Chapter 4

Are you root? Reproducible Experiments in User Space

Vinícius Garcia Pinto
Centro de Ciências Computacionais, Universidade Federal do Rio Grande
Rio Grande, Brasil

Lucas Leandro Nesi, Lucas Mello Schnorr
Instituto de Informática, Universidade Federal do Rio Grande do Sul
Porto Alegre, Brasil

Abstract

High-performance computing platforms are usually shared by users with diverse demands. Often legacy applications or those with very specific requirements and dependencies overload system administrators or simply cannot run. A solution is to build, compile, and install the application and its dependencies fully in user space. However, this task is timely expensive and error-prone, which motivates the use of automated solutions. In this chapter, we present two solutions to install packages fully in user space in a shareable and reproducible way.

Resumo

Plataformas computacionais para processamento de alto desempenho são usualmente compartilhadas por usuários com demandas variadas. Com frequência aplicações legadas ou aquelas que possuem requisitos e dependências muito específicos sobrecarregam os administradores do sistema ou simplesmente tem sua execução inviabilizada. Uma solução passa a ser configurar, compilar e instalar a aplicação e as respectivas dependências inteiramente em espaço de usuário. Entretanto, tal tarefa é custosa e propensa a erros, motivando a adoção de soluções automatizadas. Neste capítulo, apresentamos duas soluções que permitem a instalação de pacotes inteiramente em espaço de usuário de maneira reprodutível e compartilhável.

4.1. Introduction

One fundamental step when using computational platforms is the correct installation of software. When considering large, shareable resources, users usually do not have administrative permissions and must install their software stack in regular user environments. This installation may also require many dependencies that suffer from the same principle.

Although one can manually download the latest source and check the newest manual for each one of the dependencies, this would present many problems. These problems include spending significant time in this process, a chance that such steps are not portable for other platforms, and discovering that specific versions of dependencies may not work for that software stack. These problems motivate modern approaches that can handle complex dependencies stacks while maintaining the installation reproducibility of that build in a non-privileged environment.

Two initiatives, Spack and Guix, aim to install software and its dependencies in user space while being portable and reproducible. This course focuses on explaining both tools and how to create installation recipes for desired software.

The remainder of this chapter is structured as follows. Section 4.2 presents the Spack package manager. Section 4.3 presents the usage of Guix as a package manager. Section 4.4 concludes the chapter with a discussion and future perspectives.

4.2. Part I – Spack

Spack (GAMBLIN et al., 2015) is a fully user-space package manager that targets High-Performance Computing platforms. Package definitions and Spack itself are written in Python, each package being a class that inherits from a base class representing a standard build tool. To request the installation of a package, the user must prepare a package specification that can be as brief as `<package-name>` or as detailed as `<architecture compiler compiler-version package-name package-version list-of-package-options>`. The same structure applies recursively to package dependencies, resulting in a Directed Acyclic Graph (DAG). Figure 4.1 shows the DAG with the software stack of `openblas` in two stacks: one depending on `perl@5.37.9`, and the other on `perl@5.38.0`. Each node represents a unique package specification identified with a hash. At the install phase, Spack will download the tarball and build each package in the DAG, setting the correct paths at `configure` step (or equivalent). Even if identical dependencies can be recycled, as one can see in the figure, many installations of the same package can coexist.

4.2.1. Installation

Spack can be installed and deployed entirely in user space as long as the target system meets some minimal requirements. For a modern Linux distribution, the expected tools are: a Python 3 interpreter; C/C++/Fortran compilers; standard build tools such as `configure`, `make`, and `cmake`; `git`, compression/decompression tools such as `tar` and `gzip`; `curl`.

