Reusing dependencies across ecosystems: what stands in the way?

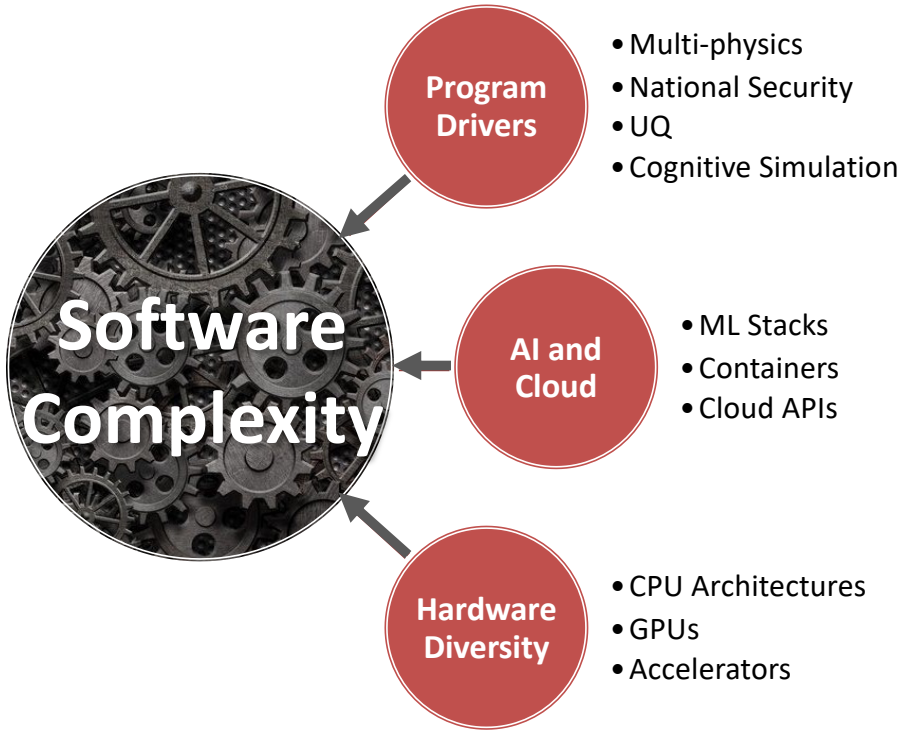
Todd Gamblin
Advanced Technology Office
Livermore Computing

FOSDEM 2021
Dependency Management Devroom

LLNL-PRES-811108
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

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

Some Supercomputers

Current

Oak Ridge National Lab
Power9 / NVIDIA

RIKEN
Fujitsu/ARM a64fx

Upcoming

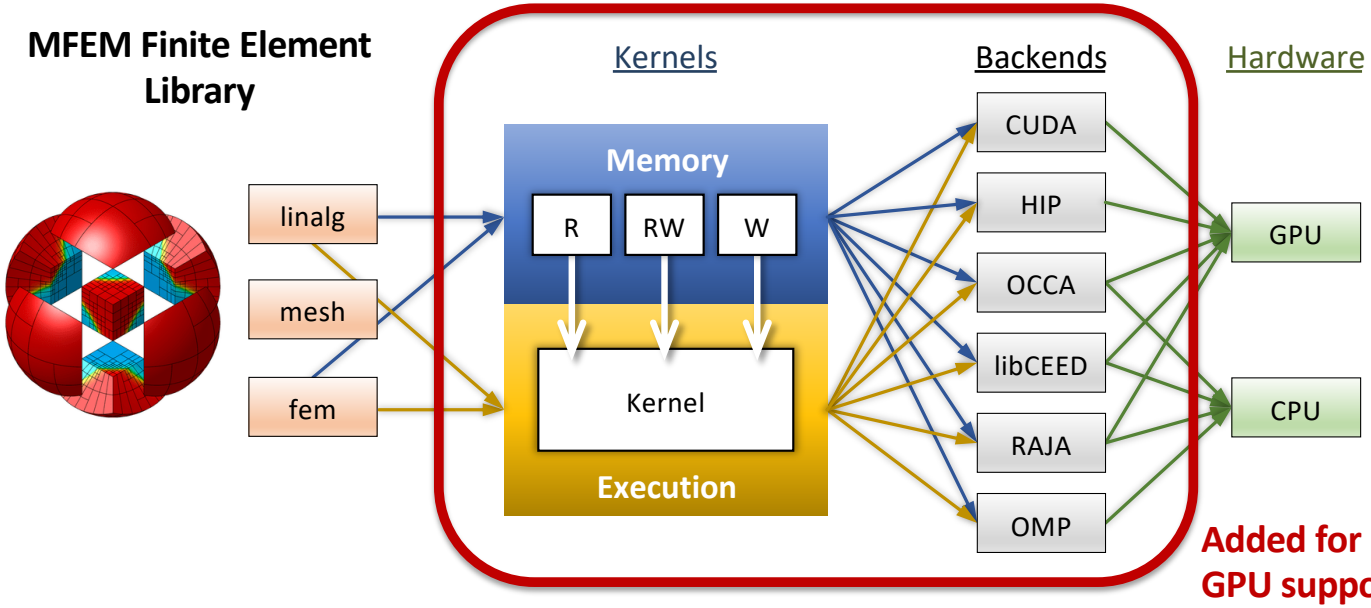
Lawrence Berkeley National Lab
AMD Zen / NVIDIA

