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

Damian Alvarez *, *Alan O'Cais* *, *Markus Geimer* *, *Kenneth Hoste* †

*Jülich Supercomputing Centre (JSC), Forschungszentrum Jülich GmbH, 52425 Jülich, Germany
Email: {d.alvarez, a.ocais, m.geimer}@fz-juelich.de

†HPC-UGent, DICT, Ghent University, Krijgslaan 281, S9, 9000 Gent, Belgium
Email: kenneth.hoste@ugent.be

***Abstract*—Managing scientific software stacks has traditionally been a manual task that required a sizeable team with knowledge about the specifics of building each application. Keeping the software stack up to date also caused a significant overhead for system administrators as well as support teams. Furthermore, a flat module view and the manual creation of modules by different members of the teams can end up providing a confusing view of the installed software to end users. In addition, on many HPC clusters the OS images have to include auxiliary packages to support components of the scientific software stack, potentially bloating the images of the cluster nodes and restricting the installation of new software to a designated maintenance window.**

To alleviate this situation, tools like EasyBuild help to manage a large number of scientific software packages in a structured way, decoupling the scientific stack from the OS-provided software and lowering the overall overhead of managing a complex HPC software infrastructure. However, the relative novelty of these tools and the variety of requirements from both users and HPC sites means that such frameworks still have to evolve and adapt to different environments. In this paper, we report on how we deployed EasyBuild in a cluster with 45K+ cores (JURECA). In particular, we discuss which features were missing in order to meet our requirements, how we implemented them, how the installation, upgrade, and retirement of software is managed, and how this approach is reused for other internal systems. Finally, we outline some enhancements we would like to see implemented in our setup and in EasyBuild in the future.

I. Introduction

The large and usually diverse user communities of HPC systems typically require a significant number of scientific software packages to be installed and maintained. This places a major burden on system administrators and user support teams, who are responsible for managing the software stacks on the HPC systems throughout their lifetime. This includes tasks such as installing requested packages, keeping these installations up to date, as well as retiring old (obsolete) versions of a software package without upsetting users.

Traditionally, the management of scientific software stacks has been a manual task. As such, the size of the team supporting it grew with the number of packages, and its members had to know the specifics of how to properly build the supported software. Due to limited human resources, and in order to focus on the scientific software packages, the OS images on many HPC clusters therefore often included a non-negligible number of auxiliary packages, either for general programming (e.g., Subversion, git, CMake, etc.) or to support components of the scientific software stack (e.g., visualization libraries, additional Python modules,...). This approach not only bloats the OS images of the cluster nodes, but also restricts the installation of missing software to a designated maintenance window. In addition, it usually falls short in the long run as newer versions of such packages may still be desired during the lifetime of the system.

Another important aspect of managing a scientific software stack is the manner in which the available software on a system is exposed to its users. In this context, many HPC sites use environment modules [1], which allow users to easily load, unload, and switch between software packages by modifying the user's environment. Traditionally, environment modules have been organized in a "flat" way, which frequently led to an unstructured and confusing module view. More sophisticated module views, like organizing environment modules in a hierarchical fashion [2], can be challenging to implement. In any case, creating consistent module files and maintaining them manually is tedious and error-prone.

Fortunately, a number of tools have emerged in recent years to assist system administrators in managing scientific software stacks, including both the installation of software packages as well as creating associated module files automatically. The lack of some features in these tools is highlighted as they are adopted by different HPC sites. Thus, despite their various advantages and the features they already provide, they have to evolve and adapt to new environments and constraints.

In this paper, we report on our experiences with deploying *EasyBuild* in conjunction with the *Lmod* modules tool [3], [4] in a large-scale cluster environment: the JURECA cluster installed at Jülich Supercomputing Centre (JSC). In particular, we discuss:

- Several features that were missing in EasyBuild to meet our requirements, and that we have subsequently implemented and contributed back to the community.
- How the installation, upgrade, and retirement of software is managed.
- How the approach presented is reused for other internal systems and which obstacles were encountered.

- How we would like our setup and EasyBuild to evolve, to make for a better user and admin experience.

The paper is organized as follows: in Section II, we first outline the requirements on the software stack imposed by both the architecture of the JURECA system as well as our user community. Next, in Section III we discuss how to present software to the users, and why we chose Lmod for it. Section IV highlights why we have chosen EasyBuild, which of the required functionality was already provided when we started deploying it, and how we improved it when needed. In Section V we provide details on the current state on JURECA, the user view and divergence from upstream. This section also discusses our software lifecycle strategy. Section VI shows an overview of how we reuse this software management approach on other internal cluster systems. Finally, we outline planned future enhancements in Section VII and conclude in Section VIII.

II. System Details and Requirements

As of June 2016, the JURECA cluster [5], [6] installed at the JSC is ranked number 57 in the Top500 list and consists of over 1,800 compute nodes with a peak performance of 1.8 (CPU) + 0.44 (GPU) petaflop/s. Each compute node features two 12-core Intel Xeon E5-2680v3 CPUs (Haswell-EP). 75 nodes are equipped with two NVIDIA K80 GPUs (four visible devices per node). Standard compute nodes are equipped with 128 GiB of DDR4 main memory. For memory-intensive applications, there are 128 nodes with 256 GiB and 64 nodes with 512 GiB of RAM. In addition, several dedicated visualization nodes with large memory configurations and two NVIDIA K40 GPUs per node complement the JURECA configuration and enhance the pre- and post-processing capabilities available to the users. All nodes are connected by an EDR InfiniBand network using a fat-tree topology.

The heterogeneous nature of the JURECA system also requires a complex software environment. The x86 CPU architecture dictates that both the Intel and GCC compilers have to be supported, while the GPU accelerators require providing CUDA as well as the PGI compilers—for the latest OpenACC standard—to support the most commonly used programming models.

