Multi-package development at Fermilab with Spack

Kyle Knoepfel[1,*]

[1]Fermi National Accelerator Laboratory

Abstract. The Spack package manager has been widely adopted in the supercomputing community as a means of providing consistently-built on-demand software for the platform of interest. Members of the high-energy and nuclear physics (HENP) community, in turn, have recognized Spack's strengths, used it for their own projects, and even become active Spack developers to better support HENP needs. Code development in a Spack context, however, can be challenging as the provision of external software via Spack must integrate with the developed packages' build systems. Spack's own development features can be used for this task, but they tend to be inefficient and cumbersome.

We present a solution pursued at Fermilab called MPD (multi-package development). MPD aims to facilitate the development of multiple Spack-based packages in concert without the overhead of Spack's own development facilities. In addition, MPD allows physicists to create multiple development projects with an interface that insulates users from the many commands required to use Spack well.

1 Spack adoption at Fermilab

Spack [1, 2] has become a preferred package-management technology used within the high-performance computing (HPC) community. Its use has become so widespread that Spack is now included as part of the High-Performance Software Foundation [3]. It has also attracted the attention of HEP software professionals charged with maintaining software stacks containing hundreds of disparate, yet interoperable, software packages. Spack is designed to satisfy such needs, albeit originally for HPC contexts.

Fermilab's experiment-support efforts have selected Spack to manage software for various reasons:

1. constraints on development effort prevent the continued maintenance of UPS, the Fermilab-specific package manager [4],

2. Fermilab and its experiments can leverage the portion of Spack's support base that exists outside of the HEP community,

3. adopting a well-known technology provides an opportunity to engage with and influence the broader computing community.

*e-mail: knoepfel@fnal.gov

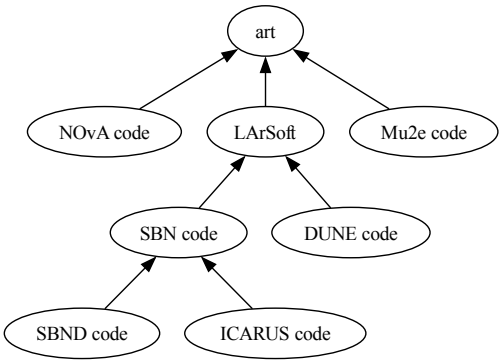

Figure 1. Dependency graph of *art*-using software projects. Each arrow points from one project to the software projects upon which it depends (e.g. ICARUS code depends on SBN code).

Since Fermilab's first Spack report at a CHEP conference [10], adoption has progressed substantially from proof-of-principle toy projects to full Spack installations of all offline code for DUNE, Mu2e, and other experiments. Achieving this has required establishing a process to create layered releases (in the form of Spack environments) of Fermilab-supported software. In addition, we have created a solution for developing multiple CMake packages together in a Spack context; this multi-package development solution is the focus of this document.

2 Developing multiple packages together

To enable development at scale, experiment code is often factorized into pieces that can be developed with some degree of isolation. This factorization, however, rarely results in completely independent bodies of code but in a directed graph of software dependencies. Figure 1 shows a simplified directed graph of *art*-using [5, 6] software projects.

It is common for an experiment to develop their own code at the same time as adjusting a piece of shared software. For example, a developer of SBN software [7] may need to adjust some code within LArSoft [8]. To facilitate coordinated development of these packages, Fermilab has provided the multi-repository build (MRB) system [9], which builds multiple CMake-based projects together as a single larger CMake project. One notable feature of MRB is that it is intended to support efficient *incremental builds*, where developers test frequent, gentle changes to the software. MRB has been widely used by Fermilab experiments, but it relies heavily on the Fermilab-specific UPS package management system. A different approach is thus required to accommodate Fermilab's migration from UPS to Spack.