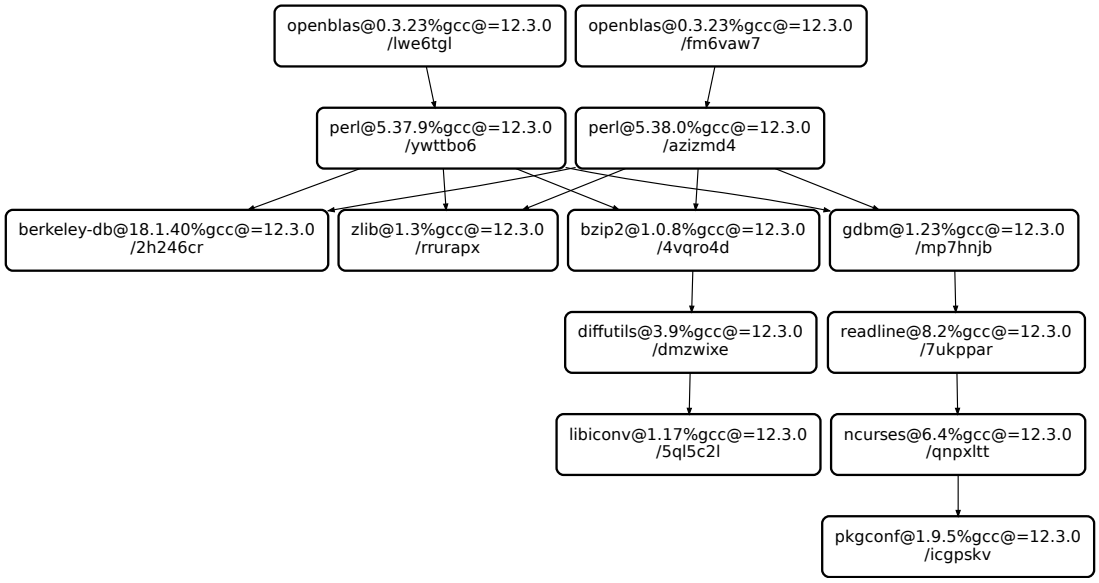

Figure 4.1. Dependency graph of two OpenBLAS installs at Spack

Once these dependencies are available, one can install Spack with:

SH

```
git clone --branch v0.20.1 https://github.com/spack/spack.git
source spack/share/spack/setup-env.sh
```

Spack will try to detect installed compilers automatically. It is possible to check which ones were detected with:

SH

```
spack compilers
```

STDOUT

```
Available compilers
-- clang ubuntu23.04-x86_64
   ------------------------------------
clang@15.0.7

-- gcc ubuntu23.04-x86_64
   ------------------------------------
gcc@12.3.0
gcc@13.2.0
gcc@4.8.5
```

After upgrades or when installing new compilers on the system, one should run `spack compiler find` before being able to use it. Inclusion of compilers installed in non-standard paths can be done with:

SH

```
spack compiler add /local/my-compiler
```

4.2.2. Installing packages

Before installing a new package, one may wish to check the full list of available packages with `spack list`. Filtering this output with a pattern is also possible:

SH

```
spack list "z*"
```

STDOUT

```
z-checker  z3  zabbix  zfp  zfs  zig  zip  zipkin  zlib  zlib-ng
    zoltan  zookeeper  zookeeper-benchmark  zopfli  zpares  zsh
    zstd  zstr  zziplib
19 packages
```

The basic command to install a new package is `spack install <package name>`. This command will download, configure, build, and install `zlib` in the current Spack tree. Any other `zlib` installation living in the root system or in another spack instance will not be touched. Let's see an example:

SH

```
spack install zlib
```

The previous `install` command used only the default parameters defined in the `zlib` package. A summary of all the package information, including its description, versions, configure and build options (i.e., the variants in the Spack jargon), and dependencies is obtained with:

SH

```
spack info zlib
```

STDOUT

```
MakefilePackage:   zlib

Description:
    A free, general-purpose, legally unencumbered lossless
       data-compression
    library.

Homepage: https://zlib.net

Preferred version:
    1.3       http://zlib.net/fossils/zlib-1.3.tar.gz

Safe versions:
    1.3       http://zlib.net/fossils/zlib-1.3.tar.gz
    1.2.13    http://zlib.net/fossils/zlib-1.2.13.tar.gz

Deprecated versions:
    1.2.12    http://zlib.net/fossils/zlib-1.2.12.tar.gz
    1.2.11    http://zlib.net/fossils/zlib-1.2.11.tar.gz
    1.2.8     http://zlib.net/fossils/zlib-1.2.8.tar.gz
    1.2.3     http://zlib.net/fossils/zlib-1.2.3.tar.gz

Variants:
    Name [Default]             When      Allowed values
       Description
    =======================    ======    ================
       ==================================================

    build_system [makefile]    --        makefile, generic
       Build systems supported by the package
    optimize [on]              --        on, off
       Enable -O2 for a more optimized lib
    pic [on]                   --        on, off
       Produce position-independent code (for shared libs)
    shared [on]                --        on, off
       Enables the build of shared libraries.

Build Dependencies:
    None

Link Dependencies:
    None

Run Dependencies:
    None
```

One of the main features of Spack is to allow the coexistence of multiple installations of

the same package. This way, one can install a second instance of `zlib` enabling (with `+`) the `pic` variant and disabling (with `~`) the `shared` and `optimize` variants:

SH

```
spack install zlib+pic~shared~optimize
```

As well as variants, installations using different version numbers, specified with `@<version-number>` and/or compiled with other compilers (using `%<compiler>`) can also coexist:

SH

```
spack install zlib@1.2.13+pic~shared~optimize%clang
```

These three installations live in the same Spack tree. Spack uses rpath (*run-time search path*) to keep each executable isolated with its dependencies. One can check a list of installed packages with `spack find`, which can include a package name to filter the search, for example:

SH

```
spack find zlib
```

STDOUT

```
-- linux-ubuntu23.04-ivybridge / clang@15.0.7
   ------------------
zlib@1.2.13

-- linux-ubuntu23.04-ivybridge / gcc@12.3.0
   --------------------
zlib@1.3  zlib@1.3
3 installed packages
```

When there are several installations of the same package, it may become difficult to differentiate them. The output of `spack find` can be extended with the options `-v` to show the full spec and `-L` to list the hash that acts as an installation's unique identifier:

SH

```
spack find -L -v zlib
```

STDOUT

```
-- linux-ubuntu23.04-ivybridge / clang@15.0.7
   ------------------
qkdk6s2blk5dqqyha3pper6uelymtwpr zlib@1.2.13~optimize+pic~shared
    build_system=makefile

-- linux-ubuntu23.04-ivybridge / gcc@12.3.0
   --------------------
xkwdb6nqvajfcwfjuc75vrc7stzc7zck zlib@1.3~optimize+pic~shared
   build_system=makefile
rrurapxw3hus5ia5vrszozxykmcoffw2 zlib@1.3+optimize+pic+shared
   build_system=makefile
3 installed packages
```

To distinguish very similar installations, one can provide hashes to `spack diff`:

SH

```
spack diff /qkdk6s2b /xkwdb6nq
```

STDOUT

```
--- zlib@1.2.13/qkdk6s2blk5dqqyha3pper6uelymtwpr
+++ zlib@1.3/xkwdb6nqvajfcwfjuc75vrc7stzc7zck
@@ hash @@
-  zlib qkdk6s2blk5dqqyha3pper6uelymtwpr
+  zlib xkwdb6nqvajfcwfjuc75vrc7stzc7zck
@@ node_compiler @@
-  zlib clang
+  zlib gcc
@@ node_compiler_version @@
-  zlib clang 15.0.7
+  zlib gcc 12.3.0
@@ package_hash @@
-  zlib 2w7eqxylpvimalu26prt37tmbdsbqbvjytjjlvpyph6yuqwl63ga====
+  zlib u7vqvwmacj5j7zngg2evhytlxmzhad35mxlxna6tmr4bjyeisgsa====
@@ version @@
-  zlib 1.2.13
+  zlib 1.3
```

Installing a package may require a long dependency chain. With `spack spec`, one can check which new packages will be installed and which ones are already installed and can be reused (i.e., those prefixed with `[+]`).

SH

```
spack spec cbc
```

STDOUT

```
Input spec
--------------------------------
 -   cbc

Concretized
--------------------------------
 -   cbc@2.10.9%gcc@12.3.0 build_system=autotools arch=
    linux-ubuntu23.04-ivybridge
 -       ^cgl@0.60.7%gcc@12.3.0 build_system=autotools arch=
    linux-ubuntu23.04-ivybridge
 -           ^clp@1.17.7%gcc@12.3.0 build_system=autotools arch=
    linux-ubuntu23.04-ivybridge
[+]      ^coinutils@2.11.9%gcc@12.3.0 build_system=autotools
   arch=linux-ubuntu23.04-ivybridge
[+]      ^osi@0.108.8%gcc@12.3.0 build_system=autotools arch=
   linux-ubuntu23.04-ivybridge
[+]          ^pkgconf@1.9.5%gcc@12.3.0 build_system=autotools
   arch=linux-ubuntu23.04-ivybridge
```

Several High-Performance Computing platforms restrict Internet access from compute nodes, making users download all external data on the front-end and then automatically exporting `/home` or `/scratch` directories to compute nodes via NFS (*Network File System*). At the same time, many of these platforms also limit user process duration on the front-end. These two policies, however, can disturb the use of package managers as Spack, since even the building of a single package can demand the download and long compilation of several dependencies. To handle this scenario, one can rely on `spack fetch` to download not only the target package but also its dependencies. For example, to fetch the default `llvm` release and all its missing dependencies (i.e., those that are not yet installed), we run:

SH

```
spack fetch --dependencies --missing llvm
```

After fetching the package and its dependencies on the front-end, the user can log into the compute nodes and proceed the installation with `spack install` using the cached files available in NFS home directory.