Argonne National Lab
Intel Xeon / Xe

Oak Ridge National Lab
AMD Zen / Radeon

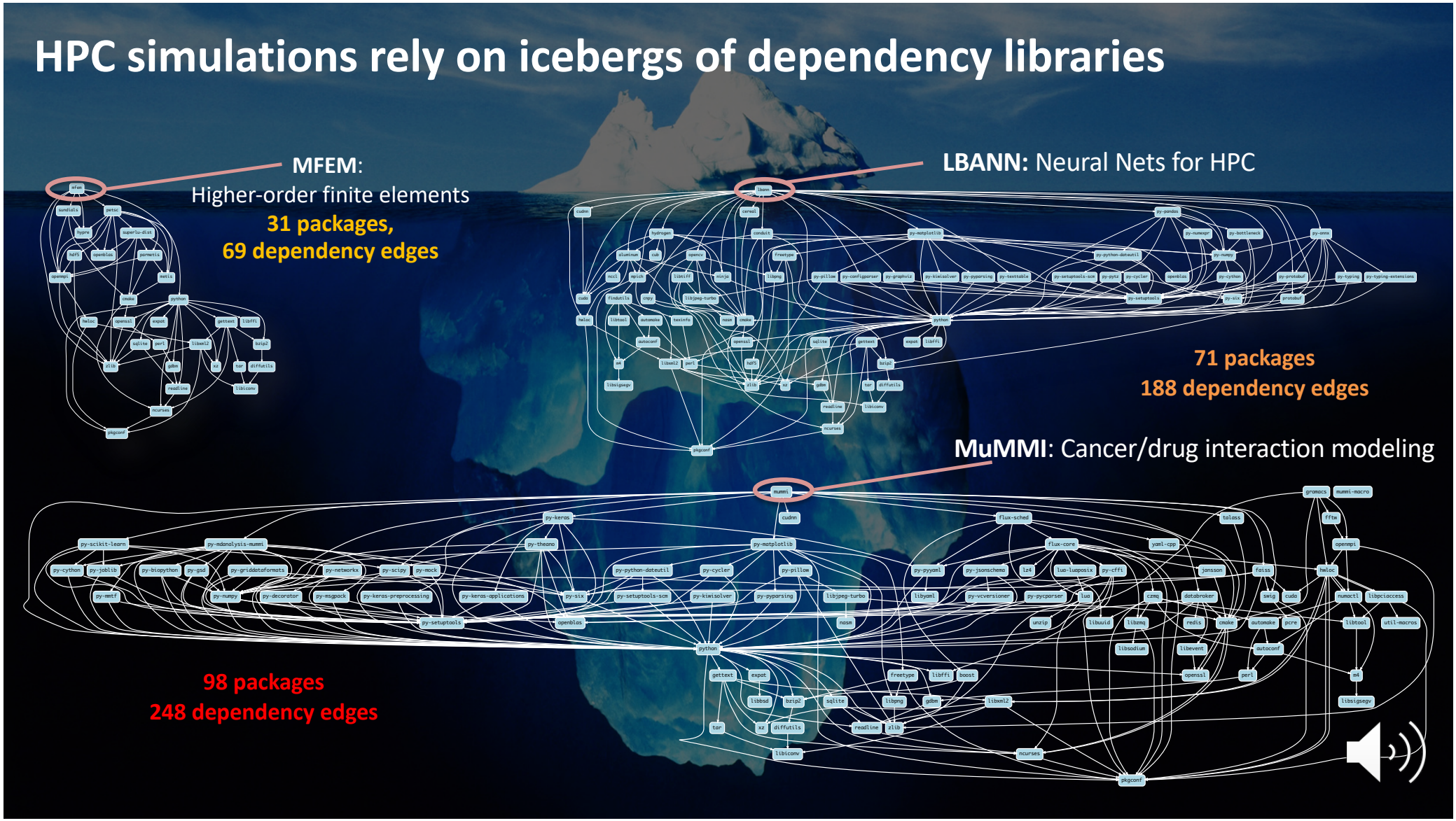
Lawrence Livermore National Lab
AMD Zen / Radeon

