Nix as HPC package management system

Bruno Bzeznik
UMS Grenoble Alpes Infrastructure de CAlcul intensif et de Données (GRICAD)
Saint Martin d'Hères, France
bruno.bzeznik@univ-grenoble-alpes.fr

Oliver Henriot
UMS Grenoble Alpes Infrastructure de CAlcul intensif et de Données (GRICAD)
Saint Martin d'Hères, France
oliver.henriot@univ-grenoble-alpes.fr

Valentin Reis
Laboratoire d'Informatique de Grenoble (LIG)
Saint Martin d'Hères, France
valentin.reis@inria.fr

Olivier Richard
Laboratoire d'Informatique de Grenoble (LIG)
Saint Martin d'Hères, France
olivier.richard@imag.fr

Laure Tavard
UMS Grenoble Alpes Infrastructure de CAlcul intensif et de Données (GRICAD)
Saint Martin d'Hères, France
laure.tavard@univ-grenoble-alpes.fr

ABSTRACT

Modern High Performance Computing systems are becoming larger and more heterogeneous. The proper management of software for the users of such systems poses a significant challenge. These users run very diverse applications that may be compiled with proprietary tools for specialized hardware. Moreover, the application life-cycle of these software may exceed the lifetime of the HPC systems themselves. These difficulties motivate the use of specialized package management systems. In this paper, we outline an approach to HPC package development, deployment, management, sharing, and reuse based on the Nix functional package manager. We report our experience with this approach inside the GRICAD HPC center[GRICAD 2017a] in Grenoble over a 12 month period and compare it to other existing approaches.

KEYWORDS

High Performance Computing; Package Management System; Nix

ACM Reference format:
Bruno Bzeznik, Oliver Henriot, Valentin Reis, Olivier Richard, and Laure Tavard. 2017. Nix as HPC package management system. In *Proceedings of HUST'17: HPC User Support Tools, Denver, CO, USA, November 12–17, 2017 (HUST'17),* 6 pages.
https://doi.org/10.1145/3152493.3152556

1 INTRODUCTION

The applications developed by institutional and industrial research teams have constantly growing computational needs. A solution to provide for these needs is to run computational jobs inside HPC centers, which allow to compute at a greatly increased scale by pooling hardware and human resources. However, the resulting communal management of hardware resources comes with significant challenges. While some of these challenges are technical, others are administrative. Notably, one has to appropriately let users access parts of the machine at specific times, with specific computational environments that suit their needs, all the while enforcing usage, security, and performance policies.

Resource and Job Management Systems such as OAR[LIG/INRIA 2003], SLURM[SchedMD 2003] or more recently Flux[Ahn 2014] provide solutions to the scheduling and user management issues, but they leave management of the user environment untouched. It is up to system administrators to manage this aspect for users. This problem is referred to as package management, e.g. the control of how the necessary software is made available in the environment provided to the end user. The package management problem in HPC is made particularly difficult by various aspects.

First, the quantity and diversity of HPC applications is rapidly increasing, due to both the emergence of new big-data yet compute-bound workflows and the development of specialized hardware (GPUS, Xeon Phi, low latency networks) [Meuer et al. 2014].

Second, application life cycles are usually longer than cluster life cycle: we can see 15 years old applications (Mumps, Vasp,...) running on clusters which typically last no more than 7 years. This increases the difficulty of maintaining up-to-date systems and makes it hard to provide stable environments to HPC users. Indeed, these applications have to be re-packaged along with the correct version of their dependencies.

Finally, HPC users require various custom and/or proprietary software which are commonly not initially packaged for mainstream operating systems. They may also need to customize the code or recompile it with some specific options during a period of optimization. Moreover, the build system of their application is often not "standard".

This article outlines an approach to HPC package development, management, sharing, and reuse. More precisely, we outline an approach using the Nix functional package manager. We report our experience with this approach inside the GRICAD HPC center[GRICAD 2017a] for a 12 month period. Section 2 reviews the currently existing approaches to this problem. Section 3 outlines the

HUST'17, November 12–17, 2017, Denver, CO, USA

ACM ISBN 978-1-4503-5130-0/17/11…$15.00
https://doi.org/10.1145/3152493.3152556

main aspects of the Nix package manager and its use in HPC. Section 4 presents our experience with using this approach at GRICAD. Section 5 concludes on aspects yet to be explored.

