Working with Multiple Package Managers

A whitepaper to Collegeville 2020, articulating impediments, challenges, opportunities or potential solutions to developer productivity for scientific software.

June 30, 2020

Axel Huebl
ATAP Division
Lawrence Berkeley National Laboratory
Berkeley (CA), U.S.A.
axelhuebl@lbl.gov

***Abstract*—User-level package managers become increasingly popular productivity tools for end-users and developers alike. While most package management solutions are in active development and some try to reach across islands of individual programming languages and communities - plenty of room persists for choosing one individual solution over another in daily tasks. Consequently, many software maintainers decide to meet potential recipients (users and developers) half way by supporting multiple solutions themselves. This whitepaper demonstrates a potential workflow for efficient usage of multiple, user-level package managers and how to avoid common pitfalls.**

***Index Terms*—software maintenance, software reusability, sustainable development, productivity**

I. Introduction

The vast majority of software packages that computational scientists and engineers directly interact with can be installed and executed with the permissions of regular user accounts. Such software is developed at various locations around the world and might release new versions rapidly, compared to traditional software release cycles. Consequently, package management software that automates tedious installation steps, tracks installed versions, finds compatible solutions and updates packages, are popular with developers and end-users alike.

These so-called user-level package managers are often designed with specific audiences and workflows in mind, some provide software for specific programming languages while others try to provide a more language-agnostic experience. As new developers usually grow into one community (such as Python and data science) or another, they will get in touch with at least one of those package management solutions. As their experience grows, the influence and interaction between libraries (from various sources) increases and so does the demand for flexible package management solutions.

At this point, reporting from our subjective experience, many developers and users find themselves confused with typical pitfalls when switching between package managers. This white paper aims to provide a few simple guidelines for coordinating an installation of multiple modern package managers.

Supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration).

II. Choice of Package Manager

Unfortunately, an overview and fair comparison of today's existing user-level package managers is infeasible for the scope of this work. Instead, we exemplify a few popular solutions and rethink workflows and potential motivation when choosing a package manager.

A few common criteria that people evaluate their package managers against are ease of use, the provided number of (working) packages for a problem set and, especially important in automated workflows such as continuous integration, time to installed solution. Mixed with background and experience, the choice for one package manager or another is done by the "downstream" consumer of a software product. As software maintainers ("upstream"), once can guide this choice by providing direct support for modern solutions. Yet in the end, software packaging and usage is a process of communication, management of expectations and maintainability for individual communities.

A. End-User Objectives

An end-user of a software project, which shall be herein defined as a person that uses software such as an application or framework - but does not extend it, is primarily interested in a working solution. In some but not all cases, optimal runtime performance of an installed software package can be relevant. Yet in many other use cases performance is secondary compared to other objectives, such as correctness of solution, specific functionality, flexibility, etc.

If one finds oneself in the position of an end-user, we found the following approach productive. One can simplify computing workflows by keeping individual aspects separate with respect to software environments. For example, building a large-scale, parallel Fortran/C++ simulation stack can be managed within a software environment provided by a package manager. Post-processing workflows, such as analyzing and plotting data with Python packages, do not need to be solved within an environment that includes the aforementioned simulation stack. Consequently, using a differing software environments or even completely different package managers can be adequate. This is also of practical relevance, since in most cases a fully consistent

solution to even those two workflows does not exist, e.g. due to conflicting constrains and limitations in some software packages to abstract and hide non-transitive dependencies.

B. Developer Objectives

A potentially beneficial mindset is to imagine software developers and maintainers to some extend as power-users. As (power-)users, there is no reason to spend significantly more time and effort for a software environment setup than the time and effort spent by end-users. This viewpoint is usually also appropriate in computational sciences: a domains scientist might use an existing software package first, considers it useful and then decides to modify and improve it to their needs.

Developer workflows have therefore the same basic requirements as users. Furthermore, developers might build software with various feature combinations, need to debug for correctness and performance, need to modify the installed solution, combine it with further software, often need to link against application binary interfaces (ABI), etc..

At this point, many people realize that some package managers describe one or the other aspect better than others. Some package managers are solely end-user focused, others expect significant additional hops for developers, again others manage both classes of objectives well but are slow to install a software environment solution. Efficient developers might cherry-pick one or another user-level package manager depending on the workflow at hand. For the rest of this document, we will focus on workflows relevant to macOS/Linux.

III. Interplay

Generally speaking, mixing user-level package managers can quickly generate complexities that lead to unstable software environments. This is unfortunate, since users of package management software experience those the moment they install such a product.

The reasons for this outcome are manifold and often rooted in missing isolation. For example, some solutions provide their own set of compilers, specific flags and standard libraries, bootstrapping all software (e.g. conda and the conda-forge respositories [1]), while others build on system compilers and vendor-deployed "userland" libraries (e.g. Homebrew [2]), while again others bootstrap the whole software stack from *any* compiler and standard library onward (e.g. Spack [3]).