In addition to the various compilers, a number of different MPI implementations are supported on the platform. While ParaStationMPI [7], [8] is the preferred MPI implementation, it is not yet CUDA-aware. Therefore, MVAPICH2-GDR [9], [10] is also supported to allow efficient communication in GPU-accelerated applications. Finally, Intel MPI is also provided as an alternative for users that might prefer it.

To ensure a consistent software stack and to avoid subtle, hard-to-diagnose errors, one can define *toolchains* consisting of a particular compiler coupled with a specific MPI implementation, and require that each software package is available within each toolchain. In the use case outlined above, this requires 9 builds, one for each combination of our three compilers and three MPI implementations.

One can easily see how this quickly leads to a very large number of environment modules which gets increasingly cluttered as new versions of software packages become available. This can make it difficult for users to find the most appropriate, and most up-to-date, version of the software they require. In the remainder of this paper, we will discuss how these aspects were dealt with.

III. Designing the User View

An intuitive perspective for end users of the scientific software we provide is our primary goal. Providing consistent software stacks (in terms of the compiler and MPI implementation used) for our supported software leads to module explosion. Through the use of a *module hierarchy* one can restrict the user view to purely the consistent stack. Many module tools do not provide the features to search for software within such a scheme, which lead us to our use of *Lmod*. In the subsections below we discuss our chosen user view and the Lmod features to support it.

A. Module Hierarchy

In order to manage the flood of modules, we have chosen to use a *module hierarchy* where a user must first select a compiler and then an MPI implementation in order to have all software built with this compiler/MPI combination visible in their view of the available environment modules.

This approach has a number of advantages compared to the traditional 'flat' organization of modules:

- Only modules that are compatible with each other are available for loading; this approach is an efficient way of specifying which modules can be loaded together, so that users do not run into (often subtle and hard to diagnose) problems by combining different incompatible software packages.
- The overview of available modules is significantly reduced, since only a subset of all existing modules is shown when users query which modules are available for loading via `module avail`; this helps in providing an easy-to-digest overview of modules to users.
- There is a clear separation between software that provides MPI functionality, and software that does not; this information can be useful to determine in which context the software can be used.

However, supporting a module hierarchy requires some very particular features in the modules tool. These features are discussed below.

B. Lmod as Modules Tool

Although the traditional Tcl-based environment modules tool in theory also supports modules organized in a hierarchical layout, a number of important features are lacking to provide users with a satisfying interface to interact with the provided modules.

Lmod is a recent Lua-based modules tool that provides a number of additional features making it an interesting alternative. As such, it was chosen for a number of reasons:

- Lmod provides a cache of existing modules, greatly improving responsiveness for systems with a large number of modules.
- Lmod provides the `module spider` command, which searches the entire set of existing modules, not just those available for loading in the current context of loaded modules in a hierarchical layout.
- Lmod has better support for hidden module files; modules that are expected to only be loaded as dependencies can be hidden from the default module view, but are still searchable with the use of a specific flag (`--show-hidden`) to the `module avail` command.
- Lmod has support for defining *module families*. For example, if each of the compiler module files define `family("compiler")` then Lmod will only allow one such module to be loaded at any one time.

These features give Lmod excellent support for module hierarchies; in fact, it was developed from the ground up with module hierarchies in mind. Next to these features additional useful options are available such as the shorthand command `ml`, support for user-defined hooks to customize various `module` subcommands, assigning properties to modules, etc. Moreover, the development of the traditional Tcl/C environment modules implementation has stalled and Lmod is an actively developed drop-in replacement.

IV. EasyBuild for Installing Scientific Software

Orthogonal to the modules tool, several tools that facilitate the installation of (scientific) software on HPC systems have emerged in recent years. These aim to reduce the burden on system administrators and support teams of maintaining a stack of scientific software that meets the demand and expectations of the users.

EasyBuild can install the software stack and generate the appropriate hierarchical module scheme. In this section we motivate our choice for EasyBuild, and outline the previously available features and those features that were added during the development of the JURECA software stack.

A. Motivation

In recent years, various tools that automate the process of installing scientific software have been made available to the HPC community, including *SWTools* [11], [12], *Smithy* [13], *maali* [14], [15], *Spack* [16], [17], and EasyBuild [18], [19].

We have chosen EasyBuild in our setup, for a number of reasons:

- EasyBuild is intended exactly for our use case, i.e., efficiently maintaining a software stack for end users on a modern HPC system. In contrast, Spack for example is more targeted towards developers of large scientific software applications, as illustrated by several of its main features including a very flexible and powerful way of dealing with dependencies.
- EasyBuild is considered to be stable and ready for production usage for a number of years (since 2012), and is used by HPC sites worldwide to manage scientific software stacks on production HPC systems. Most other tools are either considered to be in a pre-production/alpha stage (e.g., Spack[1]), or their implementation is too site-specific to be deployed easily at sites other than where they were developed (e.g., maali, Smithy).
- EasyBuild can be configured according to site policies, ranging from the software installation prefix to all aspects of the module naming scheme being used for the modules being generated. To the best of our knowledge, none of the other tools provide this level of flexibility.
- EasyBuild includes extensive support for both Lmod and module hierarchies, including generating module files in Lua syntax, as opposed to the other tools.
- EasyBuild is actively maintained, and backed by a supportive and welcoming community. Some other projects have not been updated frequently recently (e.g., SWTools, Smithy), and/or have not been adopted by other HPC sites yet and thus lack a community around them (e.g., Smithy, maali); the notable exception here being Spack.
- EasyBuild includes recipes ('*easyconfig files*') for over 1000 different software packages, which is significantly more than any of the other tools.

Together, the above aspects form a convincing argument for selecting EasyBuild as the tool to employ in this context. However, there were a couple of shortcomings that motivated the implementation of new features and enhancements, which are discussed in the following sections.