2 RELATED WORKS

This section presents the existing approaches to software configuration management along with their tradeoffs and their usability in HPC.

Virtualization and containers. Virtualization and its low-overhead counterpart, linux containerization, are the most popular tool in the service deployment community, allowing for complete control over the deployed system image. This approach has two drawbacks with impede its use in HPC. First, both techniques induce a significant computational overhead, which is prohibitive for use in compute-bound tasks. Second, the users usually have to generate full system images using a tool such as docker, which incurs a significant development time and/or additional complexity compared to, say, being provided a node with a standardized environment.

System images. A moderately spread method is to flash a whole custom system image on the nodes when they are allocated. Systems such as grid5000 [Balouek et al. 2013] use this approach powered by the OAR [LIG/INRIA 2003] resource manager. However, these are experimental machines with heavy emphasis on reconfiguration that do not run heavy production code. Indeed, production HPC hardware usually does not provide the capacities for modifying and restoring the operating system at will on compute nodes.

Distribution package managers. Classic linux distributions provide package managers such as apt that installs *deb* packages on Debian, yum for *rpm* packages on Fedora or pacman on Arch Linux. These package managers allow installing packages at the global system level, and are therefore not suitable for multi-user environments such as HPC nodes.

Environment modules. The most widely used tool in the HPC world is the Environment Module system, also known as the "module" command line tool: administrators install libraries and applications from sources into sub-directories of a directory that is shared on every node of the computing cluster. Then, environment variables such as PATH and LD_LIBRARY_PATH are configured into a "modulefile" that may be loaded by the user to set up its environment for access to the application. This method has some constraints:

- Difficulty to maintain due to the multiplication of the dependencies,
- Growth of the applications directory, as new versions or new flavors (compilation options for example) are installed into sub-directories
- Problem of reproductibility: system updates can break the application repository and may force to recompile everything,
- Problem of portability: the development of the set of modules is local and it is hard to move the code from one center to another. The set of modules is only available for one platform

Automatic build systems. Here we present a short state of the art about a new generation of tools to deploy applications. It includes:

- EasyBuild[EasyBuild 2017] (Ghent University)
- Spack[Spack 2017] developed at Lawrence Livermore National Laboratory which automates builds of software.
- Guix[Guix 2017] and Nix[Nix 2017b] are package management software which offer binaries, in contrast to the 2 above.

The main difference between Guix/Nix and Easybuild/Spack is that they do not rely on system dependencies. Easybuild and Spack are tools that provide recipes for automatically rebuilding software. But the base system may provide (or not) some dependencies for those softwares. So, the result of a recipe may vary depending on the base system and it's configuration. While Easybuild and Spack provide a way to automatically compile softwares and dependencies on a base system, Guix and Nix provide binary packages that do only depend on a Linux or a Macos kernel. To be clear, Guix and Nix embed their own libc and nothing is linked against libraries outside Guix or Nix. Moreover, they may embed a large number of different libc versions depending on the needs and the evolution of their store. This is a key feature for portability and reproducibility. Another key feature for reproducibility is that packages are identified by a unique sha256 hash. A simple modification to the code of a package leads to another hash, and then may trig an automatic recompilation of the binaries of the package or the binaries that depend on this package. Spack also relies on a hash system, but only to lead with versions/configurations/platforms/compilers combinatory of the recipes [et al. 2017].

Regarding Guix vs Nix, our choice has been very pragmatic: at the moment of migrating to this solution, Guix had very few available packages compared to Nix. Furthermore, the community seemed to be larger. A HPC-centered Guix extension, Guix-HPC [Courtès and Wurmus 2015] represents the Guix approach to the problem. Finaly, the availability of the NiXOS system, a Linux operating system based on the Nix packages management system has also be a point for us, as we would like to test it as a base system for an HPC cluster in the next few years.

Other existing centers such as Compute Canada [Baldwin 2012] use Nix as part of their workflow. We do not comment on their approach in this article for timing reasons.

In the following section, we focus on how we use Nix in our HPC center.