Some package managers support registering externally provided dependencies of pre-installed software. While this is a valuable functionality in certain use cases, e.g. on HPC systems with highly tuned system-provided software or in well-controlled continuous integration environments, this generally complicates the situation by putting the responsibility to track those dependencies back on the user of package management software.

As in the previous section, it is potentially best to use the management tool that can get a certain aspect of a workflow done at once (see mentioned examples above). In particular, most user-level package managers expose their installed packages by setting environment variables, such as compilers hints (`CC`, `CXX`), tools (`PATH`, ...) and libraries (`CMAKE_PREFIX_PATH`, `LD_LIBRARY_PATH`, `PYTHONPATH`, ...). Avoiding "activation" of multiple user-level package managers at the same time avoids most of the problems to begin with.

IV. A Practical Example

As a practical example, one can log into the same computing machine multiple times in parallel by just using multiple terminals. With each terminal, one executes one workflow with one software environment and another, potentially even related workflow, with another environment.

Activation time of a user-level package manager, especially when written in a scripting language such as Python, can be significant (multiple seconds). As this activation occurs on every login and every newly opened terminal, removing auto-activation lines in `.bashrc`/`.profile` files in one's `$HOME` directory is beneficial.

A. A Subjective Selection of Package Managers

Taking the **conda** [1] package manager as an example, one should avoid the command `conda activate` that triggers `conda init`. Unlike other package managers that only try to "install" themselves into the aforementioned files once, conda will re-try this on those commands and load its base environment in every new terminal. A simple `source activate` provides the same functionality.

In Python, the popular Python Package Installer (**pip**) [4] is not (yet) a package and dependency manager, but actually a mere package installer (as the name suggests) without sophisticated environment solution capabilities. Pip installed software can be isolated well by placing it in virtual environments [5]. Installing pip software on top of a conda environment (or other package managers) often works for pure Python packages, given that one does not change the "parent" software environment anymore (including updates) after installing anything with pip.

Brew [2] is a popular package manager on macOS that also runs on Linux. Most of its popularity is gained fast install of pre-compiled packages. It reserves itself the system-wide `/usr/local` directory for linking active packages,[1] which interferes with other package managers. (`brew unlink <formula>` deactivates packages.) Compiler support is limited to the system compiler and selected versions of GCC (sometimes mixed).

Spack [3] is a popular package manager in high-performance computing and very flexible for developer workflows. The Spack project is a comparatively young, yet already provides support for features such as binary-variants, solving and tracking of compatible software, micro-architecture tuning, etc. to name a few features. Its rapid development process poses a challenge for generation and delivery of binary artifacts, which results as of today in long installation times due to frequent recompilation from source. Nonetheless, continuously generated binary caches are planned and/or near completion.

[1] From a brew perspective, this is a convenient choice to avoid binary relocation issues between paths on builder and user machines.

The programming language **Rust** [6] provides related tooling and package management as part of its core functionality. We list it here as another example of user-level package management.

Last but not least, we will add an example on the **Emscripten SDK** [7], which is not a package manager but a relatively complex compiler framework for compiling C/C++ code to WebAssembly.

B. Manually Activating Package Managers

As motivated earlier, manual activation of user-level package managers can be beneficial for explicit control of environment compatibility in complex computing and development workflows.

In the below listing, all user-level package managers were manually installed according to their documentations into directories below `$HOME/src`. We write a simple Bash script `$HOME/bin/impl-activate-env` with the following logic:

```
case "${1}" in
    conda)
        export PATH="$HOME/src/miniconda3/bin:${PATH}"
        conda_env=${2:-base}
        source activate ${conda_env}
        if [ "${conda_env}" == "base" ]; then
            conda info --envs
            echo "Activate via: \"conda activate <env>\""
            echo "New environments:"
            echo "  conda create -n py36 python=3.6 anaconda"
            echo "  conda create -n openpmd-api -c conda-forge openpmd-api"
        fi
        ;;
    brew)
        eval $($HOME/src/brew/bin/brew shellenv)
        ;;
    spack)
        . $HOME/src/spack/share/spack/setup-env.sh
        spack env list
        echo "Activate via: \"spack env activate <env>\""
        ;;
    rust)
        export PATH="$HOME/.cargo/bin:${PATH}"
        ;;
    emsdk)
        source $HOME/src/emsdk/emsdk_env.sh --build=Release
        export CC=$(which emcc)
        export CXX=$(which em++)
        echo -n "Reminder: -DCMAKE_TOOLCHAIN_FILE=$HOME/src/"
        echo    "emscripten/cmake/Modules/Platform/Emscripten.cmake"
        ;;
    *)
        echo "Usage: $0 {conda [env]|emsdk|rust|spack|brew}"
esac
```

Listing 1: `$HOME/bin/impl-activate-env`

One can "source" the above script into a Bash shell session with `source $HOME/bin/impl-activate-env <arguments>`. Arguments are plotted to the terminal when called without (or with unsupported) arguments. In the case of conda, an additional sub-argument can be specified to re-activate an already created conda environment from prior sessions.

