Fermilab U.S. DEPARTMENT OF ENERGY | Office of Science

SpackDev: Multi-Package Development with Spack

Liz Sexton-Kennedy for Chris Green, Jim Amundson, Lynn Garren & Patrick Gartung

CHEP 2019

Outline

- What is Parallel Package Development and why is it important?
- SpackDev: introduction, rationale and goals.
- Digression—Spack as the underlying package manager for SpackDev.
- SpackDev: details and interaction with Spack.
- Progress and next steps.

Parallel Package Development (PPD) in HEP

- Code for HEP experiments is very library and plugin oriented: we use frameworks to connect algorithms and build a whole from many parts of different origins and organizational responsibilities, with the exact makeup of a particular application often being determined at runtime.
- Data definitions, utility code, algorithms, and framework code are logically distinct and often separated into different packages or repositories by category or subject matter.
- Interface and other breaking changes are common, often affecting many parts of a system—ABI compatibility is especially critical for C++ projects.
- Even for bug-fixing: “minimal reproducers” are often not straightforward, and occasionally not even possible.
- “The fundamental interconnectedness of all code”[1] leads to the need to develop packages in parallel.

[1] Apologies to Douglas Adams.

Characteristics of a PPD System

A PPD system supports the simultaneous development of multiple packages, including compilation, linking, testing and installation.

- Either part of or using a particular underlying build system.
- Usually utilizes (often centrally) pre-installed binaries and headers against which to develop higher-level packages locally—can interact with a release / package management system.
- Manages dependencies between packages being developed and pre-installed external packages, including versions where appropriate.
- Allows the developer to choose multiple packages to develop together.
- Often aware of and able to interface directly with source code management (SCM) systems like CVS, Subversion, or Git for checkouts, branch manipulation, *etc.*
- Enables the development cycle: code, build, test. . .

PPD Systems in the Wild

- SoftRelTools (SRT): BaBar, CDF, DZero, MiniBooNE, NOvA.
- Software Configuration, Release and Management (SCRAM): CMS.
- Multi-Repository Build (MRB): DUNE, ICARUS, LArIAT, MicroBooNE, Muon g-2, SBND, *etc.*
- Configuration Management Tool (CMT): ATLAS[2], LHCb[3], MINERvA.
- CMake-based systems: ATLAS, LHCb, LCG.

[2] Historical.
[3] Historical.

Why do we need a new PPD?

HEP build / release / package management infrastructure needs to evolve to matching changing requirements and best practice:

- Environmental constraints: need to support modern HPC systems in addition to more traditional farms, and desktop or laptop operation.
- OS based constraints (macOS SIP, `RPATH` / `RUNPATH` vs `(DY)LD_LIBRARY_PATH`).
- Tool evolution (Autotools, CMake, Make, Ninja...), innovation (Conda, Homebrew, Nix, Portage, Spack), and obsolescence (CMT, SoftRelTools, UPS).
- Ongoing coordinated effort via HEP Software Foundation Packaging Group[a].

[a] https://hepsoftwarefoundation.org/workinggroups/packaging.html

So, we need a PPD that:

- Is or works with modern build and packaging systems.
- Can be used by many experiments.
- Has applicability to modern HPC systems.

SpackDev

SpackDev is...

A PPD system based around the *award-winning*[a] Spack[b] open source multi-platform package management system.

[a] R&D 100 2019 Silver Special Recognition Award.
[b] "Package manager for supercomputers"—https://spack.io/.

Rationale and goals.

- Leverage a modern multi-platform package management system; Spack is one of the more promising subjects of HSF Packaging Group activities.
- Tighter package management system coupling $\implies$ looser build system coupling.
- Utilize package manager to handle dependencies, SCM interaction, per-package build system interaction.
- Aim for flexibility while ensuring binary consistency across the whole software system.

Digression: Spack

Spack highlights

- >3500 package “recipes” with a formalized build language.
- Sophisticated “concretizer” resolves dependency constraints (versions, variants).
- Install pre-built packages from a binary cache, or build dependencies as needed.
- Python, extensible.
- Large and active open source community (>400 contributors)[a].

[a] We are regular contributors and have been accepted into the Spack organization.

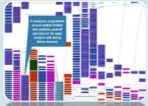
Spack recipes

- Recipes are Python classes (superclass determines and describes build system).
- Directives specify URLs, versions, patches, variants, dependencies, conflicts[a].
- Override superclass methods (`cmake_args()`, *etc.*) to describe build and install details.
- Define the build-time or runtime environment.

[a] *e.g.* package X v1.8 won’t compile with GCC <7.3

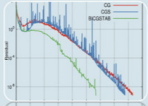
Digression: Spack

ECP software technologies are a fundamental underpinning in delivering on DOE's exascale mission



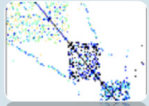
Programming Models & Runtimes

- **Enhance & prepare OpenMP and MPI programming models** (hybrid programming models, deep memory copies) for exascale
- Development of **performance portability tools** (e.g. Kokkos and Raja)
- Support alternate models for potential benefits and risk mitigation: PGAS (UPC++/GASNet) ,task-based models (Legion, PaRSEC)
- Libraries for deep memory hierarchy & power management



Development Tools

- Continued, multifaceted capabilities in portable, open-source LLVM compiler ecosystem to support expected ECP architectures, including support for F18
- **Performance analysis tools** that accommodate new architectures, programming models, e.g., PAPI, Tau

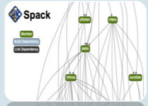


Math Libraries

- **Linear algebra, iterative linear solvers, direct linear solvers, integrators and nonlinear solvers, optimization, FFTs, etc**
- Performance on new node architectures; extreme strong scalability
- Advanced algorithms for multi-physics, multiscale simulation and outer-loop analysis
- Increasing quality, interoperability, complementarity of math libraries

Data and Visualization

- **I/O libraries: HDF5, ADIOS, PnetCDF,**
- **I/O via the HDF5 API**
- Insightful, memory-efficient in-situ visualization and analysis – **Data reduction via scientific data compression**
- Checkpoint restart
- **Filesystem support for emerging solid state technologies**.