3 Code development using Spack

Fermilab developers have explored various options for replacing MRB. One approach was SpackDev [11], which was presented to experiments that were not yet ready to consider a Spack-based development approach. Since then, Spack has added a development facility (i.e. the `spack develop` subcommand) that supports the development of any Spack package in a way that integrates cleanly with Spack's existing installation infrastructure. The natively-

Table 1. Commands supported by the Spack MPD extension. Each subcommand below follows the prefix `spack mpd` on the command line. The subcommands shaded brown are demonstrated in Section 4.1.

Subcommand	Description	
`clear`	Clear selected MPD project	
`new-project (n)`	Create MPD development project	
`refresh`	Refresh project	*Project*
`rm-project (rm)`	Remove MPD project	
`select`	Select MPD project	
`build (b)`	Build repositories	
`git-clone (g, clone)`	Clone git repositories	
`install (i)`	Install built repositories	*Development*
`test (t)`	Run tests	
`zap (z)`	Delete everything in your build and/or install areas	
`list (ls)`	List MPD projects	*Usability*
`status`	Current MPD status	
`init`	Initialize MPD on this system	

provided Spack development facility, however, requires more Spack expertise of its users, and substantial inefficiencies were encountered early on for incremental builds[1].

The approach adopted by Fermilab was to pursue a Spack extension (i.e. a `spack mpd` subcommand) for multi-package development. Spack MPD [12] is intended to be an MRB-like system tailored for iterative algorithm development with Spack providing the dependencies of the software under development.

4 Spack MPD

Table 1 lists the subcommands supported by Spack MPD, grouped according to MPD *project* management, *development* subcommands, helper subcommands to aid in *usability*, and the `init` subcommand, which is invoked only once per Spack installation. Each subcommand has additional options, which can be printed to the terminal by providing the `--help` option (e.g. `spack mpd clear --help`).

As with MRB, one of the goals of MPD is to support core development activities while minimizing the user's required knowledge of the underlying package delivery system. These core functionalities are shaded brown in Table 1 and are described in Section 4.1 as part of a typical development workflow[2]. However, there are additional functionalities that serve as improvements wrt. MRB:

- After setting up the Spack environment, an MPD project is selected for development by invoking `spack mpd select`. (Establishing an MRB development session often involved invoking more than one setup script.)
- The `spack mpd select` subcommand also makes it easy to switch to a different development project in the same shell. (This was not generally possible with MRB due to its reliance on environment variables.)

[1] As of this writing, Spack (by default) installs each package serially, only taking advantage of parallelism within the building and installation of each package. For Fermilab users, this can be expensive when building software on shared machines.

[2] Subcommands not described in this document are documented at the Spack MPD GitHub repository [12].

```
$ spack mpd new-project --name my-art-devel -T my-art-devel -E gcc-14-1 cxxstd=20 %gcc@14

==> Creating project: my-art-devel

Using build area: /scratch/knoepfel/my-art-devel/build
Using local area: /scratch/knoepfel/my-art-devel/local
Using sources area: /scratch/knoepfel/my-art-devel/srcs

==> You can clone repositories for development by invoking

  spack mpd git-clone --suite <suite name>

  (or type 'spack mpd git-clone --help' for more options)
```

Figure 2. Creating a new MPD project that inherits dependencies from an already existing `gcc-14-1` environment. Using the `-T` option creates a top-level directory that contains the build, local, and sources subdirectories. The `cxxstd=20` and `%gcc@14` specifications instruct Spack's concretizer to select dependencies for the developed packages that support C++20 and must be buildable with GCC 14.

- The `spack mpd list` subcommand prints to the terminal a list of existing projects available for development, and the `spack mpd status` subcommand gives details of the currently selected project. (Such functionality was never possible with MRB.)

These improvements were largely achieved by avoiding the use of environmental variables within the MPD infrastructure.