When removing a package, it is possible to remove only the package or include all its dependents, i.e., any package that depends on the one being removed. For example, to remove `zlib` and its dependents:

SH

```
spack uninstall --dependents zlib
```

4.2.3. External packages

In terms of reproducibility, ideally, the entire software stack should be managed by Spack. However, many production HPC platforms provide customized or third-party MPI, accelerator, compiler, or linear algebra libraries. To take advantage of these optimizations, one can rely on `spack external` commands that will incorporate these package and their respective paths into the Spack structure. For example, to add a system installed `openmpi`:

SH

```
spack external find openmpi
```

STDOUT

```
==> The following specs have been detected on this system and
   added to /home/user/.spack/packages.yaml
-- no arch / gcc@11.4.0
   ----------------------------------------
openmpi@4.1.2
```

After this, one can use the discovered package (i.e., `openmpi@4.1.2`) as a usual dependency to build other packages:

SH

```
spack install hwloc^openmpi@4.1.2
```

4.2.4. Using a package

After installing a package, Spack provides two ways to use it. The simpler one is `spack load` which will load and update the `PATH` and `MANPATH` environment variables with the install paths of the package and its dependencies. For example, one can load the `unrar` package and then use the homonymous command.

SH

```
spack load unrar
```

Another option is to use `spack view` to create a new directory populated with links to mimic the traditional UNIX tree structure, i.e., `bin/`, `lib/`, `include/`. Users can choose between symbolic or hard links. This is more powerful than `spack load` enabling not only the execution but also development using the installed packages as third-party libraries. For example, to create such structure using soft links for `openblas` and its dependencies, and then compile and execute an application:

SH

```
spack view soft blas-dir openblas
gcc blas-example.c -Lblas-dir/lib -Iblas-dir/include -lopenblas
   -o ex-openblas
LD_LIBRARY_PATH=blas-dir/lib ./ex-openblas
```

4.2.5. Sharing configs

Users can share their stack of installed packages by creating a `spack.yaml` file. This file can be later used to rebuild and deploy the setup in another moment and machine in a reproducible way. Each `spack.yaml` is associated with an environment, so we start with `spack env create` and `activate`. Once the environment is ready, we can add the desired packages. At this point, these packages are not installed, they just are included in the `spack.yaml` file. To effectively install them, we need to explicitly run `spack install`.

SH

```
spack env create --dir spack-env-wscad-2023
spack env activate spack-env-wscad-2023
spack add zlib@1.2.3
spack add nano@4.7
spack add vim features=tiny ^ncurses@6.1
spack install
spack env deactivate
```

The `spack.yaml` file contains abstract information about the packages explicitly added to the environment. This information is flexible since it does not contain details such as implicit dependencies, compiler, or the target platform. The concrete information, as returned by `spack spec`, is stored in another file named `spack.lock`. Such file allows a faithful reproduction but works only on a very similar platform, i.e., same OS version (e.g., `ubuntu22.04`) and same CPU family (`intel.ivybridge`).

YAML

```
# This is a Spack Environment file.
#
# It describes a set of packages to be installed, along with
# configuration settings.
spack:
  # add package specs to the `specs` list
  specs: [zlib@1.2.3, nano@4.7, vim features=tiny ^ncurses@6.1]
  view: true
```

Both files can be shared and later imported in another machine with `spack env create`. For example, to import on a very similar machine:

SH

```
# machine with same arch, OS, compiler
spack env create foo spack.lock
```

Otherwise:

SH

```
# on every machine
spack env create foo spack.yaml
```

4.2.6. Preparing a new package

Spack packages are written in Python. Creating a new package boils down to creating a `package.py` file containing basic package information, such as description, download URL, versions, maintainers, dependencies, and build system. The command `create` provides templates for common general-purpose build systems such as `autotools`, `meson`, and `cmake`. There is also some support for language-specific packages, e.g., `R`, `Python`, and `Ruby`. As an example, a draft package for the `pajeng` tool[1], which is built using CMake, can be obtained with:

SH

```
spack create --name pajeng --template cmake "https://github.com/
   schnorr/pajeng/archive/1.3.6.tar.gz"
```

This command will prepare a directory for the new package `pajeng` in the default spack tree. This directory contains at least the `package.py` file but can also store patch and test files.

In the code listing below, source code lines 4 to 8 describe basic information as description and homepage of `pajeng`; this information will be displayed by `spack info`. Line 10 lists the github username of package maintainers while lines 12 to 16 define two versions to install `pajeng`: release 1.3.6 and from the `git` repository. The parameter `preferred` sets the first one as preferential, i.e., the one that will be installed if no version is provided in `spack install`. The last three lines (18 to 20) declare the required dependencies to build `pajeng`. Such dependencies should also be Spack packages and, if needed, will be automatically installed by Spack.