Software Ecosystem

- Develop features in Spack necessary to support all ST products in E4S, and the AD projects that adopt it
- Development of **Spack stacks for reproducible turnkey deployment** of large collections of software
- **Optimization and interoperability of containers on HPC systems**
- Regular E4S releases of the ST software stack and SDKs with regular integration of new ST products

NNSA ST

- Projects that have both mission role and open science role
- Major technical areas: New programming abstractions, math libraries, data and viz libraries
- Cover most ST technology areas
- Open source NNSA Software projects
- Subject to the same planning, reporting and review processes

SpackDev: PPD with Spack

- Coordinated build & test for integration, or initialize an environment for rapid build & test cycles of a particular package.
- CMake-based build coordinator insulates per-package build systems: no inherent requirement on one package build system.
- Parallel build can take advantage of multiple cores.
- Configure multiple development areas using the same Spack installation and installed package pool.

SpackDev: Details

- Specify a dependency tree—“release”, with versions, variants—and packages therefrom to develop.
- Rather than being combined as a single “super package,” package builds are invoked in (automatically calculated) dependency order by the build coordinator and installed before being used by their dependents.
- Uses Spack functionality and package recipes to:
  - Calculate and install dependencies.
  - Identify missing “intermediate” development packages for checkout to maintain build consistency.
  - Determine configuration and build instructions from package recipes.
  - Determine the optimum build order for development packages.
  - Check out required packages for development from SCM systems if necessary.

SpackDev: Workflow

- Set up an area to develop the specified packages:

  ```
  spack dev init [--dag <install-spec-file>] <package>...
  ```

  Dependencies will be found and/or installed from the currently-configured Spack setup, and packages will be checked out if necessary.

- Configure the current environment and build all checked-out packages together:

  ```
  spack dev build
  ```

- Start a subshell with the correct environment to execute a rapid development cycle for one package:

  ```
  spack dev build-env --cd <package>; make; ctest
  ```

SpackDev: Progress So Far

- Demonstrator[4] with bootstrap Spack installation.
- Successfully demonstrated parallel development of selected LArSoft packages or the full set —a suite of ~20 interdependent packages with >100 external dependencies.
- Currently supports development of CMake-based packages (not an inherent limitation).
- Currently recipes must not override Spack's default build, install or test functions for that build type (still more permissive than existing PPDs, "Packages must be built with this build system only").
- Initializing a new development area can be slow (a few minutes for Spack to concretize fully a system of >100 packages). Generally negligible compared to dependency build time (see also upcoming Spack improvement, next slide).
- Significant speed-up from early versions by refactoring as a Spack extension command (recent Spack feature) *vs* a Python application using Spack as an external command (multiple concretization steps).

[4] https://cdcvs.fnal.gov/redmine/projects/spack-planning/wiki#The-MVP-1aLArSoft-Edition

Next Steps

- Support other build types for development of packages (pure Make, autotools. . .).
- Explore support for non-“simple” recipes.
- Interface to Spack’s upcoming new, (much) faster concretizer and explore improvements for SpackDev-specific tasks such as identification of missing intermediate packages for build consistency.
- Tools to facilitate construction of dependency trees, and storage or reuse of precalculated trees.
- Integration with other Spack facilities such as “chains,” and “environments.”
- SpackDev is applicable beyond HEP since it does not inherently rely on only one package build system $\implies$ integrate into Spack proper.

More information:

- The SpackDev GitHub page: https://github.com/FNALssi/spackdev.
- Wiki: https://cdcvs.fnal.gov/redmine/projects/spack-planning/wiki.

Backup slides. . .

Example Spack recipe

```
class Root(CMakePackage):
    homepage = "https://root.cern.ch"
    url = "https://root.cern.ch/download/root_v6.16.00.source.tar.gz"
    version('6.16.00',
            sha256='2a45055c6091adaa72b977c512f84da8ef92723c30837c7e2643eecc9c5ce4d8',
            preferred=True)
    patch('find-mysql.patch', level=1, when='@:6.16.00')
    variant('fftw', default=False, description='Enable FFT support')
    depends_on('fftw', when='+fftw')
    conflicts('%intel')  # Can't compile ROOT with the Intel compiler.

    def cmake_args(self):
        spec = self.spec
        options = [ '-Dfftw3:BOOL=%s' % ('ON' if '+fftw' in spec else 'OFF') ]
        return options

    def setup_environment(self, spack_env, run_env):
        run_env.set('ROOTSYS', self.prefix)
        run_env.set('ROOT_VERSION', 'v{0}'.format(self.version.up_to(1)))
        run_env.prepend_path('PYTHONPATH', self.prefix.lib)
        if 'lz4' in self.spec:
            spack_env.append_path('CMAKE_PREFIX_PATH',
                                  self.spec['lz4'].prefix)
        spack_env.set('SPACK_INCLUDE_DIRS', '', force=True)
```

Selected excerpts from the ROOT recipe; original at https://github.com/spack/spack/blob/develop/var/spack/repos/builtin/packages/root/package.py.