3 NIX OVERVIEW

This section presents the main features of the Nix package manager and its application to a multi-user HPC Linux platform.

A purely functional package manager. Nix is a tool which allows automated builds of software. It was developed by Eelco Dolstra as a part of his thesis at Utrecht University [Dosltra 2006]. The main motivation for the creation of such a tool was rethinking the idea of packaging with the wish to add features such as reproducible builds, portability, ...

Before presenting *Nix*, we have to define 3 terms:

- *Nix*, the multi-platform package manager system based on its own purely functional language,
- *Nixpkgs*, the repository of available source codes on Github [Nix 2017a],

- *NixOS*, a Linux distribution based on *Nixpkgs* which you can install if you want to explore further this tool[DOLSTRA et al. 2010].

Here, we only focus on the key features which respond to issues raised by the prior section and which make it a robust and interesting tool as a software deployment system on an HPC cluster.

In the following section, we will consider the following definition for "software component" given by Eelco Dolstra [Dosltra 2006]:

> *- A software component is a software artifact that is subject to automatic composition.*
> *It can require, and be required by, other components.*
> *- A software component is a unit of deployment.*

▹ Nix store

The Nix system can be described as a global installation: a single directory called `nix` is created on the system. The *Nix store* (`/nix/store`) designates a Read-Only directory where all the applications are stored. There is no other dependence on the system and therefore "no pollution of the file system".

▹ Derivations & Cryptographic hashes

A *derivation* is a set of functions placed in a file, generally called `default.nix`, and written in the *Nix Expression Language* which allows building of the software component. After building, each software component is stored and traceable by its own *Store path* composed by a unique 32-character long cryptographic hash generated from a combination of sources, configuration files, libraries version, ... attached to its name. For example, for the current version of the Netcdf library with default options defined in the derivation file in the *Nixpkgs* packages collection [contributors 2017], we have:

```
/nix/store/s9li252l2ajjh7z0pn3m1rwdj17333f6
                -netcdf-4.3.3.1
```

The main idea behind those characteristics is the singularity of the name thus generated by the script used to build the software. When users or administrators want to install a new version a new hash is created which ensures its singularity and avoids conflict between different versions of the software.

▹ Profiles and Generations.

Another important notion is the *profile* which is a set of software components. Each user can have one or more *profiles*, whose environments don't interfere which each other. Each profile provides an isolated environment. Profiles are versioned as *generations*. Indeed, every profile version obtained by imperatively installing (`nix-env -i`) or removing (`nix-env -e`) software remains accessible via `nix-env --list-generations`. When a user rolls back to a prior *generation* (`nix-env --switch-generation ID`) they can then delete this *profile* by running the following command line: `nix-env --delete-generations xx`, where `xx` means the ID of the generation to delete.

▹ Nix garbage-collector

The *Nix garbage-collector* allows to reduce disk space occupation and only deletes packages which are not used by a *profile* or a *generation*.

▹ Channel & binary-cache

The *channel* is an http server providing a *binary-cache* directory and a tar archive of the maintained Nix-expressions.
In the last section, we present our experiment and how the GRICAD HPC center uses its own channel[GRICAD 2017b].

Shared Nix Store. One of the most important things to set-up to allow users to use Nix is a `/nix` directory shared on all the computing and head nodes. This path must be exactly */nix* in order to be able to use the official Nix binary-caches. It's possible to install the Nix store in another path, but in this case, every package and its dependencies will need to be recompiled as the pre-compiled binaries will not be usable. Package recompilation is a task which Nix does very well, on the fly, at installation time but the problem is that it may require quite a lot of time and resources on the head node. So, for example, you would typically set-up an NFS file-system mount-point on all of your nodes.

Daemon & multi-user mode.

> *The Nix daemon is necessary in multi-user Nix installations. It performs build actions and other operations on the Nix store on behalf of unprivileged users*

Nix will allow the users to automatically build (i.e. compile) the content of the packages. In a multi-user environment, Nix provides a daemon which is responsible for the security of the shared Nix store. The builds and package installations are not directly done by the users, but by the `nix-daemon` which is a pool of anonymous build users. This principle also allows you (as the system administrator) to have some control over the build process, for example if you want to limit the number of build processes that can be run at the same time.