4.1 MPD development workflow

After initializing MPD for the installed Spack instance, a development workflow begins by creating a new project (see Figure 2). Each project must have (a) a name associated with it, (b) a source code area for hosting repositories to develop, (c) a build area where compiled libraries are created, and (d) an area of maintaining the Spack artifacts necessary for providing external software (denoted as the `local` area in Figure 2). In addition, variants and compiler specifications may be listed on the same command line to constrain Spack's concretizer when determining the required dependencies of the code under development. It is also possible, using the `-E` option, to provide one or more environments whose concretized dependencies will be used by the MPD project.

After creating the new project, MPD allows the user to clone (and optionally fork) repositories that are desired for development (see Figure 3). The cloned repositories (hereafter *developed packages*) will be placed in the source code area regardless of the current working directory when the `spack mpd git-clone` subcommand is invoked.

```
$ spack mpd git-clone --fork cetlib cetlib-except hep-concurrency

==> Cloning and forking:

  cetlib .................. done    (cloned, added fork knoepfel/cetlib)
  cetlib-except ........... done    (cloned, created fork knoepfel/cetlib-except)
  hep-concurrency ......... done    (cloned, created fork knoepfel/hep-concurrency)

==> You may now invoke:

  spack mpd refresh
```

Figure 3. Cloning and forking repositories.

```
$ spack mpd refresh

==> Refreshing project: my-art-devel

...

==> Concretizing project (this may take a few minutes)
==> Environment my-art-devel has been created
==> Updating view at /scratch/knoepfel/spack/var/.../my-art-devel/.spack-env/view
==> Concretization complete

==> Ready to install MPD project my-art-devel

==> Would you like to continue with installation? [Y/n]
==> Specify number of cores to use (default is 12)
==> Installing my-art-devel
[+] /usr (external glibc-2.34-hjl43avhawltutkgujn2ns3577kjowlq)

...

[+] /scratch/knoepfel/spack/.../intel-tbb-2021.9.0-gtkaoizm5i4m6goy7rptg7v3i5q2jrg7

==> my-art-devel is ready for development (e.g type spack mpd build ...)
```

Figure 4. Refreshing a MPD project.

To determine which dependencies are required for the developed packages, the project must be *refreshed* (see Figure 4), which usually takes a few minutes. The refresh stage performs multiple steps:

- The recipes of the developed packages are used to create an anonymous development environment of those packages' dependencies.
- A list of uninstalled dependencies required to proceed with development is printed to the terminal. The developer then has the option to directly install the packages.
- Artifacts are generated to create one CMake project that includes the developed packages.

The refresh step must be invoked only when another repository is cloned for development or whenever there are changes to a recipe of one of the developed packages.

Once the refresh step is complete, the developed packages may be built (see Figure 5). The `spack mpd build` step automatically activates the anonymous development environment that was created as part of the refresh step. With the activated environment, the CMake configuration step is then invoked, followed by the build command, which uses either GNU Make or Ninja (depending on the MPD project configuration) [13, 14]. Note that MPD uses CMake directly *and not Spack* to build the developed packages.

After a successful build, any CMake-based unit tests [15] can be executed by invoking the `spack mpd test` subcommand (see Figure 6). As in the build stage, the development environment is automatically activated by the `test` subcommand to ensure that run-time paths are properly established.

4.2 Guidance in using MPD

To develop a package using MPD, the package must have a Spack recipe and it must be a CMake package. MPD will ignore any non-CMake packages that have been cloned to the sources area.

In addition, it is highly encouraged to find alternatives to using environment variables as crucial ingredients to the building and testing stages of development. Whereas MRB

```
$ spack mpd build -j12

==> Configuring with command:

cmake --preset default /scratch/knoepfel/my-art-devel/srcs ...

Preset CMake variables:

  CMAKE_BUILD_TYPE:STRING="RelWithDebInfo"
  ...

-- Found TBB: /.../lib64/cmake/TBB/TBBConfig.cmake (found version "2021.9.0")
-- The C compiler identification is GNU 14.1.0
...
-- Configuring done (2.2s)
-- Generating done (0.2s)
-- Build files have been written to: /home/knoepfel/scratch/my-art-devel/build

==> Building with command:

cmake --build /scratch/knoepfel/my-art-devel/build -- -j12

[0/2] Re-checking globbed directories...
[278/278] Linking CXX executable cetlib/bin/ntuple_t
```