B. Shortcomings

The complex use cases and large community of JURECA quickly required hundreds of software packages to be installed with each of the employed toolchains. A number of issues rapidly became apparent:

- There was unnecessary redundancy in providing tools like CMake using 9 toolchains.
- Supporting visualization software requires the integration of X11 libraries within our EasyBuild installation, swamping the scientific applications in the default module view with these libraries and their dependencies.
- Some visualization libraries do not build with anything other than the GCC compiler, making the visualization libraries unavailable in toolchains that include other compilers.
- Exposing the cryptic names of the employed toolchains led to confusion and support issues.

These issues led to a number of requirements from our site in order to maintain a coherent user view restricted only to relevant scientific and development packages. The result is a number of enhancements that were implemented in EasyBuild, which we discuss below.

[1]Quote from the Spack wiki at https://github.com/LLNL/spack/wiki\#status: "Spack is currently alpha software. It will remain so until it hits v1.0."

C. User-Driven Enhancements

Even though EasyBuild already provided lots of functionality that aligned well with our requirements, we have implemented several enhancements since it was initially adopted at JSC. Most of these are related to improving the user's interface to the installed software, i.e., the provided modules that are generated automatically by EasyBuild. These enhancements were contributed back to the central code repository of EasyBuild, and were quickly embraced by the EasyBuild community as useful features.

The support for maintaining a module hierarchy through EasyBuild allowed us to organize the module stack better than before, in a fully automated way. However, over time several shortcomings were exposed that motivated further enhancements and additional features to be implemented. The subsections below discuss the changes that we have implemented in EasyBuild for an improved user-experience.

*1) **Support for Hiding Modules**:* The first of these requirements was the ability to have EasyBuild support the creation of hidden modules for some dependencies.

A hidden module essentially can be created by prefixing the module file name with a dot ('`.`'), just like any other hidden file in a Unix-based operating system. This makes the module file invisible in the default `module avail` output, thus tidying up the list of modules exposed to the user. Such hidden modules, however, can be loaded as usual and are thus ideal for packages that are only used as dependencies of other packages.

The support for hidden modules implemented in EasyBuild allows to specify hidden modules in four different ways:

- Using the command line '`--hidden`' option. This allows to hide a particular package installation.
- The '`hidden`' easyconfig parameter. This can be set in easyconfig files to consistently generate a hidden module for a particular software.
- The '`EASYBUILD_HIDE_DEPS`' environment variable. This variable contains a list of dependencies to hide by default when installing the needed dependencies.
- The '`EASYBUILD_HIDE_TOOLCHAINS`' environment variable. This variable contains a list of toolchains to hide. It is useful to avoid exposing the `GCCcore` layer underneath the compilers layer.

*2) **Common Base Compiler For Toolchains**:* The second issue was that visualization packages in particular necessitate having an underlying GCC compiler which can build and provide these packages across all toolchains. On a typical distribution this is the role of the system compiler (and the package manager). One should note however that the binutils versions distributed with some OSes do not support the instruction sets of the latest CPUs, meaning that distributed packages may be far from optimized for the modern architectures found in typical HPC systems. For this reason, the EasyBuild community wish to maintain control over the entire compilation environment and compilation options. This lead to the development of the `GCCcore` concept within EasyBuild, which essentially acts as a replacement for the system GCC compiler and also allows us to exercise control over which binutils installation is used for all software installations.

In addition, the Intel and PGI C/C++ compilers rely on a sufficiently recent version of GCC (e.g., for proper C++11 support). Through `GCCcore`, this requirement can be fullfilled for several toolchains at once.

Finally, `GCCcore` provides an additional layer underneath all compilers (and therefore all toolchains) in which one can provide universal software packages (such as CMake, git,...), see Figure 1.

*3) **Enhanced Dependency Resolution**:* With `GCCcore` in place, another required enhancement of EasyBuild quickly came to light. The dependency resolution mechanism implemented by EasyBuild was rather straightforward: usually, the toolchain that was employed to build and install a particular software package was simply inherited by the dependencies as well, even though they may not require some of the functionality that was provided by that toolchain (for example MPI support, BLAS/LAPACK libraries, etc.). Although this is of little concern when a flat module naming scheme is used, it has important side effects in a module hierarchy since initially only a small subset of modules are made available for loading.

Although EasyBuild supported specifying a custom toolchain for each dependency, this was tedious and introduced a significant potential for errors since one needs to manually ensure that the specified toolchain (e.g., `GCCcore`) is compatible with the toolchain of the parent software. In addition, this hardcoding of subtoolchains conflicted with other useful features like the `--try-toolchain` command line option to easily install a particular software stack with a different toolchain.

To alleviate this, the dependency resolution mechanism was enhanced to also take into account the entire hierarchy of so-called *subtoolchains*, where the compiler-only and compiler+MPI-only subsets of a 'full' toolchain (also providing BLAS/LAPACK/FFTW functionality) are considered first for dependencies, rather than directly inheriting the toolchain from the 'parent' software. Through this mechanism, the straightforward specification of dependencies (without an explicit toolchain) can be retained, while the potential for human error is removed.

This concept of *minimal toolchains* opens the door to a much more intuitive module hierarchy: global tools and libraries can be moved to `GCCcore`, software that requires a specific compiler (typically for performance reasons, such as Python) can sit at the `Compiler` level of the hierarchy, with all other applications sitting at the `MPI` level of the hierarchy. This also dramatically simplifies the software environment from a maintenance perspective. Once software is built with the toolchain at the appropriate level, the number of builds with different toolchains is minimized.

This feature was implemented to be enabled optionally via the `--minimal-toolchains` configuration setting, but is likely to become the default behavior in the near future. Together with this, an effort was made to motivate the community

to use the appropriate toolchains according to the requirements of the software package that is being installed.

Ultimately these improvements mean that even though over 800 software installations are currently supported on JURECA (with about 300 at the level of `GCCcore`), only about 150 are ever visible to an end user. The rest are either hidden, or in another branch of the hierarchy.