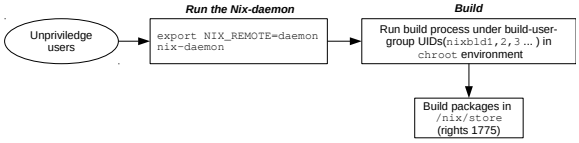

Figure 1: Build processes, `nix-daemon` & rights

To use Nix in multi-user mode, your users will have to source a shell script into their environment where administrators might have to add/customize some environment variables.
Basically, this script:

- sets the `PATH` to Nix tools binaries,
- sets the `NIX_PATH` variable, which may be necessary for some advanced operations and the use of a custom channel,
- initializes per-user directories and configuration files,
- sets the `NIX_REMOTE` variable which is necessary to use the Nix daemon

4 NIX IN THE GRICAD HPC CENTER

A complete description of the installation of Nix on our HPC system is available on our web site [Bzeznik 2017].

GRICAD HPC center environment. What we call an HPC cluster here, is simply a set of interconnected Linux computing nodes and one or several head nodes. The computing nodes and the head

nodes share some common network filesystem such as NFS, Lustre or BeeGFS. Users log on the head nodes and submit jobs on the computing nodes by using the OAR[LIG/INRIA 2003] batch scheduler. Users have no direct access to the nodes but may do some interactive tasks for preparing the jobs on the head nodes.

Nix is installed on 2 clusters: a BullX HPC cluster with 3,200 cores CentOS based and a Debian based heterogeneous cluster. It has been introduced to the users during a local HPC user day: a one day exhibition offering an overview about the HPC center developments and actualities. Regular internal trainings are scheduled to present how to use it on our HPC cluster.
Today, Nix and Module Environment coexist on our clusters. Nix has only relatively recently been deployed on our clusters and is mostly used by early adopters. Current adoption of Nix is reflected by user ratio: we have twenty-five out of about three-hundred users on the HPC cluster and thirty-four out of about one hundred users on the heterogeneous cluster.

▹ Usage Report.

On the HPC cluster, Nix was installed in February 2016 in a dedicated partition. At the time of this publication, the following statistics can be gleaned:

- Size. the `/nix/store` takes up ~ 100 Gb on the first cluster and ~ 44Gb on the second cluster. These usage figures were obtained without using the functionality of the *garbage-collector* with the `nix-collect-garbage` command.
- Derivations. There is currently a total of 24390 derivations in the Nix store.
- Users. There were 53 users involved in the use of Nix on the system, which was optional.
- Generations. There was a total of 1235 generations, which corresponds to about the same or greater number of calls to the nix-env command, whether to install or remove packages in the current environment. This amounts to an average of 23 generations per user.

Among these 53 users, we typically encountered two types of opposite reactions. On one hand, some users did not adapt well to the tool, instead relying on the system administrator to perform environment management operations on their behalf. On the other hand, another group of more experienced users were very happy to have the system available for their package management needs. Users from this second group usually already familiar with the packaging problem, or had used Nix and its ecosystem independently from their HPC needs. These users particularly appreciated the ability to build software identically on their personal machine or on the HPC machine.

▹ Package development.

We have created a custom channel [GRICAD 2017b] to hold packages of non-free applications and package variants we have contributed to but that are not already in the official distribution. Additionally, this chanel contains packages that can not be pushed upstream due to the aspects mentionned in the following paragraph. This is the equivalent to the Guix-Hpc initiative [?]. Currently we provide these packages for our users:

```
{ stdenv, fetchurl, glibc, gcc }:

stdenv.mkDerivation rec {
  version = "2017";
  name = "intel-compilers-${version}";
  sourceRoot = "/scratch/intel/${version}";

  buildInputs = [ glibc gcc ];

  phases = [ "installPhase"
    "fixupPhase" "installCheckPhase"
    "distPhase" ];

  installPhase = ''
    cp -a $sourceRoot $out
    # Fixing man path
    rm -f $out/man
    mkdir -p $out/share
    ln -s ../compilers_and_libraries/\
      linux/man/common \
      $out/share/man
  '';

  postFixup = ''
    echo "Fixing rights..."
    chmod u+w -R $out
    echo "Patching rpath and interpreter..."
    find $out -type f \
      -exec $SHELL -c 'patchelf \
      --set-interpreter \
      $(echo ${glibc}/lib/ld-linux*.so.2) \
      --set-rpath \
      ${glibc}/lib:${gcc.cc}/lib:\
      ${gcc.cc.lib}/lib:\
      $out/lib/intel64:$out/\
      compilers_and_libraries_2017.4.196/\
      linux/bin/intel64 2>/dev/null {}' \;
    echo "Fixing path into scripts..."
    for file in \
      `grep -l -r "$sourceRoot" $out`
    do
      sed -e "s,$sourceRoot,$out,g" -i $file
    done
  '';

  meta = {
    description = "Intel compilers
      and libraries 2017";
    maintainers = [
      stdenv.lib.maintainers.bzizou ];
    platforms = stdenv.lib.platforms.linux;
  };
}
```