[1] <https://github.com/schnorr/pajeng>

PYTHON

```python
from spack import *

class Pajeng(CMakePackage):
  """PajeNG is a re-implementation of the well-known Paje
     visualization tool for the analysis of execution traces."""

  homepage = "https://github.com/schnorr/pajeng"
  git = "https://github.com/schnorr/pajeng.git"
  url = "https://github.com/schnorr/pajeng/archive/1.3.6.tar.gz"

  maintainers = ['viniciusvgp', 'schnorr']

  version('1.3.6',
          sha256 = '1a2722bfaeb0c6437fb9e8efc2592edbf14ba01172f9
              7e01c7839ffea8b9d0b3',
          preferred = True)
  version('develop',
          git = 'https://github.com/schnorr/pajeng.git')

  depends_on('boost')
  depends_on('flex')
  depends_on('bison')
```

From this point, it is now possible to run `spack install pajeng`. Editions in packages' receipts, both user-created or existing ones, are possible through `spack edit`. For example, to look up for other releases from the base URL and update the package file, one can use `checksum`:

SH

```
spack checksum --add-to-package pajeng
```

The result is a list of several lines containing all available releases with the respective `sha256` sum. Such lines can now be appended to our initial package definition.

PYTHON

```
version('1.3.6',
  sha256 = '1a2722bfaeb0c6437fb9e8efc2592edbf14ba01172f97e01c7839ffea8b9d0b3')
version('1.3.5',
  sha256 = 'ea8ca02484de4091dcf57289724876ec17dd98e3a032dc609b7ea020ca2629eb')
version('1.3.4',
  sha256 = '284e9a590a2861251e808542663bf1b77bc2c99650a1fbf945cd5bab65402f9e')
version('1.3.3',
  sha256 = '42cf44003d238fd5c4ab512bdeb445fc12f7e3bd3f0526b389f080c84b83b19f')
version('1.3.2',
  sha256 = '97154415a22f9b7f83516e988ea664b3990377d69fca859275ca48d7bfad0932')
version('1.3.1',
  sha256 = '4bc3764aaa7e79da9a81f40c0593b646007b689e4ac20886d06f271ce0fa0a60')
version('1.3',
  sha256 = '781b8be935e10b65470207f4f179bb1196aa6740547f9f1af0cb1c0193f11c6f')
version('1.1',
  sha256 = '986d03e6deed20a3b9d0e076b1be9053c1bc86c8b41ca36cce3ba3b22dc6abca')
version('1.0',
  sha256 = '4d98d1a78669290d0a2e6bfe07a1eb4ab96bd05e5ef78da96d2c3cf03b023aa0')
```

Software evolves over time; new versions include new build options or dependencies and deprecate old ones, resulting in specific dependencies among different versions. For example, older versions of `pajeng` require `qt` versions `4.x` while the version under development needs `fmt` library. To handle these cases, we can add new dependencies that are restricted to some versions:

PYTHON

```
depends_on('qt@:4.999+opengl', when='@:1.3.2')
depends_on('freeglut', when='@:1.3.2')
depends_on('fmt', when='@develop')
```

Note that we use the Python colon (`:`) syntax to deal with ranges of versions, e.g., `depends_on('qt@:4.999+opengl', when='@:1.3.2')` means `pajeng` up to version `1.3.2` depends on `qt` library up to version `4.99` with `opengl` variant enabled.

Packages usually have some build options allowing users to enable or disable features, e.g., passing `cmake` or `configure` parameters. To illustrate this capability, we add

package variants to enable static linking, documentation, and to disable the building of `libpaje` and auxiliary tools:

PYTHON

```
variant('static',
        default = False,
        description = "Build as static library")
variant('doc',
        default = False,
        description = "The Paje Trace File documentation")
variant('lib',
        default = True,
        description = "Build libpaje")
variant('tools',
        default = True,
        description = "Build auxiliary tools")

def cmake_args(self):
  args = [
    self.define_from_variant('STATIC_LINKING', 'static'),
    self.define_from_variant('PAJE_DOC', 'doc'),
    self.define_from_variant('PAJE_LIBRARY', 'lib'),
    self.define_from_variant('PAJE_TOOLS', 'tools')
  ]
  return args
```

Note the use of the `default` parameter to set or unset variants by default. To become effective, the new variants must be transposed to the build options expected by the package building system during compilation and install. Since `pajeng` is built with `cmake`, we have to override the `cmake_args` method to write the proper variables (e.g., `PAJE_DOC`).

The addition of variants and version-specific dependencies introduces a new issue, e.g., incompatible options. The only solution in such cases is to prevent the installation and tell the user the reason. For example, previously, we defined two variants for pajeng: `lib` and `tools`. While requiring the installation of `pajeng+lib~tools` is completely valid, the contrary, i.e., `pajeng~lib+tools`, is inconsistent since building the auxiliary tools (`+tools`) depends on building the library (that was disabled with `~lib`). Besides handling contradictory package options, the same feature is useful for dealing with known bugs or declaring incompatibilities with a given compiler or dependency.

