Reusing dependencies across ecosystems: what stands in the way?

Todd Gamblin
Advanced Technology Office
Livermore Computing

FOSDEM 2021
Dependency Management Devroom
Software complexity is increasing dramatically

Applications today are highly integrated
- HPC simulations use many numerical libraries and physics packages
- Increasingly, AI stacks are being integrated into traditional physics workflows

Getting performance requires us to use all the available hardware
- GPU, fast network, processor

Frequently, we also want to reuse native dependencies
- System libraries
- Vendor-supplied drivers, runtime libraries, compilers

Integrating at this scale requires reusing code from many software ecosystems
Some fairly common (but questionable) assumptions

- **1:1 relationship between source code and binary (per platform)**
  - Good for reproducibility (e.g., Debian)
  - Bad for performance optimization

- **Binaries should be as portable as possible**
  - What most distributions do
  - Again, bad for performance

- **Toolchain is the same across the ecosystem**
  - One compiler, one set of runtime libraries
  - Or, no compiler (for interpreted languages)

Outside these boundaries, users are typically on their own
High Performance Computing (HPC) violates many of these assumptions

- **Code is typically distributed as source**
  - With exception of vendor libraries, compilers

- **Often build many variants of the same package**
  - Developers’ builds may be very different
  - Many first-time builds when machines are new

- **Code is optimized for the processor and GPU**
  - Must make effective use of the hardware
  - Can make 10-100x perf difference

- **Rely heavily on system packages**
  - Need to use optimized libraries that come with machines
  - Need to use host GPU libraries and network

- **Multi-language**
  - C, C++, Fortran, Python, others all in the same ecosystem
Supporting accelerated architectures require more components, and more complexity in the software stack

- MFEM has redesigned memory management and added 6 libraries
  - MFEM has not yet started to support the Intel GPUs on Aurora

- El Capitan will have a new, rapidly changing set of libraries, compilers, and language runtimes
HPC simulations rely on icebergs of dependency libraries

MFEM: Higher-order finite elements
31 packages, 69 dependency edges

LBANN: Neural Nets for HPC
71 packages
188 dependency edges

MuMMI: Cancer/drug interaction modeling
98 packages
248 dependency edges
Spack enables Software distribution for HPC

• Spack automates the build and installation of scientific software
• Packages are templated, so that users can easily tune for the host environment

No installation required: clone and go

$ git clone https://github.com/spack/spack
$ spack install hdf5

Simple syntax enables complex installs

$ spack install hdf5@1.10.5
$ spack install hdf5@1.10.5 %clang@6.0
$ spack install hdf5@1.10.5 +threadsafe
$ spack install hdf5@1.10.5 cppflags="-O3 -g3"
$ spack install hdf5@1.10.5 target=haswell
$ spack install hdf5@1.10.5 +mpi ^mpich@3.2

• Ease of use of mainstream tools, with flexibility needed for HPC tuning

• Major victories:
  • Fugaku OSS software stack deployment
  • Deployment time for 1,300-package stack on Summit supercomputer reduced from 2 weeks to a 12-hour overnight build
  • Used by teams across U.S. Exascale Computing Project to accelerate development

$ git clone https://github.com/spack/spack
spack install hdf5

No installation required: clone and go

Simple syntax enables complex installs

$ spack install hdf5@1.10.5
$ spack install hdf5@1.10.5 %clang@6.0
$ spack install hdf5@1.10.5 +threadsafe
$ spack install hdf5@1.10.5 cppflags="-O3 -g3"
$ spack install hdf5@1.10.5 target=haswell
$ spack install hdf5@1.10.5 +mpi ^mpich@3.2

• Ease of use of mainstream tools, with flexibility needed for HPC tuning

• Major victories:
  • Fugaku OSS software stack deployment
  • Deployment time for 1,300-package stack on Summit supercomputer reduced from 2 weeks to a 12-hour overnight build
  • Used by teams across U.S. Exascale Computing Project to accelerate development
Spacc environments enable users to build customized stacks from an abstract description

- `spack.yaml` describes project requirements
- `spack.lock` describes exactly what versions/configurations were installed, allowing them to be reproduced.
- Frequently used to maintain configuration along with Spack packages.
  - Versioning a local software stack with consistent compilers/MPI implementations
- Allows users to specify external packages to integrate

---

**Simple spack.yaml file**

```yaml
spack:
  # include external configuration
  include:
    - ./special-config-directory/
    - ./config-file.yaml

  # add package specs to the `specs` list
  specs:
    - hdf5
    - libelf
    - openmpi
```