Finally, for increased productivity, we register a bash helper-function `activate-env <arguments>` by adding a few additional lines to the file `$HOME/.bashrc`. After opening a new terminal, one can now activate a package manager with `activate-env <conda [env]|emsdk|rust|spack|brew>` This additional level of explicit activation provides a per-

fect level of control over user-level package managers and their environments, while saving terminal and login startup time by avoiding unnecessary `source` activation calls. The `$HOME/.bashrc` snippet reads as follows:

```
# register a bash function that calls the script above
activate-env () {
  . $HOME/bin/impl-activate-env $@
}

# bash completion for the above function
_activate-env()
{
    local cur prev names
    cur="${COMP_WORDS[COMP_CWORD]}"
    prv="${COMP_WORDS[COMP_CWORD]}"
    names="conda emsdk rust spack brew"                 # all primary options
    if [ "${COMP_WORDS[1]}" == "conda" ]; then          # sub envs: conda
        local cenvs=$(ls $HOME/sw/miniconda3/envs)
        COMPREPLY=( $(compgen -W "${cenvs}" -- ${cur}))
        return 0;
    fi
    if [ ${COMP_CWORD} -gt 1 ]; then return 0; fi       # no other sub
    if [ "${cur}" == "" ]; then                         # no arg
        COMPREPLY=(${names})
    else                                                # during arg eval
        COMPREPLY=($(compgen -W "${names}" -- ${cur}))
    fi
}
complete -F _activate-env activate-env
```

Listing 2: A snippet for `$HOME/.bashrc`

V. Alternative Implementations

The last section exemplifies how a one-liner at the beginning of a workflow, e.g. when opening a new terminal or logging into a computing machine, can achieve a high level of control for user-level package management. There are without a doubt more gentle implementations possible, e.g using environment modules [8], [9].

VI. A Note on Applications

Contrary to all warnings above, well-written executables are pretty robust when used in mixed workflow environments. Especially popular developer tools such as CCache, CMake, etc. come stand-alone in the sense that they properly link and hint their runtime dependencies - even when switching software environments. Also, when building a new software with them they do *not* become a runtime dependency (contrary to e.g. shared libraries such as MPI). One can just update and activate them from whatever distribution, package manager or container that appears handy.[2]

[2]The first executable found in the list of directories in `PATH` will be taken.

VII. Summary

In summary, users and developers will use a package management that they understand and that delivers fast and reliable results for their current workflow. Frustration can be avoided by following a few simple rules when trying new user-level package mangers, such as activating one package provider at a time. Explicit activation is generally more stable than implicit activation - and it is worth demoting some of the self-activating tools such as brew and conda to achieve that goal. A simple implementation of such a workflow in pure bash was shown here.

Acknowledgment

We would like to thank the communities of Spack, Conda-Forge and Homebrew for effective productivity enhancement in data science and HPC workflows. A. H. is a co-maintainer of Spack and declares the competing interest of being able to express all his development, user support, and HPC deployment workflows with it. We are grateful for the communities of PIConGPU and WarpX that provided inspiration for this report.

A. H. receives support by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department

of Energy organizations (Office of Science and the National Nuclear Security Administration).[3]

REFERENCES

[1] Conda community and Anaconda Inc., "Conda version 4.8.2," 2020. [Online]. Available: https://www.conda.io

[2] Brew community, "Homebrew version 2.4.2," 2020. [Online]. Available: https://www.brew.sh

[3] 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 *SC15: International Conference for High-Performance Computing, Networking, Storage and Analysis*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2015, pp. 1–12. [Online]. Available: https://www.spack.io

[4] Python Packaging Authority: Pip Developers, "Pip version 20.1.1," 2020. [Online]. Available: https://pip.pypa.io

[5] Python Packaging Authority: Virtualenv Developers, "Virtualenv version 20.0.25," 2020. [Online]. Available: https://virtualenv.pypa.io

[6] N. D. Matsakis and F. S. Klock, "The rust language," in *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, ser. HILT'14. New York, NY, USA: Association for Computing Machinery, 2014, p. 103–104. [Online]. Available: https://www.rust-lang.org

[7] A. Zakai, "Emscripten: An llvm-to-javascript compiler," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, ser. OOPSLA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 301–312. [Online]. Available: https://emscripten.org

[8] J. L. Furlani, "Modules: Providing a flexible user environment," in *Proceedings of the fifth large installation systems administration conference (LISA V)*, Sep 1991, p. 141–152. [Online]. Available: http://modules.sourceforge.net

[9] R. McLay, K. W. Schulz, W. L. Barth, and T. Minyard, "Best practices for the deployment and management of production hpc clusters," in *State of the Practice Reports*, ser. SC'11. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: https://lmod.readthedocs.io

[3]Phew, the last chance to link this: https://xkcd.com/1987/