*4) **Custom Module Naming Scheme**:* Initially EasyBuild was developed with a flat naming scheme in mind. With the introduction of hierarchical schemes, the community realized that each site may have particular requirements for their own module naming scheme, leading us to introduce a mechanism to allow sites to customize their module view. By default EasyBuild includes both the flat and hierarchical schemes and these can be leveraged as examples for custom schemes. JURECA employs such a custom scheme (based closely on the hierarchical scheme) to control the exact structure of the hierarchy and the naming of some specific modules (such as the compilers).

*5) **Installation Directories Independent of Module Naming Scheme**:* With the introduction of the custom module naming scheme, there was also a realization that the installation directories of software should be unique and independent of the naming scheme employed. This feature is now also supported within EasyBuild and, with the addition of the `--module-only` option, allows the support of multiple parallel naming schemes for the same software stack. This is potentially of use if one, for example, wishes to deliver subsets of the software stack to a particular user community.

*6) **Performance Improvements**:* A tool like EasyBuild has to be reasonably fast to be usable. It is common to perform dry runs to verify what will be installed, and that correct dependencies are being picked up. However, as features were added, its speed diminished noticeably in certain scenarios. In some cases, when using a hierarchical module naming scheme (HMNS) and software packages with a complex dependency tree, the time spent on dependency resolution was several minutes. The reasons for this slowdown were twofold:

1) Multiple instantiations of `ModulesTool`, a class that used to be *singleton* before multiple instances were required to support installing packages from multiple toolchains in HMNS setups.
2) Numerous calls to `module show` and `module avail`, which were particularly expensive when using minimal toolchains.

Fixing these two issues, by minimizing the number of `ModulesTool` instances to the number of easyconfig files plus one, and by caching the output for `module show` and `module avail`, resulted in a speedup of up to 10x.

*7) **Refactoring of Support for MPICH-based MPI Libraries**:* Another small but significant change was the refactoring of support of MPICH derivatives. MPICH [20] is one of the most successful MPI implementations, with multiple commercial (Intel MPI, Cray MPI, IBM MPI, etc.) and free and open source (ParaStation MPI [7], [8] and MVAPICH [9] among others) derivatives.

This refactoring was motivated by one particular trait of the MPICH compilation process. Normally, `configure && make` based software use `$CFLAGS`, `$LDFLAGS` and related environment variables to determine the compiler and linker flags to be used during the compilation process. However, during the configuration step, MPICH takes these flags and transparently adds them to its MPI compiler wrappers [21]. Since EasyBuild sets these variables, this is not the desired behavior at installation time, and forces a change in the internal behavior of EasyBuild to unset these flags and export another set of variables to achieve the intended behavior. Such customizations are done in *easyblocks* within EasyBuild which are specific to a particular software package. Since this trait is inherited by other open source MPICH-based MPI implementations, a new family of easyblocks was created with `EB_MPICH` as a base and derived easyblocks like `EB_MVAPICH2` and `EB_psmpi` (for ParaStation MPI).

V. Current State in JURECA

Section II described a set of requirements gathered from the users and the support group, based on the experiences gathered during the first few months of operation of JURECA. Together with the capabilities of EasyBuild at that time, described in Section IV-A, we have designed and implemented a set of features and enhancements, described in Section IV-C, that have been incorporated upstream. In this section we describe how our current setup looks like, as a result of these requirements, capabilities, and improvements.

A. Toolchains

Currently, in the so-called 2016a *stage* (see Subsection V-E), there are a total of 15 unique toolchain definitions, which reflect multiple combinations of compilers (GCC, Intel and PGI), MPI runtimes (ParaStationMPI, Intel MPI and MVAPICH2) and mathematical libraries (MKL and OLF[2]). Software maps naturally to any of these three levels. As previously mentioned, the Intel and PGI C/C++ compilers rely on GCC. This is in most cases provided by the OS.

However, to decouple our setup from the OS packages, we have two isolation layers: the `dummy` toolchain, where software is installed right on top of the OS, and the `GCCcore` toolchain (see Subsection IV-C), which provides an EasyBuild-controlled GCC, required by the compilers.

The requirements outlined in Section II imposed a series of restrictions on the installation structure, due to version incompatibility. The main ones are:

- The installed versions of Intel, PGI and CUDA compilers do not support GCC 5 underneath. This forces us to have a `GCCcore` for version 4.9.3 and another one for version 5.3.0, if we want to provide the latest GCC to our users.
- CUDA 7.5 does not support Intel 2016 compilers. That creates the need for another compiler-level toolchain, as Intel 2016.2 cannot be used together with CUDA.