Figure 5. Building a project. The command line includes the parallelism option `-j12`, which is then passed directly to the build command (see the second CMake command in cyan).

```
$ spack mpd test -j12

==> Testing with command:

ctest --test-dir /scratch/knoepfel/my-art-devel/build -j12

Internal ctest changing into directory: /home/knoepfel/scratch/my-art-devel/build
Test project /home/knoepfel/scratch/my-art-devel/build
        Start   1: coded_exception_test
        Start   2: demangle_t
        Start   3: exception_collector_test
        Start   4: exception_test
        Start   5: exception_category_matcher_t
        Start   6: exception_message_matcher_t
        Start   7: exception_bad_append_t
        Start   8: runThreadSafeOutputFileStream_t.sh
        Start   9: assert_only_one_thread_test
        Start  10: serial_task_queue_chain_t
        Start  11: serial_task_queue_t
        Start  12: waiting_task_list_t
  1/100 Test   #1: coded_exception_test .........................   Passed    0.01 sec
...
100/100 Test  #55: cpu_timer_test ..............................   Passed    0.55 sec

100% tests passed, 0 tests failed out of 100
```

Figure 6. Testing a project. The command line includes the `-j12` parallelism flag, which is passed to the `ctest` shown in cyan.

would automatically set certain environment variables (e.g. `LD_LIBRARY_PATH`) for the convenience of the user, MPD does not do so. Spack itself does support the setting of environment variables during its own build and installation phases of software, but reducing the use of environment variables better insulates packages from each other and from the user environment.

5 Future work

Shortly after this work was presented in Krakow, it was discovered that MPD implicitly assumes each developed package uses Fermilab-specific CMake infrastructure [16]. Moving forward, we hope to generalize MPD so that more than just Fermilab efforts are supported by it. We also anticipate strengthened engagement with the rest of the Spack committee now that Fermilab has joined Spack's technical steering committee.

6 Acknowledgments

This work has been authored by Fermi Forward Discovery Group, LLC under Contract No. 89243024CSC000002 with the U.S. Department of Energy, Office of Science, Office of High Energy Physics.

References

[1] T. Gamblin, *et al*, The Spack Package Manager: Bringing Order to HPC Software Chaos, Supercomputing 2015 (SC'15), https://doi.org/10.1145/2807591.2807623

[2] https://spack.io

[3] https://hpsf.io

[4] M. Votava, *et al*, UPS UNIX Product Support, in Seventh Conference Real Time '91 on Computer Applications in Nuclear, Particle and Plasma Physics Conference Record, pp. 156–159 (1991)

[5] C. Green, *et al*, in Proceedings, 19th International Conference on Computing in High Energy and Nuclear Physics (CHEP 2012) **396**, 022020 (2012)

[6] https://art.fnal.gov

[7] https://sbnsoftware.github.io

[8] E.L. Snider and G. Petrillo, LArSoft: toolkit for simulation, reconstruction and analysis of liquid argon TPC neutrino detectors, J. Phys. Conf. Ser. **898** 042057 (2017)

[9] https://github.com/art-framework-suite/mrb

[10] C. Green, *et al*, Spack-Based Packaging and Development for HEP, EPJ Web of Conferences **214**, 05013 (2019) https://doi.org/10.1051/epjconf/201921405013

[11] C. Green, *et al*, SpackDev: Multi-Package Development with Spack, EPJ Web of Conferences **245**, 05035 (2020) https://doi.org/10.1051/epjconf/202024505035

[12] https://github.com/FNALssi/spack-mpd

[13] https://www.gnu.org/software/make

[14] https://ninja-build.org

[15] https://cmake.org/cmake/help/latest/manual/ctest.1.html

[16] T. Madlener, Problems building dependent packages, https://github.com/FNALssi/spack-mpd/issues/10 (2024)