HSF Packaging Group: Common Directions for the Future

HEP Software Foundation

Graeme Stewart and Ben Morgan
(for the HSF Packaging Group)

2018-05-30

Group’s Goals

- Building, packaging and distributing software is a problem faced right across the HEP community (so, not just LHC or even CERN)
    - Every experiment and software group need to put effort into doing this
        - Naively it seems easy, quickly it gets complicated
    - Developers of libraries and toolkits need to care about easy integration into a stack
- So, prima facie, this is an area where we can work together to improve
    - Common build recipes and tools
    - How to take most advantage of technologies like containers
    - Proper support for developers in our collaborations
- Experiment production stacks are vital, but good tools and solutions will be completely portable to other use cases, e.g., lightweight releases for analysis or machine learning

http://hepsoftwarefoundation.org/activities/packaging.html

The N-Dimensional Problem

- Where N is surprisingly large...
    - Package set
        - Including versions, that may be locked or floating
        - Dependencies of packages
    - Target Architecture
        - Including micro-architecture variations
    - Compiler suite
        - gcc, clang, icc @my version
    - Set of compiler options
        - Usual opt and debug, plus any other variants
    - Host OS
        - May supply system libraries and build tools
- This is a very large space, but only sparsely filled

Packing Group Report

- Group was very active in 2015 and 2016 and looked at many solutions in the space of build orchestration
    - That is, the problem of building a stack, as opposed to building an individual package
        - For the single package problem, CMake seems to be the de facto choice of the community for C/C++ projects, now widely used - that does nicely simplify things for many of our HEP specific packages
    - System needs to manage
        - Dependencies of each individual project
        - Setup of the correct build environment for each piece
        - Manage artefacts from the build for subsequent installation
    - Looked at tools from the community and in the wider FOSS world