[2]OLF: OpenBLAS, LAPACK, ScaLAPACK and FFTW

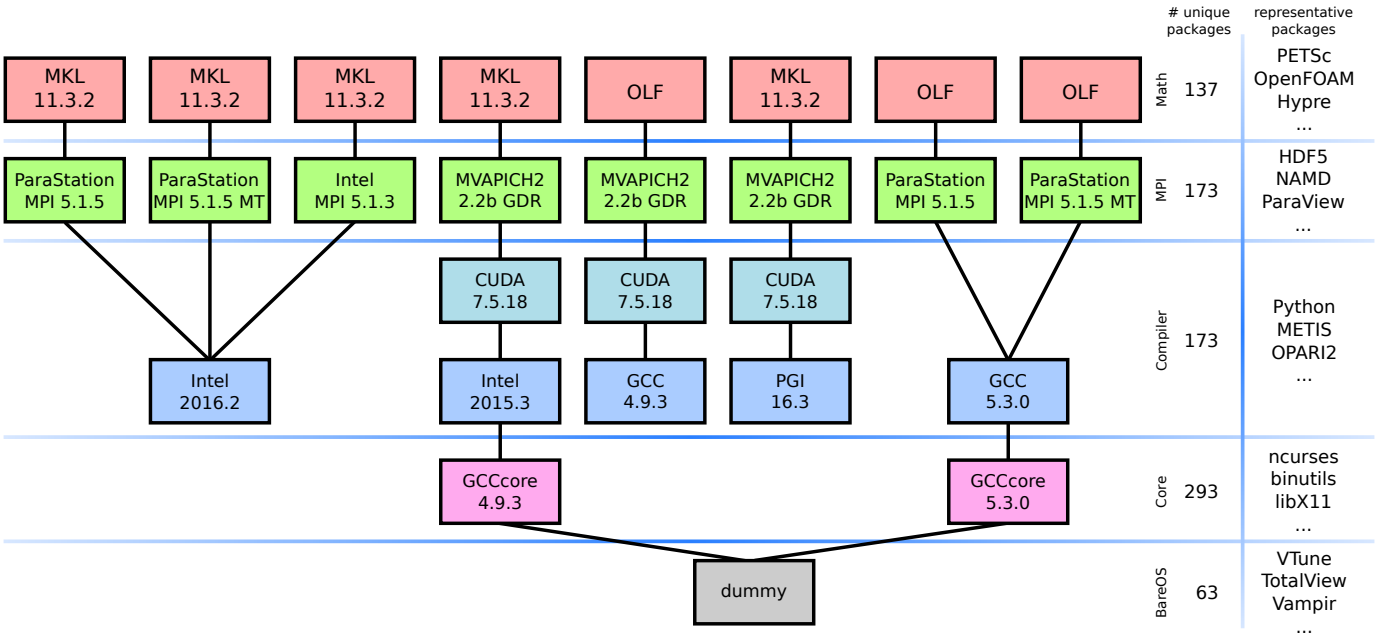

Fig. 1. Toolchain tree in JURECA, in the 2016a *stage*. The total number of unique packages in the toolchain hierarchy is provided on the right.

The above incompatibilities, together with the described user requirements (OpenACC support with PGI, an MPI runtime that is CUDA-aware, Intel MPI for users not familiar with ParaStationMPI, and latest versions of GCC and Intel compilers), forces a toolchain tree of significant size and complexity. This tree can be observed in Figure 1.

B. Hidden Modules and User View

Due to the notable number of compiler and MPI runtimes combinations, the user gets a very basic view on login. There are two main groups of visible modules: (1) Compilers and (2) binary tools installed in `dummy`. Other tools, installed at the `GCCcore` level, are not initially reachable, since the particular version of this level depends on the compiler choice. Therefore the first task of the user is typically choosing a compiler and version. The version defaults to the newest one if it is not explicitly selected. In our hierarchical module naming scheme, this results in an extended `$MODULEPATH` (an environment variable that holds the directories where Lmod will look for modules), that enables the loading of additional modules. The new set of modules are divided in three groups: modules belonging to `GCCcore`, modules belonging to the compiler selected, and a list of MPI runtimes available for that particular compiler. In this way, we expand the user view step by step, as they climb up the toolchain tree. Finally, loading the desired MPI runtime expands the `$MODULEPATH` once more, showing all the software installed at the MPI *and* the mathematical libraries levels. This is possible because there are no multiple branches per MPI runtime at the top of the tree.

With the selection of compiler and MPI a user has all the software available for that part of the tree. That means that the amount of software to be displayed is considerable. The total number of packages for the Intel compiler and ParaStationMPI is 372. This is a sizable number that clutters the output and contributes to user confusion. For this reason, we make extensive use of hidden packages. The vast majority of these hidden packages are helper libraries and tools installed at the `GCCcore` level. Using this feature the number of packages visible by default is reduced to 172.

C. Bundling Extensions

Some popular scripting languages are highly modular. Often, users need multiple packages for these languages; Python, Perl and R are notable examples. These three languages share a common idea: they all have central repositories that hold large numbers of packages for them, which are easily accessible and installable. As a side effect of this relative simplicity, it is common to have very diverse requirements from users using these environments.

During the first months of operation of JURECA we could gauge the user needs. Typically, requests for new language packages resulted in new module files, which cluttered the user view. To keep things more convenient for the users we decided to make extensive use of bundles. This way, in a single module, we could aggregate as many language packages as necessary. In this context, language packages are called *extensions*. That necessitates relatively large easyconfig files, where all the needed extensions are included. Thanks to this bundling, we have included a large number of extensions in a few modules, grouped by functionality.

Python packages are a prime example. Currently, there are two Python modules at the compiler level, one for version 2.7.11 with 30 extensions, and another one for 3.5.1 with 24 extensions. In HPC, `numpy` and `scipy` are Python packages which are very commonly used. Together with `matplotlib`, `iPython` and other packages, they conform the so-called

SciPy stack. We provide them at the top level toolchains, since they require mathematical libraries. They are bundled together with their dependencies, for a total of 22 packages, in the `SciPy-Stack` module. Other modules following the same idea are `PyCUDA`, with 6 extensions, and `numba`, with two.

Perl and R are arguably less used in HPC, and JURECA's users of these languages have more homogeneous needs. For this reason, there is a single Perl and a single R module. Perl is normally not performance critical, and it sits therefore in `GCCcore`, with 217 extensions. R, on the other hand, needs a significant number of libraries, like MPI, HDF5 and BLAS. Therefore, the R modules sit at the top of the toolchain hierarchy, with 365 extensions. Of those, 12 where added specifically for JSC's users, whereas the others were already present in the EasyBuild repository.

D. Finding Software

The use of a hierarchical module naming scheme, hidden modules and bundles sometimes causes *false negatives*, when an unfamiliar user tries to find software in JURECA. `module avail` shows little information by default, and the user has to use alternative commands to find software. This subsection describes how to find software in JURECA.

The first important command is `module avail`. This command is the primary means to know which modules are immediately available for loading. However, it doesn't show hidden modules. Lmod has the `--show-hidden` option to overcome this.