Figure 2: Intel suite compilers 2017 derivation

- Intel suite compilers
- Gildas
- irods
- openmpi
- singularity

Other packages are under development such as: OpenFoam, CDO.

Listing 2 shows an example of Nix derivation which packages the Intel suite compilers (provided as binary form). It is to note that this package can not be rebuilt on the nodes themselves. Its availability depends on the existence Nix *binary-cache*. Indeed, the `/scratch/intel` exists only on the node that was initially used to build the package.

Another HPC-specific situations include building MPI with OFED support. In such a case, the OFED library is built with Nix as usual, and looks as per default at runtime in the system-configured `/etc/libverbsd` folder on the machine.

Yet another case is intstalling opengl drivers. In such a case, the Nix package manager will rely on the availability of the proper platform-specific drivers in the standard `/run/opengl/drivers` folder.

▹ Limitations.

We have come across two main limitations in our experience with the system.

First, the system can lack conviality for new users. Indeed, it is not easy to install a package with a custom compilation attribute using a one-liner. The user will have to go through our previously mentioned tutorial and create a file containing the necessary derivation to achieve this effect. In this regard, the Spack [Spack 2017] system exhibits a better behavior.

Second, the Nix system still has some unresolved issues such as the interplay of language-specific, often source-based package managers. For example, the case of Ocaml or even Python often gives rises to question from the users. Users have to decide between using the de-facto standard *pip* python package manager, or using perhaps slightly delayed Nix packages.

5 CONCLUDING DISCUSSION AND FUTURES PERSPECTIVES

Even though the module environment is the most used tool to manage deployment of application in the HPC administrator community, some of its properties are significant obstacles in daily HPC system administration. To offset those difficulties and in response to driving needs, new package manager systems appear. In our HPC Center, we looked into installation of Nix, a purely functional package manager based on its own language. Reviewing the situation over the past year, we can made a sort critical appraisal on the obtained results and give some perspectives.

Feedback. Technically, it is necessary to point out some artefacts which could discourage the new users which are:

- the Nix-expressions, if you are not used to the functional language,
- the lack of the combinatorial aspect, such as is available in Spack[et al. 2017].

On the other side, Nix delivers strong technical capabilities such as its easiness of installation. As mentioned above, Nix deployment is reduced to one directory and avoids pollution of the entire system already in place. To go even faster in this approach, you can have both Module Environment and Nix installed to test it without having any side effects on your basic environment as the Nix environment does not spread. To continue in the "system way", centralized package management on a repository insure coherency of the repository and it can be shared with other community. On the user side, when a new version of a software is available, it possible to upgrade it without interfering with the old version. It is only a switch of environment or rollback if the upgrading does not work. Writing a configuration file also offers benefits as it allows automation of the task of software building and also singularity of the package due to the cryptographic hash. These benefits allow to install this software on an another HPC system where Nix is already, we can repeat this build and also ensure reproductibility of a set of experiences.

Perspectives. To sum-up, in the GRICAD HPC Center, we provide Nix as a reproducible and portable computing environment for the users of our High Performance Computing facilities. We aim to continue to develop new HPC software components in order to create an international working-group in the Nix workflow. We also look forward to deploy an entire 80 node HPC cluster using the NiXOS Linux distribution within the GRICAD HPC center.

