy porting to similar systems
- Still room for improvement
Thank you for listening!
You can meet more EasyBuild folks at:

2\textsuperscript{nd} EasyBuild User Meeting
Jülich Supercomputing Centre (Germany), February 8-10
https://github.com/hpcugent/easybuild/wiki/2nd-EasyBuild-User-Meeting

FOSDEM'17 HPC, Big Data, and Data Science Devroom
Brussels (Belgium), February 4
https://hpc-bigdata-fosdem17.github.io/