PYTHON

```
conflicts('+tools',
          when = '~lib',
          msg = "Enable libpaje to compile tools.")
```

4.2.7. Publishing a new package

There are two ways to make a new Spack package public. The first one, and the more comprehensive, is submit it to the official repository. At the time this chapter was written, this repository stores more than 7000 packages. Submissions of new packages can be done with pull requests. Once the package passes by the automated tests, it will be available for installation by any user. A complete version of the `pajeng` package, following the steps listed in this chapter, was submitted to the Spack official repository and be found at https://github.com/spack/spack/blob/develop/var/spack/repos/builtin/packages/pajeng/.

The second approach to publishing a new package is through the creation of additional public or private repositories. Such repositories can be created with `spack repo create` and later shared with other users who can add them to their local instances by running `spack repo add`. Additional repositories are useful for software whose access is restricted or for Spack packages in development that are not yet mature enough to be committed to the official repository. Some institutions maintain external and public Spack repositories to publicize and facilitate the installation of *software* developed by them. Another use case for additional repositories is when it is necessary to override a package from the official repository for some reason.

4.3. Part II – Guix

Guix (COURTÈS; WURMUS, 2015; GNU, 2023a; GNU, 2023b) provides user-space commands to automate package builds with a strong focus on reproducibility. The building process executes in a controlled environment to isolate details about host and user configuration. Guix follows Nix (DOLSTRA; JONGE; VISSER, 2004) strategy and makes the build be executed by a daemon on a Linux kernel container. This way, all package builds execute with a dedicated Guix user and with a restricted set of environment variables containing only explicitly declared dependencies. Built packages reside on a system-wide directory (e.g., `/gnu/store/`) and can be shared among different users. Each install is identified by a hash computed using package source code, compiler, libraries, scripts, and dependencies. When a user requests a package install that matches an existing hash, the build is skipped, and Guix only links the files on the user directory, avoiding duplicated builds. Figure 4.2 shows the DAG for Openblas at Guix.

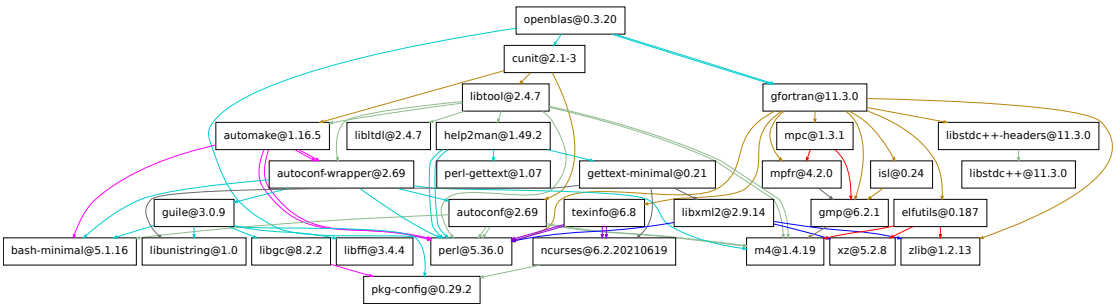

Figure 4.2. Dependency graph of OpenBLAS at Guix

4.3.1. Installation

Unlike Spack, Guix requires administrator rights to be installed. On a Debian-like Linux, one can run[2]:

SH

```
sudo apt install guix
```

From this command, all package install and administration directives can run entirely in user space. Alternatively, in systems where the `guix` daemon itself is not installed and the install cannot be requested, `guix pack` can be used with some limitations (see Section 4.3.4). Before executing install commands, Guix recommends that every user runs some bootstrap configuration for locales:

SH

```
guix install glibc-locales
export GUIX_LOCPATH="$HOME/.guix-profile/lib/locale"
GUIX_PROFILE="$HOME/.guix-profile" source "GUIX_PROFILE/etc/
   profile"
```

4.3.2. Installing packages

The first step when installing a new package is to check the list of all available packages with `guix package --list-available`. At the time this chapter was written, it reported more than 25000 packages. Since the database of packages is huge, a better option is to search using a pattern. For example, to get a filtered list of all packages starting with "ascii" in their synopsis or description fields, one can run:

SH

```
guix package --search="^ascii*"
```

Or to restrict the search to the package name:

SH

```
guix package --list-available="^ascii"
```

STDOUT

```
ascii           3.18    out     gnu/packages/shellutils.scm:67:2
ascii2binary    2.14    out     gnu/packages/textutils.scm:415:2
asciidoc        9.1.0   out     gnu/packages/documentation.scm
   :110:2
asciinema       2.3.0   out     gnu/packages/terminals.scm:226:2
```