ACKNOWLEDGMENTS

We thank the rest of the Pôle-Calcul team for their implication in this project. We also gratefully acknowledge the GRICAD users community for their contribution in developing new packages. Warm thanks go to Olivier Richard, Michael Mercier and Adrien Faure for helpful discussions. for helpful discussions. Special thanks to Lucas Bruno for his blog post[Bruno 2017] and the Nix development team.

REFERENCES

Dong H. Ahn. 2014. Flux: A Next-Generation Resource Management Framework for Large HPC Centers. (2014). Retrieved July 28, 2017 from https://flux-framework.github.io/papers/Flux-SRMPDS-final.pdf

Susan Baldwin. 2012. Compute Canada: advancing computational research. In *Journal of Physics: Conference Series*, Vol. 341. IOP Publishing, 012001.

Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. 2013. Adding Virtualization Capabilities to the Grid'5000 Testbed. In *Cloud Computing and Services Science*, Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan (Eds.). Communications in Computer and Information Science, Vol. 367. Springer International Publishing, 3–20. https://doi.org/10.1007/978-3-319-04519-1_1

Lucas Bruno. 2017. Lethalman Blog. (2017). Retrieved July 28, 2017 from http://lethalman.blogspot.fr/2014/07/nix-pill-1-why-you-should-give-it-try.html

Bruno Bzeznik. 2017. Setting up NIX on a multi-users HPC environment. (May 2017). Retrieved July 28, 2017 from https://gricad.github.io/calcul/nix/hpc/2017/05/15/nix-on-hpc-platforms.html

Nixpkgs contributors. Retrieved 2017. NetCDF software component. (Retrieved 2017). Retrieved July 28, 2017 from https://github.com/NixOS/nixpkgs/blob/master/pkgs/development/libraries/netcdf-cxx4/default.nix

Ludovic Courtès and Ricardo Wurmus. 2015. Reproducible and User-Controlled Software Environments in HPC with Guix. In *2nd International Workshop on Reproducibility in Parallel Computing (RepPar)*. Vienne, Austria. https://hal.inria.fr/hal-01161771

EELCO DOLSTRA, ANDRES LÃŰH, and NICOLAS PIERRON. 2010. NixOS: A Purely Functional Linux Distribution. *Journal of Functional Programming*. 20, 5-6 (2010), 577–615. https://doi.org/10.1017/S0956796810000195

Eelco Dosltra. 2006. *The Purely Functional Software Deployment Model*. Ph.D. Dissertation. University. Retrieved July 28, 2017 from http://nixos.org/~eelco/pubs/phd-thesis.pdf

EasyBuild. 2017. EasyBuild: building software with ease. (2017). Retrieved July 28, 2017 from https://easybuilders.github.io/easybuild/

Melara et al. 2017. Using Spack to Manage Software on Cray Supercomputers. (2017). Retrieved July 28, 2017 from https://cug.org/proceedings/cug2017_proceedings/includes/files/pap153s2-file1.pdf

GRICAD. 2017a. GRAICAD Web site. (2017). Retrieved July 28, 2017 from hgj,

GRICAD. 2017b. GRICAD Ciment packages collection. (May 2017). Retrieved July 28, 2017 from https://github.com/Gricad/nix-ciment-channel

Guix. 2017. The Guix System Distribution. (2017). Retrieved July 28, 2017 from https://www.gnu.org/software/guix/

LIG/INRIA. 2003. OAR Website. (2003). Retrieved July 28, 2017 from https://oar.imag.fr/

Hans Werner Meuer, Erich Strohmaier, Jack Dongarra, and Horst D. Simon. 2014. *The TOP500: History, Trends, and Future Directions in High Performance Computing* (1st ed.). Chapman & Hall/CRC.

Nix. 2017a. Nix packages collection. (2017). Retrieved July 28, 2017 from https://github.com/NixOS/nixpkgs

Nix. 2017b. Nix. The Purely Functional Package Manager. (2017). Retrieved July 28, 2017 from https://nixos.org/nix/

SchedMD. 2003. Slurm Website. (2003). Retrieved July 28, 2017 from https://slurm.schedmd.com/

Spack. 2017. A flexible package manager that supports multiple versions, configurations, platforms, and compilers. (2017). Retrieved July 28, 2017 from https://spack.io/