Preliminary evaluation of Spack as a new backend for LCG releases

G. Ganis, D. Konstantinov, I. Razumov

Special LiM meeting, 15 March 2021

Introduction

What is Spack?

Spack is a package management tool designed to support multiple versions and configurations of software on a wide variety of platforms and environments. It was designed for large supercomputing centers, where many users and application teams share common installations of software on clusters with exotic architectures, using libraries that do not have a standard ABI. Spack is non-destructive: installing a new version does not break existing installations, so many configurations can coexist on the same system.

Introduction

Why Spack?

- Is recommended by HSF Packaging group

Introduction

Why Spack?

- Is recommended by HSF Packaging group
- After CHEP'19 talk by G. Stewart, the Experiments (ATLAS and LHCb) has expressed interest in evaluating Spack

Introduction

Why Spack?

- Is recommended by HSF Packaging group
- After CHEP'19 talk by G. Stewart, the Experiments (ATLAS and LHCb) has expressed interest in evaluating Spack
- Already used by FAIR, Key4hep (CLIC/ILC, CEPC, FCC)

The Plan

- As a first step, we have created an equivalent of LCG_98python3_ATLAS_5 layer for one platform (`x86_64-centos7-gcc8-opt`) for the experiments to test.
- We are still working on porting some recipes, the installation will be updated as they are merged upstream.

Usage model

The foreseen usage model of Spack in LCG is quite different from a typical usage of Spack:

- Take as much from the underlying OS as possible
  - `X11` libraries, common tools (`diff`, `patch`, `tar`, ...), etc.

Usage model

The foreseen usage model of Spack in LCG is quite different from a typical usage of Spack:

- Take as much from the underlying OS as possible
  - `X11` libraries, common tools (`diff`, `patch`, `tar`, ...), etc.
- Use pre-built compilers

Evaluation

The Good:

- Spack have an easier syntax (Python) for recipes, has templates (e.g. for autoconf or python packages)
- A lot of "hacks" we have or want in lcgcmake (e.g. setting a fixed version of dependency, using git commit id in place of version, "layered" releases[1]) are already implemented in Spack
- Recipes for many packages are already available in Spack: of 387 "external" packages (i.e. everything but generators), Spack has recipes for about 279.
  - We are missing a few Python packages (mostly Jupyter add-ons), all GRID packages, packages written in 'go' (unified support for this type is being worked on)
  - Migrating the recipes for them is not expected to be a difficult task, just time consuming.
- Parallel builds are supported

[1] a.k.a. environments

Evaluation

Open Issues and Relevant Future Developments:

- The `concretizer` (a part of Spack that resolves versions and dependencies of packages) has a few problems :
  - In some cases (e.g., python packages like matplotlib) one needs to explicitly specify versions of dependencies, otherwise the concretizer will pick a default one, and will complain about incompatibility
  - "Virtual" packages (i.e. packages that have more than one implementation: blas, lapack, java) are often not resolved correctly — even when you explicitly set what package provides what in `packages.yaml`.
  - With v0.16.0 a new concretizer based on clingo, a constraint solver library has been introduced to spack.
  - The new concretizer does not immediately solve the issues related to concretization, but provides the technical foundation to do so in a future release.

Evaluation

Open Issues and Relevant Future Developments:

- The `concretizer` (a part of Spack that resolves versions and dependencies of packages) has a few problems
- Data Packages :
  - In some cases (e.g., Geant4, lhapdf), software packages include large data sets whose format is independent of the compiler or operating system used for a build.
  - It would be nice to reuse them between builds
  - Not yet possible with Spack, but making compiler a first-class dependency is planned
    - Workaround: declare these datasets `external` and `buildable: False`

Main questions

- Build everything with Spack or continue relying on system packages?
  - LHCb: build everything with spack
- Build gcc and clang with Spack or use existing ones?
- ROOT dependencies: builtin or external?
  - LHCb: external, unless there is a special reason to do otherwise
- Target architectures for `x86_64`: `Haswell`? `Nehalem`? Base architecture for g̈enericL̈CG build?
  - LHCb: Nehalem and newer
  - ATLAS: Use new RedHat notation (x86_64-v1, x86_64-v2, ...; only available for gcc11+ / clang10+, also for gcc10 on Ubuntu)
- Use `rpath` (default), `runpath` or remove relocation information and use LD_LIBRARY_PATH (requires changes to Spack core)
  - LHCb: whatever Spack does by default (`rpath`)

What is available now

- SFT repository with Spack config and some recipe overrides (changes frequently)
- Source cache on EOS: here
- Experimental CVMFS installation, includes module files and a view: `/cvmfs/sw.hsf.org/sft-spack/`
- Valentin will give a quick explanation how to use them