[2]Commands presented here consider Guix version 1.3.0

Once the exact package name is known, the installation can proceed with:

SH

```
guix install asciidoc
```

Option `package --list-installed` lists all the installed packages with their respective version and path. For example:

SH

```
guix package --list-installed
```

STDOUT

```
glibc-locales   2.35    out     /gnu/store/03
   v1svhv6wj9pd6awpdi5zn4wd31b23f-glibc-locales-2.35
asciidoc        9.1.0   out     /gnu/store/91
   h31gnblv2jgqx7majqxa4wvjyr0ax5-asciidoc-9.1.0
```

Software evolves with time, and maybe you want to check if there are new releases for installed packages:

SH

```
guix pull
guix upgrade
```

You can remove `asciidoc` package with:

SH

```
guix remove asciidoc
```

4.3.3. Using a package

Once a package was installed, you might want to use it to develop another software. Guix provides `guix shell` to set-up a proper environment with dependencies and environment variables. For example, to start a shell to code something using `openblas` as a library:

SH

```
guix shell --development openblas gcc
```

Another solution is creating a new directory populated with symbolic links to mimic the traditional UNIX tree structure with `bin/`, `lib/`, and `include/`. For example, to compile an `openblas` code:

SH

```
guix install openblas --profile=$HOME/blas-dir
gcc blas-example.c -Lblas-dir/lib -Iblas-dir/include -lopenblas
   -o ex-openblas
LD_LIBRARY_PATH=blas-dir/lib ./ex-openblas
```

4.3.4. Sharing configs

There are two approaches to exporting a Guix setup to another machine. If both machines have Guix, the first way is to use the `guix archive` command that packs a list of packages into a single file. For example:

SH

```
# on machine A
guix archive --export openblas gcc > foo.nar
# on machine B
guix archive --import foo.nar
```

The second strategy allows to export software to a machine without Guix. The command `guix pack` combines a list of packages in a standalone binary tarball that can be extracted on the other machine with standard user-space commands.

SH

```
# on machine A
guix pack openblas gcc
# on machine B
tar xf pack.tar.gz
```

4.3.5. Preparing a new package

Guix packages, or "package definitions" in `guix` jargon, are written in GNU Guile, an implementation of the Scheme language, which itself is a dialect of Lisp. Once again, we use the `pajeng` tool to illustrate a package creation:

SCHEME

```scheme
(use-modules
 (guix packages)
 (guix download)
 (guix build-system cmake)
 (guix licenses)
 (gnu packages boost)
 (gnu packages bison)
 (gnu packages flex)
)
(package
 (name "pajeng")
 (version "1.3.6")
 (source
   (origin
      (method url-fetch)
      (uri
        (string-append
          "https://github.com/schnorr/pajeng/archive/refs/tags/"
          version
          ".tar.gz"))
      (sha256
         (base32
          "1cyhp6lgx7w3qw0pxybj26h4pwfv5rcw5vz8p5zl7imhmszj49qs"
          ))))
 (build-system cmake-build-system)
 (arguments
  `(#:tests? #f
    #:validate-runpath? #f)
 )
 (inputs (list
           boost
           bison
           flex
         ))
 (synopsis
    "PajeNG: library and associated tools for Paje trace files")
 (description
    "PajeNG is a re-implementation of the well-known Paje
    visualization tool for the analysis of execution traces.
    PajeNG comprises the libpaje library, and an auxiliary tool
    called pj_dump to transform Paje trace files to
    Comma-Separated Value (CSV). The space-time visualization
    tool called pajeng had been deprecated (removed from the
    sources) since modern tools do a better job (see
    pj_gantt).")
 (home-page "https://github.com/schnorr/pajeng")
 (license gpl3))
```

The first 9 lines load required modules, i.e., the Scheme equivalents of libraries or packages. Lines 2 to 5 refer to Guix internal modules, while lines 6-9 load `pajeng` dependencies. The package definition itself starts on line 10 with package name, version, and tarball download instructions. Lines 23 to 27 specify the `cmake` building system. We disable tests and validate-runpath phases due to incompatibilities between `pajeng` building configurations and Guix expected ones. Lines 28 to 32 declare package dependencies. Finally, lines 33 to 40 contain package synopsis and description texts, which are presented to the user in commands as `guix show` and `guix search`. The last two lines define the package home-page and license.