**Concrete spack.lock file (generated)**

```json
{
  "concrete_specs": {
    "6s68o2krtp2xvjezglndmavy6l3nu1": {
      "hdf5": {
        "version": "1.10.5",
        "arch": {
          "platform": "darwin",
          "platform_os": "mojave",
          "target": "x86_64"
        }
      },
      "compiler": {
        "name": "clang",
        "version": "10.0.0-apple"
      },
      "namespace": "builtin",
      "parameters": {
        "cxx": false,
        "debug": false,
        "fortran": false,
        "h": false,
        "mpi": true
      }
    }
  }
}
```
Build integration complexity has caused large delays in the MuMMI developer workflow

- LLNL’s MuMMI code is used to model drugs, including cancers and, more recently, COVID-19.
- When standing up Sierra, the team was at the mercy of constant system updates.
- We use system dependencies (MPI, compilers) to:
  - get the best performance from the machine.
  - avoid long build times.
When standing up Sierra, the team was at the mercy of constant system updates including OS updates, cancers, and, more recently, COVID. LLNL’s MuMMI code is used to model drugs, biomembranes, and lipid bilayers that form cellular membranes. This membrane association is an under-explored area of cancer biology that may be relevant to therapeutic intervention against cancer. We use lipid concentration gradients and protein aggregation simultaneously with current technologies. MD simulations can simulate the action between RAS proteins and a plasma membrane. We used up to 4000 nodes of the Sierra supercomputer, concurrently utilizing over 16,000 GPUs and 176,000 CPU cores, and running up to 36,000 parallel applications able to simulate problems of unprecedented scale instances simultaneously. Here, we describe the creation of the Multiscale Machine-Learned Modeling Infrastructure (MMI) framework, a massively parallel, adaptive simulation platform coupled with a highly scalable hierarchical job scheduler, and detail the central work governing the simulation workflow manager that ties these applications together. In addition, we present performance data from microscopic length- and time-scales hard to access through high-performance computing methods.

We use system dependencies (MPI, compilers) to:
- get the best performance from the machine.
- avoid long build times.

### Build integration

**When standing up Sierra, the team was at the mercy of constant system updates including cancers, and, more recently, COVID. LLNL’s MuMMI code is used to model drugs, biomembranes, and lipid bilayers that form cellular membranes. This membrane association is an under-explored area of cancer biology that may be relevant to therapeutic intervention against cancer. We use lipid concentration gradients and protein aggregation simultaneously with current technologies. MD simulations can simulate the action between RAS proteins and a plasma membrane. We used up to 4000 nodes of the Sierra supercomputer, concurrently utilizing over 16,000 GPUs and 176,000 CPU cores, and running up to 36,000 parallel applications able to simulate problems of unprecedented scale instances simultaneously. Here, we describe the creation of the Multiscale Machine-Learned Modeling Infrastructure (MMI) framework, a massively parallel, adaptive simulation platform coupled with a highly scalable hierarchical job scheduler, and detail the central work governing the simulation workflow manager that ties these applications together. In addition, we present performance data from microscopic length- and time-scales hard to access through high-performance computing methods.**

---


---

**Lawrence Livermore National Laboratory**

**LLNL-PRES-811108**
Transitive dependency requirements can cause cascading issues

- Team “just” needed a new version of Keras
  - which needed a new version of Theano
  - which needed a new version of Numpy
  - which needed a newer version of OpenBLAS
  - But the team was using the system OpenBLAS, which was too old
  - Team had to build several versions of OpenBLAS before they found one compatible with all other packages in the DAG
  - Then had to rebuild the entire stack for ABI compatibility

- This particular issue consumed **36 person hours**

- Frequent OS updates causing ABI incompatibilities between Flux, PMI, and the system MPI cost **hundreds of person hours**
Other teams (Axom, Serac, xSDK, E4S) have similar issues with managing configuration complexity

- Teams *really* like to lock versions down for testing:
  - Axom team tests on several systems and pins about 20 package versions to consistent (at the time) values
  - Serac team pins more system versions than this
  - xSDK and E4S stacks from ECP pin specific versions for each package

- Incompatibilities arise and builds fail in one or both of two ways:
  1. Spack upgrade leads to failure because new versions and options enter the Spack repository that are incompatible
  2. OS upgrades at a local site change local versions underneath a package

- Inevitably, this version locking effort is spent over and over again for subsequent releases
Even outside of HPC, dependencies are the most frequent cause of build errors and software release delays