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



Spack enables Software distribution for HPC

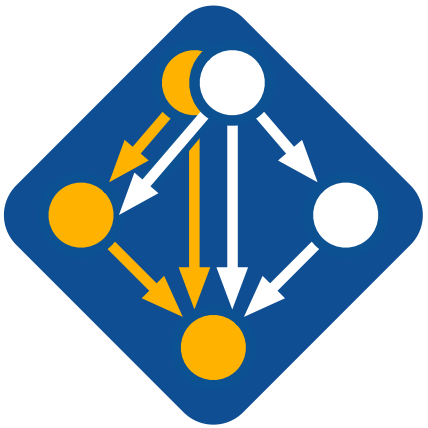
- Spack automates the build and installation of scientific software
- Packages are *templated,* so that users can easily tune for the host environment

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

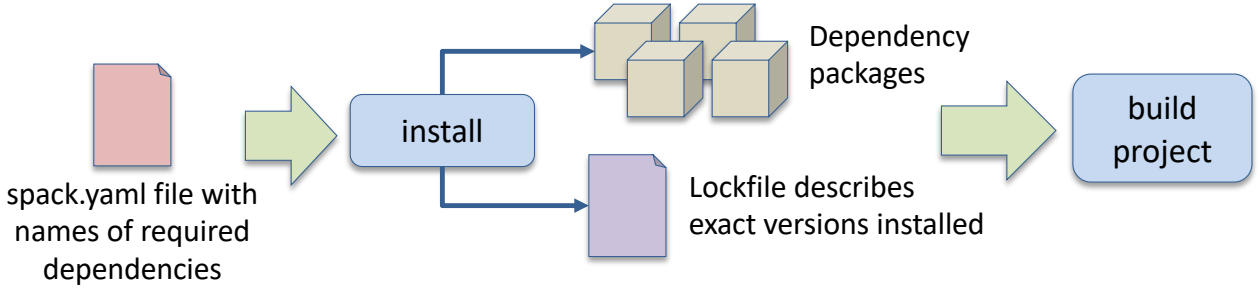


github.com/spack/spack

- 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**

Spack 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, allows 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