From this point, one can try to build and install the created package with:

SH

```
guix build --file=/path-to-created-file/pajeng.scm
guix package
   --install-from-file=/path-to-created-file/pajeng.scm
```

In Guix, working with different version numbers or compilation options requires creating a new package. Fortunately, it is possible to rely on inheritance to extend from an existing package. For example, to create a new package for `pajeng` building from its git repository (at a given commit):

SCHEME

```
(let ((commit "ca24c95cc5b4e53455058e180f01d5a5febccac6")
      (revision "1"))
(package (inherit pajeng)
  (name "pajeng")
  (version (git-version "1.3.6" revision commit))
   (source
     (origin
       (method git-fetch)
       (uri (git-reference
         (url (string-append
                "https://github.com/schnorr/"
                name))
                   (commit commit)))
       (sha256
         (base32
          "1c1ggfgdl5xxq8jkvf774440j5lyn37n8qll354d4lbqxm81v9av")
        )
      )
    )
   (inputs (modify-inputs (package-inputs pajeng)
                     (prepend fmt)))
  )
)
```

Inheritance is also helpful in building with different compilation options, which again requires creating a new package:

SCHEME

```
(package (inherit pajeng)
   (name "pajeng-doc")
   (build-system cmake-build-system)
   (arguments
     '(#:configure-flags '("-DPAJE_DOC=ON")))
   (inputs (modify-inputs (package-inputs pajeng)
                          (prepend texlive-scheme-basic
                             ghostscript poppler asciidoc
                             texlive-titlesec texlive-setspace
                             texlive-listings
                             texlive-gsftopk)))
)
```

4.3.6. Publishing a new package

Guix provides three ways to make a new package public. The first and simpler one is just make available the `scm` file of a package. After obtaining it, one can just run `guix package`, passing the file to the option `--install-from-file`. The `scm` files created here for `pajeng` can be downloaded from:
<https://exp-hpc.gitlab.io/wscad-spack-guix-2023>.

The second option is to create a Guix channel, which is just a Git repository containing one or more `scm` files providing Guix package definitions. The `scm` files stored in the repository should have some slight adjustments from the above examples due to some Guix idiosyncrasies. The final step is to add the new channel to the Guix user local configuration by including the following lines to `~/.config/guix/channels.scm`:

SCHEME

```
(cons
  (channel
   (name wscad-mc-2023)
   (url
      "https://gitlab.com/exp-hpc/guix-channel-2023-wscad.git"))
  %default-channels)
```

The last way is to push the contributions to the official Guix repository directly. One can checkout `https://git.savannah.gnu.org/git/guix.git` and commit the `scm` file of the new package. This is the recommended method for mature packages that fully comply with the coding style and contributing rules of Guix.

4.4. Conclusion

This short course covered the fundamental basic concepts for managing software packages at the user level. Facilitating experimental reproducibility, a paramount topic in general research activities, we present the Spack and the Guix tools. We hope that the general goal provided in this short course will cause a positive involvement of new students working in High-Performance Computing and the importance of keeping track of their software versions along their experiments. We expect that these tools provide a more rigorous data collection and observation methods for new researchers. As examples of Spack and Guix usage, we aggregated the code snippets presented in this text in a companion material which is available at: <https://exp-hpc.gitlab.io/wscad-spack-guix-2023/>.

Acknowledgments

We would like to thank Jessica Imlau Dagostini which was our coauthor in a first version of this chapter, prepared in Portuguese, for the *Escola Regional de Alto Desempenho* 2021 (DAGOSTINI et al., 2021) covering the Spack instructions. We also would like to thank the developers and the community of Spack and Guix.

References

COURTÈS, L.; WURMUS, R. Reproducible and User-Controlled Software Environments in HPC with Guix. In: *Euro-Par 2015: Parallel Processing Workshops*. Cham: Springer International Publishing, 2015. p. 579–591. ISBN 978-3-319-27308-2.

DAGOSTINI, J. I.; PINTO, V. G.; NESI, L. L.; SCHNORR, L. M. Are you root? Experimentos Reprodutíveis em Espaço de Usuário. In: CHARÃO, A.; SERPA, M. (Ed.). *Minicursos da XXI Escola Regional de Alto Desempenho da Região Sul*. Porto Alegre: Sociedade Brasileira de Computação - SBC, 2021. cap. 3, p. 70–87. ISBN 9786587003504.

DOLSTRA, E.; JONGE, M. de; VISSER, E. Nix: A safe and policy-free system for software deployment. In: *Proceedings of the 18th USENIX Conference on System Administration*. USA: USENIX Association, 2004. (LISA '04), p. 79–92.

GAMBLIN, T. et al. The Spack package manager: bringing order to HPC software chaos. In: *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. [S.l.: s.n.], 2015. p. 1–12.

GNU. *GNU Guix Cookbook*. 2023. Disponível em: <https://guix.gnu.org/cookbook/en/guix-cookbook.html>.

GNU. *GNU Guix Reference Manual*. 2023. Disponível em: <https://guix.gnu.org/manual/en/html_node/index.html>.