Developers avoid updates to avoid problems with dependencies, leading to:
- Security vulnerabilities
- Lack of performance
- Stagnation as upgrades become harder and harder over time


Survey of 491 individuals from 691 teams at ING. Kula et al., Releasing fast and slow: an exploratory case study at ING. ESEC/SIGSOFT FSE 2019.
Three main ways to deal with dependencies have emerged in the past 10-20 years

<table>
<thead>
<tr>
<th>Bundled Distribution</th>
<th>Semantic Versioning</th>
<th>Live at Head</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Examples</strong></td>
<td>Bundled Distribution</td>
<td>Semantic Versioning</td>
</tr>
<tr>
<td>Linux distributions (Red Hat, Debian)</td>
<td>Spack</td>
<td>Google, Facebook, Twitter</td>
</tr>
<tr>
<td>E4S, xSDK, Anaconda</td>
<td>NPM, Cargo, Go</td>
<td></td>
</tr>
<tr>
<td>Spack with locked versions</td>
<td>Most language dependency managers</td>
<td></td>
</tr>
<tr>
<td><strong>Idea</strong></td>
<td>Curate a large set of mutually compatible dependencies</td>
<td>Use uniform version convention, Solve for compatible set</td>
</tr>
<tr>
<td><strong>Pros</strong></td>
<td>Stability (if software is included)</td>
<td>Frequent updates Only relies on local information Works in theory</td>
</tr>
<tr>
<td><strong>Cons</strong></td>
<td>Infrequent updates High packaging/curation effort Lack of flexibility</td>
<td>Versions are coarse Developers over-constrain/over-promise Errors start to dominate at scale</td>
</tr>
</tbody>
</table>

- All of the approaches have serious drawbacks
- Need a way to guarantee stability, frequent updates, and version/config flexibility
The fundamental problem with integrating native dependencies is lack of compatibility information

In workflows we’ve seen so far:

1. No information on how system libs were built
2. Build repeatedly to find compatible libraries
3. Hard constraints (pinned versions, etc.) hide information and limit choice

Each team ends up curating its own configuration with baked-in, incompatible assumptions

We need a way to reuse package builds among different communities of developers.

If all projects add restricted versions, conflicts will eventually arise that prevent all packages from building.
We’d really like to be able to reuse binary packages as we find them

1. When OS updates happen underneath a stack:
   — Know what changed by examining the binaries’ ABI
   — Identify what in the stack is no longer compatible
   — Rebuild compatible configurations

2. When user requirements change (e.g., due to a new version):
   — Know which packages need to change to meet the new requirements
   — Identify existing binaries (system or packaged) that satisfy the requirements
   — Install binaries or rebuild as necessary

3. When information is not available:
   — Extrapolate what and how to build based on past, similar builds

We will enable binary code reuse to reduce iteration in developer workflows
We’re kicking off the BUILD project at LLNL to address some of these issues

<table>
<thead>
<tr>
<th>Research Questions</th>
<th>Models</th>
<th>Binary Analysis</th>
<th>Dependency Solvers</th>
</tr>
</thead>
</table>
| What makes software packages compatible? | • What determines binary compatibility of functions and data structures? | • How to reconstitute interface information from binaries? | • Which NP-complete problem solver should we use?  
• ASP? SMT? |
|                    | • How can we model changes over time? | • How to know what data types are used by other binaries? | • How can we encode large problems for efficient solves? |
|                    | • How do changes affect other packages in a graph? | • How to efficiently manage debug information? | • What heuristics can be used to accelerate the solve? |
|                    | • Which changes are breaking? | • How to associate ABI information with versions? | • Do we need to develop new solver algorithms? |

BUILD: Binary Understanding and Integration Logic for Dependencies
What’s missing from current package ecosystems?

Current model is coarse

- Humans define rules like these:
  - A version 1.0 depends on B version 2.0
  - B version 2.0 depends on C version 3.0 or 4.0

- Humans update rules each time there is a new release

- Specification is incomplete
  - Runtime libraries, compilers, etc. are not modeled
  - Not clear whether updates to C require us to rebuild A
    - no ABI information

- No place for global constraints in the model, e.g.:
  - e.g., “must link with C++ compiler if any dependency uses C++”
  - “gcc and clang are ABI incompatible when ...”
Lack of ABI information about runtime libraries has bitten our code teams over and over again