A secondary command is `module spider`. With it, Lmod crawls the module tree, and displays three different kinds of information:

- A description of the modules that matches the query.
- Which versions are installed, if no particular version is queried.
- Which particular modules enable the loading of the desired version, when a version is queried.

This is illustrated in the example below:

```
$ module spider software
----------------------------------------
  SOFTWARE:
----------------------------------------
  Description:
    Some description

   Versions:
      software/0.1

----------------------------------------
  For detailed information about a
  specific "software" module (including
  how to load the modules) use the
  module's full name.
  For example:

     $ module spider software/0.1
----------------------------------------

$ module spider software/0.1
----------------------------------------
  SOFTWARE:
----------------------------------------
  Description:
    Some description

  You will need to load all module(s) on
  any one of the lines below before the
  "software/0.1" module is available to
  load.

Intel/2016.2.181 IntelMPI/5.1.3.181
Intel/2016.2.181 ParaStationMPI/5.1.5-1
```

The `spider` command also supports `--show-hidden`, so users can look for hidden modules anywhere in the hierarchy.

Lastly, the third command helps users to find extensions in bundles. `module key` looks for keywords in the description of the modules, as exemplified below:

```
$ module key numpy

----------------------------------------
The following modules match your search
criteria: "numpy"
----------------------------------------

 SciPy-Stack: SciPy-Stack/2016a-Python...
   SciPy Stack is a collection of open
   source software for scientific
   computing in Python. It contains NumPy,
   SciPy, matplotlib, pandas, SymPy,
   IPython and nose. - Homepage:
   http://www.scipy.org
```

It should be noted though, that in order to display the correct extensions contained in a bundle, they have to be manually listed in the module description in the easyconfig file.

E. Upgrading and Retiring Software Installations

The expected lifetime of JURECA is roughly five years. Within that period one can expect updates to compilers every few months and updates to MPI implementations as the latest standards are integrated. This would mean that the entire software stack will require frequent upgrades. During such upgrades it is natural to expect that one would install the latest version of any particular software package.

The project cycle for JURECA lasts 12 months with two cycles per year. When new users get access to the machine, we want them to only be exposed to the latest software with the latest compilers. For this reason, we have chosen six months as our upgrade period and we chose to retire outdated software versions with the same frequency. We call these software upgrades *stages*. For each *stage*, we select the toolchains that we will support and rebuild the latest versions of our supported software with these toolchains. We chose a prototype toolchain as a template and, once fully populated, migrate the changes to our other toolchains.

We expect members of the support team to contribute to software installations since it is common that application software requires specific knowledge to be installed and tested appropriately. We provide a special development *stage* with the latest toolchains for the support team where they can prepare

their easyconfig files for inclusion in the upgrade. Once a software package has been successfully built and tested, it is added to a *Golden* repository to be used for the *stage* upgrade.

The default *stage* visible to users is controlled by a symbolic link. *Stage* upgrades are prepared in a separate environment to this default. Once the upgrade has been implemented, users are given three weeks notice and the symbolic link is updated during a maintenance window. Users are provided with the capability of continuing to use a retired *stage* if they wish to do so. However, additional software requests can only be made for the current default *stage*.

While *stage* upgrades may introduce some overhead for existing users (they may need to recompile their code and modules may be named differently in particular cases), there are clear benefits to using the latest compilers and software stack. In addition, these upgrades provide us with the opportunity to potentially change our module hierarchy or introduce new features related to Lmod.

F. Ensuring Consistency and Quality

As mentioned in the previous subsection, specific members of the support team should contribute to installing new software. However, they are not expected to have an in-depth knowledge of EasyBuild or of the precise details of our internal configuration. To ensure consistency and quality standards, just a very reduced number of people (typically 1-2) is allowed to install software in the *production stage*, using a dedicated account.

Before any software is installed in the *production stage*, it is evaluated in the *development stage*. It is there where all the members of the support team are allowed to install software. This *stage* is hidden from the default user view. Once software is successfully evaluated there, the easyconfigs are sanitized before being stored in the *golden* repository and the software is then installed in the *production stage*.

To facilitate installations through EasyBuild, we have developed a custom module that correctly sets the needed environment variables: installation prefix; build paths; paths for the dependency resolution; list of hidden dependencies; source paths; default use of minimal toolchains; location of custom easyblocks, toolchains and module naming schemes; modules tool and syntax; backend job configuration and finally permissions for software maintainers. The permissions are set in a way that guarantees that all the software installed belongs to the same group, with `EASYBUILD_SET_GID_BIT=1`, and that no other users have write permissions, with `EASYBUILD_UMASK=002`. Additionally, in the *production stage* this module sets `EASYBUILD_STICKY_BIT=1`, whereas in the *development stage* `EASYBUILD_GROUP_WRITABLE_INSTALLDIR` is set to `1`, guaranteeing that nobody besides the designated people can modify the *production stage*, and that everybody in the support team can modify software in the *development stage* if necessary, regardless of ownership.

G. Divergence from Upstream EasyBuild

The deployment of the setup used in JURECA is not straightforward, as most of the easyconfig files currently used there are not included upstream. There are two reasons for this:

1) We provided the latest available software at deployment time.
2) We switch lots of packages from top-level toolchains to their correct placement in the hierarchy due to our default use of minimal toolchains.

Whereas this is a tedious task that requires lots of manual checking, the truth is that the differences between the easyconfigs installed in JURECA and the easyconfigs upstream are typically quite small. Most of these differences are simply the toolchain, toolchain version, or software version. Version changes can be automated up to a certain extend, as discussed in Section VII. However, toolchain changes require manual tweaking, as the heuristics to determine the optimal package placement (base, compiler, MPI or top levels) are not straightforward. Table I shows the number of files tweaked for JURECA's setup, that highlights a significant effort in this tweaking process.

TABLE I
EASYCONFIGS USED IN JURECA. MOST OF JSC'S EASYCONFIGS DIVERGE JUST DUE TO SMALL VERSION TWEAKS.

