Some updates on binary cache handling in Spack

9th annual MVAPICH User Group Meeting (MUG’21)
August 25, 2021

Todd Gamblin
Livermore Computing
The Spack community continues to grow!

5,600+ software packages
820+ contributors

Package contribution rate increased in the past year

All time high of 3,700 monthly active users this March
Four of the top six most wanted features in Spack are tied to the new concretizer

- Complexity of packages in Spack is increasing
  - many more package solves require backtracking than a year ago
  - Many variants, conditional dependencies, special compiler requirements

- More aggressive reuse of existing installs requires better dependency resolution
  - Need to be able to analyze how to configure the build to work with installed packages

- Separate resolution of build dependencies also requires a more sophisticated solver
  - Makes the solve even more combinatorial
  - Needed to support mixed compilers, version conflicts between different package’s build requirements
E4S team has built a binary cache with over 37,000 Spack binary packages

- Built for multiple OS’s, architectures
- E4S team is working with ECP projects to accelerate their build pipelines
- Improved performance of cloud CI for one project by 10-100x
  - Previously, builds took too long for free cloud CI
  - Project can now iterate faster using Spack/E4S binaries

https://e4s.io (click on “build cache”)
We are expanding our CI builds to include every pull request!

- Spack Contributions on GitHub
- Pipelines running in AWS
- Pipelines at LLNL (in progress)
- Pipelines at U. Oregon (in progress)

gitlab.spack.io

E4S spack.yaml configuration

spack ci

GitLab CI builds (changed) packages
- On every pull request
- On every release branch
- Different compilers (Intel soon!)

This check is (finally) required!

- New security support contributions from forks
  - Sandboxed build caches for test builds
  - Authoritative builds on mainline only after approved merge
Instructions for building MVAPICH2-GDR from build caches are... baroque to say the least 😞

- Clone spack
- Register externals
- Add mirror
- `wget` and trust key
  - Can now be replaced with: `spack buildcache keys --install --trust`
- Install buildcaches by hash, **per dependency**
  - And deal with weird errors if you encounter them
- Eventually get the right mvapich2-gdr package installed from the build cache
Spack has had compiler detection for a while
- Finds compilers in your PATH
- Registers them for use

We can find many packages now
- Package defines:
  - possible command names
  - how to query the command
- Spack searches for known commands and adds them to configuration

Community can easily enable tools to be set up rapidly
The new concretizer arrived in Spack v0.16

- Leverages Clingo (see https://potassco.org)

- Clingo is an Answer Set Programming (ASP) solver
  - ASP looks like Prolog; leverages SAT solvers for speed/correctness
  - ASP program has 2 parts:
    1. Large list of facts generated from our package repositories and config
       - 30,000 facts is typical – includes dependencies, options, etc.
    2. Small logic program (~700 lines), including constraints and optimization criteria

- New algorithm on the Spack side is conceptually simpler:
  - Generate facts for all possible dependencies, send to logic program
  - Optimization criteria express preferences more clearly
  - Build a DAG from the results

- New concretizer solves many specs that current concretizer can’t
  - Backtracking is a huge win – many issues resolved
  - Currently requires user to install clingo with Spack
  - Solver will be automatically installed from public binaries in 0.17.0

Some facts for the HDF5 package
The new concretizer enables significant simplifications to packages, particularly complex constraints in SDKs

- Dependencies and other constraints within SDKs could get very messy

- The new concretizer removes the need for some of the more painful constructs

- Also allows for new constructs, like specializing dependencies
  - When conditions are now much more general
  - Can be solved together with other constraints.

In some cases we needed cross-products of dependency options:

- Before
  - depends_on('foo+A+B', when='+a+b')
  - depends_on('foo+~A~B', when='+a~b')
  - depends_on('~foo+~A+B', when='~a+b')
  - depends_on('~foo+~A~B', when='~a~b')

- After
  - depends_on('foo')
  - depends_on('foo+A', when='+a')
  - depends_on('foo+B', when='+b')

Specializing a virtual did not previously work:

- depends_on('blas')
- depends_on('openblas threads=openmp', when='^openblas')
People seem to like the new concretizer a lot!

- Resolution to many issues is now “try the new concretizer”
- More expressive constraints in packages
- People building large stacks especially find that:
  - Can concretize “together” without false conflicts
  - Easier to supply constraints on the CLI
  - Easier to specify constraints in stacks
- Still quite a few open issues
  - Can iterate much more rapidly on concretization issues
  - Changes to support new criteria are generally small
  - Getting optimization order right is tricky
Clingo is a C++ library with a python interface

- This means Spack now has a very central, non-python dependency (Clingo)

- Concretization is the first thing you have to do to install anything
  - But now the concretizer depends on Clingo!