http://hepsoftwarefoundation.org/notes/HSF-TN-2016-03.pdf

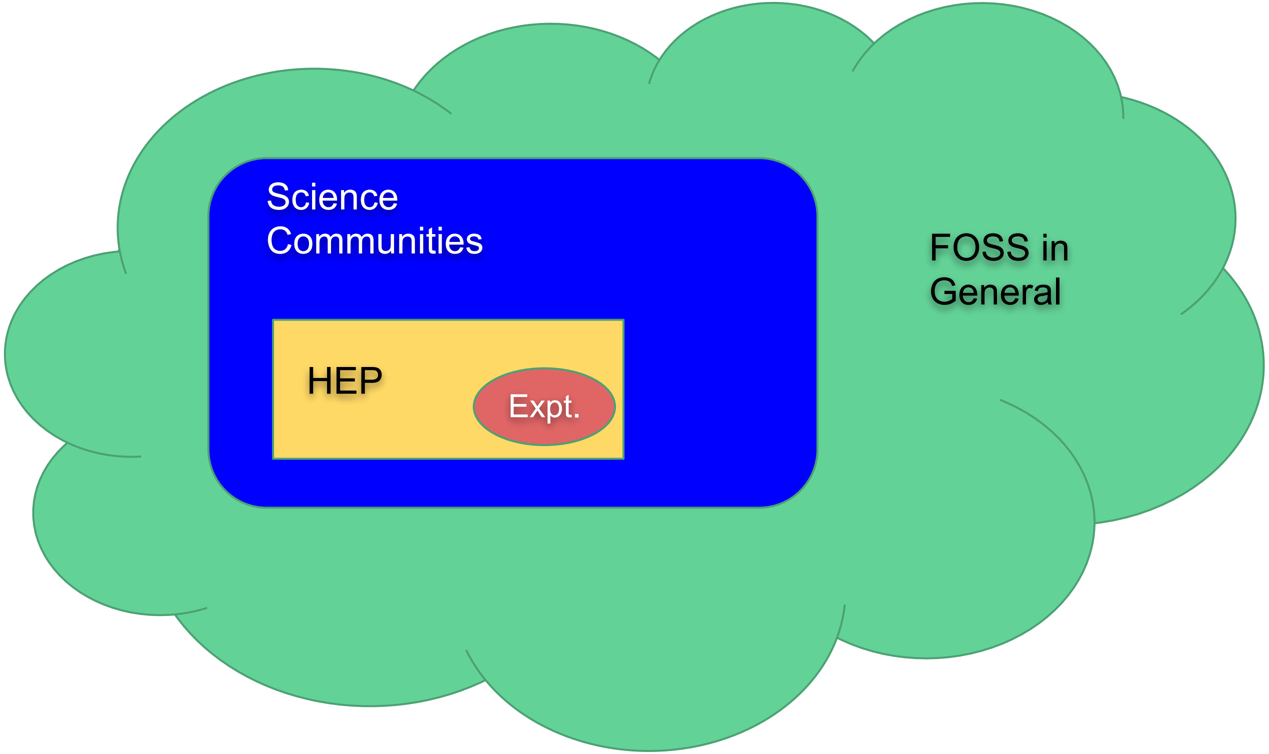
Post-Report

- Checkpoint: Most promising tools seemed to be
  - From our community
    - LCGCMake
    - aliBuild
  - From wider scientific community
    - Spack
- Some prototype work done with Spack to adapt to our use cases
  - Proved Spack community was rather receptive to patches we provided for upstream
- Things went a bit quiet after that
  - Usual case of people being pulled off to other projects...

Restart of activities

- Restarted activities in Autumn last year
  - Ben Morgan and Graeme Stewart took over from previous convenors
    - Thanks to Benedikt Hegner and Liz Sexton-Kennedy for their work
- Landscape changes
  - New tools arrive
    - Often with some enthusiastic proponents!
  - Use cases evolve
  - Experience is gained
- However, don’t start from zero - build on what we know

The Sociological View



- We have a hierarchy of communities
- The bigger the community the more likely useful effort from others
- But the solution may not quite fit our needs
- Have to find the *sweet spot*

Use Cases

- Write down what the our use cases actually are
  - Define what problem it is that we want to solve
    - As opposed to “how” we solve it today
  - In particular, where do we differ from “normal” (e.g., a Linux distribution)
    - This can drive us to more specialised areas in the solution space

https://docs.google.com/document/d/1h-r3XPIXXxmr5tThIh6gu6VcXXRhBXtUuOv14ju3oTI/edit?usp=sharing

Use Case Highlights I

- Determinism
    - Better know what we did and have confidence we can redo it
- Multiple Build Flavours
    - Different compilers, different build flags must be supported (for the same code base)
- System Component Use
    - *Should* be able to build using some or other components of some base system (e.g., the base OS)
    - This can be in contradiction with the requirement for determinism - may affect reproducibility of the build and *even the runtime*
- Build Efficiency
    - Should exploit parallelism available during a build
    - Should be able to share identical components between builds (e.g., Python modules)
    - Reuse binary artefacts from a previous compile
- Chain Builds Together
    - E.g., LCG Build ➔ Experiment Build ➔ Developer Build

Use Case Highlights II

- Share Recipe Knowledge
    - Build recipes should be easy to write as well, to maximise cooperation
- Deploy to Different Systems
    - E.g., Local, CVMFS, Container - *install time relocation*
- Deployment Independence
    - Deployed releases should not interfere with one another
        - But artefact sharing is very desirable
- Patch and Remove
    - Deployed releases should be updatable and removable
- Runtime Environment
        - Essential to set this up correctly
        - Needs to be flexible enough to support development
        - Can be partial to reflect a subset of a software build, such as a *view*

HEP Specificities and talking points

- The need to have multiple releases, with different build flavours, deployed side by side
  - Relatively common in other sciences too
- Install time relocatability
  - Usually software is built knowing where it will be deployed too
  - Install time relocation sometimes poorly supported
    - A lot of community knowledge exists as to how to do this, but results are not always 100% reliable
- Use or exclusion of the system libraries
  - Here there is a significant difference between building a single piece of code and building a stack
    - For a single package, use of the system is essential
    - For a stack, as interdependence becomes larger, less clear

Test Stack

- Define a basic set of software that would be representative of a small HEP experiment
    - Not meant to be complete, but not trivial either
    - ~45 packages
        - With their own implied dependencies
    - Can be used to ‘test drive’ different solutions

https://docs.google.com/document/d/1LW8OsTFFA9QwsJ9fASkRoJ2E6Gk3UGnOQIcEICL8UCM/edit?usp=sharing

Test Drive I

- Now that we defined a set of use cases we want to satisfy and a set of packages we want to support we can make some objective tests of the strengths and weaknesses of different tools
    - Someone who knows the tool should prepare a base environment in which the tool is setup correctly
        - We ask for this to be done as a Docker container on top of a CentOS7 base image
            - Dockerfile is great for showing exactly what needs to be done
    - Add some instructions that demonstrate the basic steps of building using the tool and pointers to other documentation
        - This is the bootstrap guide