EB upstream EasyConfigs	47
JSC EasyConfigs	777

Toolchain definitions describe, among other things, how different options are specified in each toolchain and how the different toolchain components are related. They are, therefore, part of the EasyBuild framework. For JURECA all the compiler toolchains were already there. However, the toolchains that included MVAPICH2, had to be defined. Similarly, all the toolchains built on top of those needed to be defined as well. This is summarized in Table II.

TABLE II
TOOLCHAINS USED IN JURECA.

	EB upstream toolchains	**JSC toolchains**
Comp.	3	0
Comp.+MPI	3	3
Comp.+MPI+Math	3	3

Regarding easyblocks, the vast majority of them were already present in EasyBuild. However, some required tweaking to work with new software. Additionally, JSC developed additional easyblocks, some of which have already made it upstream. This is summarized in Table III.

Lastly, we have adapted the hierarchical module naming scheme to fit our needs, as already described in Section IV-C4.

VI. PORTING TO OTHER CLUSTERS

JURECA is the largest cluster at JSC, and where most of our efforts are focused. However, we mimic the same setup

TABLE III
EASYBLOCKS USED IN JURECA.

EB upstream EasyBlocks	±65
JSC tweaked EasyBlocks	5
JSC merged EasyBlocks	5
JSC private EasyBlocks	4

in other internal clusters: JUROPA3, a cluster with 672 cores, and JUAMS, a cluster with 1960 cores. This section describes the issues that arose when porting our setup from JURECA to JUROPA3 and JUAMS.

JUROPA3 and JUAMS were originally testbeds for JURECA admins and users, prior to JURECA's deployment. These clusters are fairly similar in configuration, provided by the same vendor, with the same family of processors (x86_64), the same OS family (Red Hat based linux distributions), same network technology (InfiniBand) and very similar filesystem setup. Nevertheless, they also have some significant differences. The processors, although all being x86_64, are from different generations (Intel Haswell on JURECA and JUAMS, and Intel Sandy Bridge on JUROPA3). This causes binary incompatibilities when using AVX instructions. The OS is CentOS 7.2 in JURECA and Scientific Linux 6.7 in JUAMS and JUROPA3, which ship different packages that impact our setup. Some nodes have special characteristics (JUROPA3 has additional Xeon Phi nodes and NVIDIA K20X GPUs, whereas JURECA has nodes equipped with NVIDIA K80 GPUs).

All these small but important differences make it impossible to share the same binary software installations. Nevertheless, the easyconfigs and general layout can be reused. Thus, in theory, only a simple recompilation per system is necessary. To do that, we have set up a common repository in a path shared by all three clusters. Having the same filesystem layout also allowed us to reuse the same EasyBuild configuration, provided by the so-called `InstallSoftware` module. As in JURECA, this module is provided to users belonging to the `software` group. Despite the similarities and our efforts to make the setup completely reusable, EasyBuild's *"sandboxing"* is not perfect, and small differences in the packages shipped with the OS caused the issues described below:

- Some easyconfigs used in JURECA where deficient, lacking dependencies or incorrect options. However, the packages provided by the OS compensated for this errors and these easyconfigs where inadvertently installed. Using them in JUAMS and JUROPA3 exposed these mistakes, which where then fixed.
- The `Subversion` version provided by Scientific Linux 6.7 did not meet the `UltraScan3` 3.3 requirements. Hence, it had to be listed as an extra dependency. Installing `Subversion` through EasyBuild pulled in also four extra dependencies.
- The `Python` version provided by Scientific Linux 6.7 was also too old for `GObject-Introspection`, `LLVM`, and `SCons` (pulled in by `Subversion`), requiring a new 'bare' Python installation, with minimal functionality at the bottom end of the toolchains hierarchy.
- JUAMS users required an old `GSL` version, which was not necessary in JURECA.

These issues required the creation of extra easyconfigs to supply the missing software, to adapt dependencies and/or to fix existing easyconfigs. Besides these changes, the number of toolchains and software in JUROPA3 and JUAMS is less than in JURECA, due to its different number of users and their requirements. It has to be stressed that the overall number of changes is minimal, as described in Table IV. This proves the high reusability of EasyBuild setups.

TABLE IV
SOFTWARE IN JUAMS AND JUROPA3.

Total packages in JUAMS	671
Total packages in JUROPA3	658
Ad-hoc packages in both	15

VII. FUTURE WORK

During the deployment of the 2016a *stage* in JURECA some requests for improvements were made. Tweaking hundreds of easyconfig files to change the version of some dependencies, or their own version, is an error prone and time consuming task. In the future, we aim at automate these tasks by implementing two additional features:

- Automatic upgrade of dependency versions. This will enable EasyBuild to use an already installed dependency, regardless of the specified version in the easyconfig, and save time by automatically updating all the easyconfigs that depends on an updated library. Of course this feature can be partly automated already using external scripts, but that would still require knowing specific versions and extra effort from the maintainer of the infrastructure.
- Automatic upgrade of software versions. This will enable EasyBuild to automatically crawl the source URLs, find new software versions, and do a best-effort attempt to compile it.

These two features used together would result in an easier upgrade of the whole software stack.

The hierarchical module naming scheme used in JURECA can be confusing for users that are not familiar with different compilers and MPI runtimes, and simply want to run an installed applications to perform their simulations. As a future work, we would like to provide default views depending on the user profile. A few setups worth considering are:

- Default set of compiler and MPI runtime modules for standard nodes.
- Default set of compiler and MPI runtime modules for GPU nodes.
- Default set of modules for visualization software.

Whether this is better achieved with an alternative module naming scheme, with saved collections of modules (another feature of Lmod), or with any other method, remains to be discussed.