- Clingo itself has some dependencies:
  - py-cffi (build/run)
  - python (build/run)
  - cmake (build)
  - re2c (build)
  - bison (build)

- Most of complexity is in build dependencies

- Also requires C++14 to build
Step 1: just make the new concretizer available (v0.16.0)

- Ensure that clingo is installed in the Python you use to run Spack
- Add this to config.yaml:

```yaml
config:
  concretizer: clingo  # default is “original”
```

- Fails if clingo isn’t actually importable from python
- Requires user to deal with installing clingo
Step 2: Bootstrap automatically from source (now in v0.16.2)

- Same config change

- No need to manually install clingo
  - Spack bootstraps a spec using the old concretizer
  - Builds the spec in a special “Bootstrap store”
  - Uses regular old Spack packages to achieve this

- Upside: fully automatic

- Downsides:
  - Time consuming
  - Build may fail (e.g., with an old C++ compiler)
  - Building from source is slow
  - High cost (in time) to pay for every Spack installation
Step 3: Bootstrap from binaries
(done on develop, will be in v0.17.0 release)

- We can now build Clingo binaries on GitHub

- GitHub actions workflow uses manylinux approach developed for Python to build very compatible binaries
  - Build in Centos5 image for old glibc
  - For each of macOS, Linux and for each Python version

- Spack uses these binaries by default
  - Fast -- no more building clingo from source!
  - Users can still opt out and bootstrap from source if they want.
Hash caching

- Hashes in Spack are created by concretization process
  - Spack as originally designed doesn’t remember hashes unless they’re installed

- Build caches are registries of hashes
  - Currently Spack can show you what’s in a build cache
  - It doesn’t remember what the hashes mean

- Same for spack spec and concretized environments
  - Spack forgets hashes!

- We’re implementing a hash cache so that you can be specific about things in build caches
  - Remember every hash we’ve ever shown
  - Allow users to refer to them on install

- This will allow you to install specific mvapich—gdr hashes without all the pain

Very unsatisfying!
Final frontier: reusing concretizer

- New concretizer allows us to add arbitrary optimization criteria
  - We can tell it to maximize number of hashes included in a build
  - Or minimize number of builds vs. binary installs

- Need to tell the concretizer about hashes in advance

- This is thankfully quite simple in ASP!
Minimizing builds, maximizing hashes automatically

Add facts about installed packages:

```prolog
installed_hash("openssl","lwatuuysmwkuharrncywvn77icdhs6mn").
imposed_constraint("lwatuuysmwkuharrncywvn77icdhs6mn","node","openssl").
imposed_constraint("lwatuuysmwkuharrncywvn77icdhs6mn","version","openssl","1.1.1g").
imposed_constraint("lwatuuysmwkuharrncywvn77icdhs6mn","node_platform","openssl","darwin").
imposed_constraint("lwatuuysmwkuharrncywvn77icdhs6mn","node_os","openssl","catalina").
imposed_constraint("lwatuuysmwkuharrncywvn77icdhs6mn","node_target","openssl","x86_64").
imposed_constraint("lwatuuysmwkuharrncywvn77icdhs6mn","node_compiler","openssl","apple-clang").
imposed_constraint("lwatuuysmwkuharrncywvn77icdhs6mn","node_compiler_version","openssl","apple-clang","12.0.0").
imposed_constraint("lwatuuysmwkuharrncywvn77icdhs6mn","hash","openssl","lwatuuysmwkuharrncywvn77icdhs6mn").
imposed_constraint("lwatuuysmwkuharrncywvn77icdhs6mn","depends_on","openssl","zlib","build").
imposed_constraint("lwatuuysmwkuharrncywvn77icdhs6mn","depends_on","openssl","zlib","link").
imposed_constraint("lwatuuysmwkuharrncywvn77icdhs6mn","hash","zlib","x2ankgssxsxa7pcnhzg5k3dhgacglze").
```

Tell concretizer to minimize builds by choosing hashes:

```prolog
% the solver is free to choose at most one installed hash for each package in the solution
{ hash(Package, Hash) : installed_hash(Package, Hash) } 1 :- node(Package).
% if a hash is selected, we impose all of its constraints
impose(Hash) :- hash(Package, Hash).
% if we haven't selected a hash for a package, and if it is not external, we build it
build(Package) :- not hash(Package, _), not external(Package), node(Package).
#minimize { 1018, Package : build(Package) }.
```
Reuse coming soon in 0.17!