Test Drive II

- Now have test drive instructions for number of tools
  - Nix
  - Portage
  - aliBuild
  - Spack
  - LCGCmake (being prepared)
- N.B. Being able to take the tool for a test drive is a pretty basic test (“Look! I am not broken”)
  - But it gives people a flavour of each tool
    - Important to test the ‘look and feel’
  - Serves as a basis for the other use cases (e.g., patching, moving binary artefacts elsewhere, etc.)
- You are very much encouraged to give this a try:
  - https://github.com/HSF/packaging/tree/master/testdrive

Talking Points

Shallow vs. Deep Builds

- Building relying on the host system's libraries and tools has been the usual way to build our stacks
  - Reduces build times
  - Offload maintenance to underlying OS
- However, this comes at a price
  - Builds become tied to the underlying OS
  - OS updates lead to reproducibility issues
- Building deep, up to or even including libc, increases build time *once*, but removes the axis of underlying OS from the equation
  - That provides some simplification and reliability
    - It's one of the few axes we can remove from the space

Package Hashes

- Dealing with a large multi-dimensional space of packages, dependencies and their build options
  - Encoding all options via the path is not very scalable
    - Paths can get really long
    - Metadata in “names” is fragile
- Very common solution is to convert the sources, dependencies and build options into a hash value
  - Keeps paths under control
  - Adaptable to a variable number of inputs into a package’s build formule
- Of course hashes are horrible to actually have in your path
  - Common to the construct a view of the release with some soft links
    - LCGCmake, Spack, Nix all use this mechanism

Relocatability

- Traditionally we always supported this
    - Various mechanisms to do it, e.g., making relocatable RPMs, using simple tarballs
- Requires some gymnastics to ensure that the configuration gets updated correctly with the relocated paths
    - Has been a real pain point and can be hard to debug
    - Especially if system paths are left
        - Falling back to old system libraries can break things in subtle ways
- We do this to economise on CPU
    - But human cycles are much more costly than CPU cycles
- With the reduction in the number of paths used in practice (CVMFS) is relocatablity worthwhile investment anymore?
    - At least some people in the group think it may not be
- Install time relocation certainly missing from tools like Nix
    - In Spack we have added support for it

Next Steps

- Continue with the evaluation of tools
    - Test drive makes this reasonably objective
    - Tradeoffs are part of life
        - We will not find one tool to rule them all
            - We adapt code (contribute) and might trade off use cases (relocation)
- Conclude on a best practice recommendation
    - This may be suite of tools that cooperate nicely
        - Probably not desirable to have a monolithic solution
- Develop support and documentation for the community

Any work that people are doing in the build, packaging and deployment area is interesting for the group, so contributions very much welcomed

Backup

Early Observations on Tools

- Nix
  - Pure functional package manager
  - Generic FOSS
  - Builds very deeply (even libc) - excellent reproducibility
  - Excellent support for multiple versions and flexibly constructed sub-environments
  - Not binary relocatable - install path (default, /nix) is a part of the package hash
    - One area to install and deploy, must be writable
    - Hard for CVMFS (read only) and for users (overlay FS?)
  - Package description language is a customised functional DSL - alien for HEP people
- Guix
  - GNU functional package manager
  - Very like Nix (see above)
  - Uses scheme instead of DSL (also alien)

Early Observations on Tools

- Portage
  - Package manager from Gentoo Linux (generic)
  - Can be installed “on top” of any other Linux base OS as well
  - Builds deep, own libc
  - Supports multiple versions, upgrade and rollback, but only one active version at a time
    - At least on any single path “prefix” (but you can have a few of these)
  - Does support relocation
- Spack
  - Developed at LLNL for supporting HPC software
  - Significant number of other users from difference science communities
  - Builds deep (by default), but can be told about system libraries
  - Support for relocation was added by us
  - Chained builds (one Spack on top of another) is a PR
  - Runtime/development environment is a WIP (Chris Green, FNAL)

Early Observations on Tools

- aliBuild
  - Used only by ALICE (maybe SHiP?)
  - Optimised for HEP use
  - Very flexible in use (or not) of system libraries
    - Good for end users in particular
  - Robust relocation
  - Limited support at the moment (Giulio cloning required)
- LCGCMake
  - Well known to everyone here!
  - Shallower builds by default (different default from other systems)
  - Small user community (SFT++)
    - We own it - get to fix, enhance and break it as we like
  - Known operational issues are a lot of the focus of this workshop
    - But, rather unfair to directly compare in this respect to non-battle tested tools