EasyBuild helps to decouple the system software from the scientific software. However, as a side effect, it results in a large number of software packages being installed in non-standard locations. With dynamic linking, this requires that the `lib` directory of each software package is added to the `$LD_LIBRARY_PATH` of the running shell. The result of this is some extremely long strings contained in that variable. Another potential problem with dynamic linking is ending up with a wrong version of a library prepended on the library path, which can cause unexpected problems. Linking with `-rpath` is being considered in the EasyBuild community to solve these issues. However, implementing it in a portable and reliable way is not trivial. Should this happen, we would evaluate this method to minimize the described problem.

Currently, in our setup we do not keep track of modules usage. Lmod provides hooks that can link the loading and unloading of modules to particular actions. This can be used as a crude way to gather software usage metrics, which in turn would help us to focus on the most used packages. As a next step, we would like to evaluate XALT [22] for collecting more detailed data.

The use of *minimal toolchains* is already a very significant step forward towards reusing software installations, eliminating redundant packages and installing software at the right level in the toolchain hierarchy. However, we are currently considering reshuffling some binary packages to provide a more streamlined experience. CUDA, for example, is a binary package that is not compiled. By installing it at the compiler level we create additional restrictions, which results in users not being able to use CUDA code in toolchains with Intel 2016.2—even though GCC 4.9.3 is available through `GCCcore`—or ParaStationMPI. This is a constraint in our system that we would like to eliminate. Likewise, due to MKL being installed at the top level of the hierarchies, software that depends on the BLAS component of the MKL but does not require MPI, such as `numpy`, can not be installed in the compiler toolchains, which is an extra restriction.

Lastly, to reduce the burden that is maintaining hundreds of easyconfig files, we would like to develop and adopt *fat easyconfig* files support. In a first step, this would allow to merge easyconfigs for multiple toolchains into a single easyconfig file.

VIII. Conclusion

In this paper we have discussed the deployment of EasyBuild in a large scale cluster, with a non-trivial set of constraints and requirements. Our experience highlights that EasyBuild enables a small team to deploy and manage a tremendous amount of software, and that it is an alive and active project. As such, it evolves with the help of its community, and needed changes to meet our requirements. It also helps to keep an organized and efficient software deployment and retirement strategy. However, our experience also highlights that an important customization effort is still required to:

- Minimize software replication across toolchains.
- Provide the latest and greatest software versions, both from toolchain components and from other software.
- Provide a meaningful user view while meeting all other requirements.

Another important observation is the ability to migrate a well defined setup to similar systems, with low effort. This enables a coherent view for users of multiple systems in our site.

Finally, we would like to remark that managing a complete scientific software stack remains a complex endeavour. We have identified a few items to simplify some common tasks, and we would like to implement them in the future and add these improvements to our workflow.

References

[1] J. L. Furlani, "Modules: Providing a flexible user environment," in *Proc. of the 5th Large Installation System Administration Conference (LISA V)*, September/October 1991, pp. 141–152.

[2] M. Geimer, K. Hoste, and R. McLay, "Modern scientific software management using EasyBuild and Lmod," in *First Int'l Workshop on HPC User Support Tools (HUST-14)*, Nov. 2014, pp. 41–51.

[3] J. Layton, "Lmod – alternative environment modules," Admin HPC, http://www.admin-magazine.com/HPC/Articles/Lmod-Alternative-Environment-Modules.

[4] R. McLay, "Lmod: Environmental modules system," http://www.tacc.utexas.edu/tacc-projects/lmod.

[5] E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer, "JURECA in the Top 500 List," https://www.top500.org/system/178718.

[6] Jülich Supercomputing Centre, "The JURECA cluster," http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JURECA/JURECA_node.html.

[7] Par-Tec Cluster Competence Center, "ParaStationMPI," http://www.par-tec.com/products/parastation-mpi.html.

[8] ——, "ParaStationMPI github website," https://github.com/ParaStation/psmpi2.

[9] The Ohio State University, "MVAPICH," http://mvapich.cse.ohio-state.edu/.

[10] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. K. Panda, "GPU-Aware MPI on RDMA-Enabled Clusters: Design, Implementation and Evaluation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 10, pp. 2595–2605, Oct 2014.

[11] N. Jones and M. R. Fahey, "Design, Implementation, and Experiences of Third-Party Software Administration at the ORNL NCCS," in *Proc. of the 50th Cray User Group Meeting (CUG2008)*, 2008.

[12] Oak Ridge National Laboratory, "SWTools," 2011, https://www.olcf.ornl.gov/center-projects/swtools.

[13] A. Di Girolamo, "Smithy," http://anthonydigirolamo.github.io/smithy/.

[14] R. C. Bording, C. Harris, and D. Schibeci, "Using maali to efficiently recompile software post-CLE updates on a Cray XC system," in *Proc. of the Cray User Group Meeting (CUG2015)*, 2015.

[15] Pawsey Supercomputing Centre, "maali," https://github.com/Pawseyops/maali.

[16] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral, "The Spack package manager: Bringing order to HPC software chaos," in *Proc. of the Int'l Conf. for High Performance Computing, Networking, Storage and Analysis 2015 (SC'15)*, Nov. 2015, pp. 40:1–40:12.

[17] Lawrence Livermore National Laboratory, "Spack," https://github.com/LLNL/spack.

[18] K. Hoste, J. Timmerman, A. Georges, and S. De Weirdt, "EasyBuild: Building software with ease," in *Workshop on Python for High Performance and Scientific Computing (PyHPC)*, Nov. 2012.

[19] Ghent University, "EasyBuild," http://hpcugent.github.io/easybuild/.

[20] Argonne National Laboratory, "MPICH," http://www.mpich.org/.

[21] ——, "MPICH Compiler Optimization Levels - Section 2.4," http://www.mpich.org/static/downloads/3.2/mpich-3.2-installguide.pdf.

[22] K. Agrawal, M. R. Fahey, R. McLay, and D. James, "User Environment Tracking and Problem Detection with XALT," in *First Int'l Workshop on HPC User Support Tools (HUST-14)*, Nov. 2014, pp. 32–40.