- C++11 brought about binary incompatible changes to the C++ runtime library
  - All C++ codes had to be rebuilt with new compilers to be compatible
  - New builds could not be linked with old ones
- The C++ library version was not bumped, so strange runtime errors would happen
- ABI models would show us the differences in data type definitions between old and new
- This type of error happens frequently when mixing Fortran compilers, as well.
**Semantic versioning** is the de-facto standard for conveying compatibility information

- **Pitfalls:**
  - Applies to the whole package, but packages may only depend on a subset of functions
  - May over-promise: packages may break despite developers’ intentions
  - May over-constrain: pinned dependency versions are common but lead to false unsatisfiable cases

- **Relies on developers to specify versions correctly**
  - Relies on broad community participation

https://semver.org/
Humans do not accurately specify version information

- Plots show version restrictions across 4 package ecosystems.
  - **Permissive** leaves room for lots of room for incorrect builds
  - **Restrictive** rules out builds that would work
  - Non-specialized (white) leaves everything open (no constraints)

- HPC ecosystem is most like Ruby (right) – dated, with many permissive constraints
  - Most projects don’t use semantic versioning and won’t switch
  - Likelihood of build errors is very high
What if the package manager could model more aspects of ABI?

Current model is coarse

A version v1

B version v2

C version v3

C++ runtime version v4 (not modeled)

Complete model represents how changes affect code

A version v1

B version v2, defines t2

C version v3, defines t3

C++ runtime version v4 defines t1

- Libraries at call granularity:
  - Entry calls
  - Exit calls
  - Data type definitions & usage

- Runtime libraries behind compilers
  - C++, OpenMP, glibc
  - GPU runtimes

- Changes in the graph
  - “If C changes, what needs to be rebuilt?”
  - We will model semantics of interfaces
  - “If h(t3) changes, is B still correct?”

This model allows us to reason about compatibility
Containers in industry are used in an isolated way
  — No need (until recently) to use special hardware
  — Container has its own OS image

HPC containers assume tighter coupling between containers and host
  — Choose MPI or other layer as an interface
    • Omits transitive library considerations
    • Works until it doesn’t – no way to check or warn

Example: glibc compatibility issue at NERSC (right)

We will build models to check configurations and ensure binary compatibility in the full stack
  — Here we could fix the problem by including glibc 2.17
  — Our models will tell us that this is what is needed
To reason about ABI, we need better binary tools

- Binaries are the *ground truth* for compatibility
  - They contain the functions and data types we must link against
  - Can’t tell compatibility directly from source (depends on build)
  - Binaries are final output

- We often want to be able to make use of existing binaries, but it’s difficult to do in a cross-platform way

- There aren’t tools that can tell us when the thing we’ll build or link is *actually* compatible with an existing binary
  - Metadata and OS conventions vary by system
  - Debug information may not be available
  - Other communities’ builds may be very different from ours

We need better debug information to rely on prebuilt packages.
With the right tools, we can extract more compatibility information from binaries

- **OS-provided libraries**
- **Stripped proprietary binaries**
- **Prebuilt packages**

**Binary Analysis**

- **B version v2, defines** t2
- **f(t1)**
- **g(t1, t2)** entry points
data structures & semantics
- **h(t3)**
- **i(t1, t3)** exit points

**Compatibility Models**

- Libraries like *libabigail* and *dyninst* can do parts of this *with debug information (DWARF)*
  - Typically requires accurate debug information, which is not always shipped with the system

- Tools like *debuginfod* may allow us to look up debug information *without* having it installed
  - Not available for all distros
  - Needs to be on the critical path for package managers for broader availability

- Still some gaps even with debug information:
  - Dynamically loaded libraries
  - Can’t tell statically which ones are used
With compatibility information, we can improve the way we do dependency solving

- Package managers produce *valid* but not *sound* graphs.
  - ABI info gives us what we need for soundness

- Solvers could use models generated by binary analysis (ground truth)

- Past 10-20 years have brought enormous improvements in solver technology
  - CDCL algorithms
  - Optimizing SMT and ASP solvers

- Time is right to attack packaging with better solving

We aim to integrate binary compatibility checks into dependency solvers
Work on BUILD can be used across many package ecosystems

- **Currently, software curation effort is duplicated across packaging ecosystems**
  - Each has different assumptions that can affect compatibility
  - Each must be maintained in a different, closed world
  - Humans maintain metadata about packages for each

- **The C++ ecosystem suffers from many of the same issues as HPC**

- **The BUILD project aims to unify these domains**
  - Build a model for binary dependency solving that can be used by many package managers
  - Initially implement in Spack
  - Export tools and solvers for other packaging ecosystems

- **Ideally, not every system will need its own strategy for managing native dependencies**