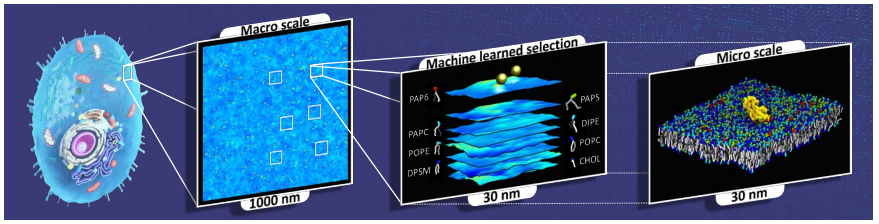
```
spack:
  # include external configuration
  include:
  - ../special-config-directory/
  - ./config-file.yaml

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

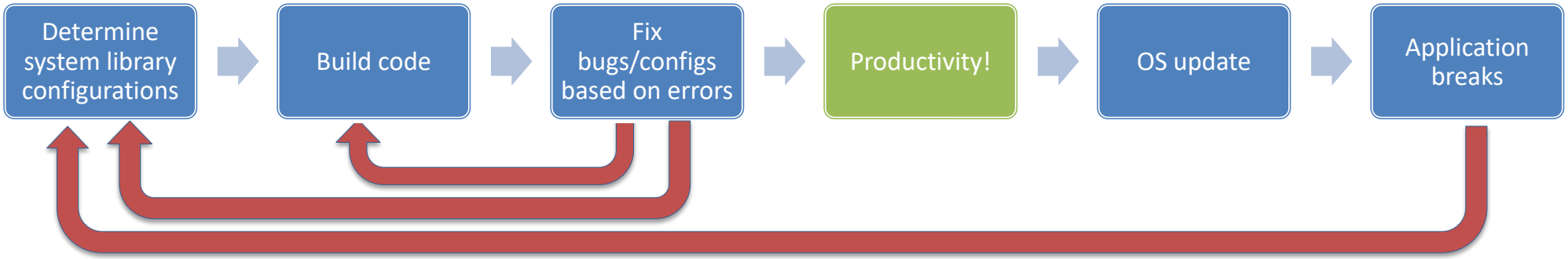
Concrete spack.lock file (generated)


```
{
 "concrete_specs": {
   "6s63so2kstp3zyvjezglndmavy6l3nul": {
     "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,
             "hl": false,
             "mpi": true,
```

Build integration complexity has caused large delays in the MuMMI developer workflow



- LLNL's MuMMI code is a 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

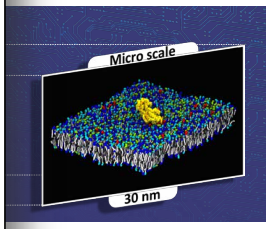
Di Natale et al. *A Massively Parallel Infrastructure for Adaptive Multiscale Simulations: Modeling RAS Initiation Pathway for Cancer*. In **Supercomputing 2019** (SC '19). 2019 **st paper.**

Build integratio… MuMMI devel… …he

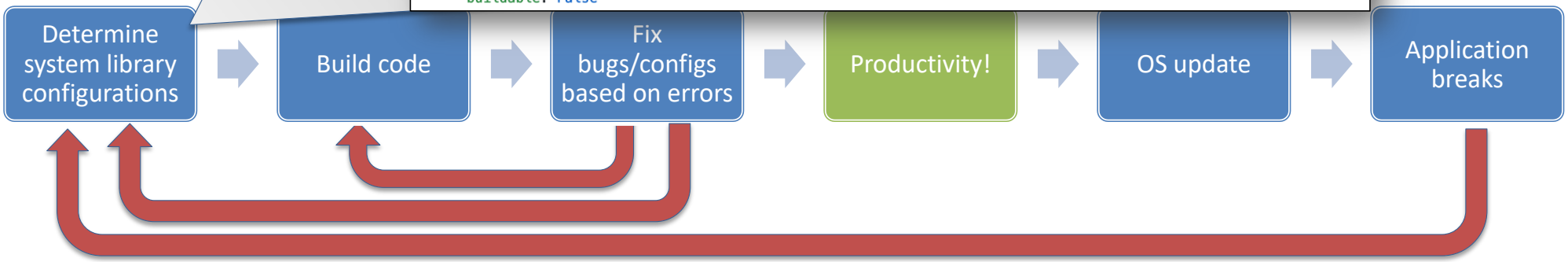
```
opengl:
  paths:
    opengl@1.7.0: /usr
  buildable: False
openglu:
  paths:
    openglu@1.3.1: /usr
  buildable: False

# Lock down which MPI we are using
mvapich2:
  paths:
    # clang mvapich2
    mvapich2@2.3%clang@9.0.0 arch=linux-rhel7-ivybridge: /usr/tce/packages/mvapich2/mvapich2-2.3-clang-9.0.0
    # gcc mvapich2
    mvapich2@2.3%gcc@8.1.0 arch=linux-rhel7-ivybridge: /usr/tce/packages/mvapich2/mvapich2-2.3-gcc-8.1.0
    # intel mvapich2
    mvapich2@2.3%intel@19.0.4 arch=linux-rhel7-ivybridge: /usr/tce/packages/mvapich2/mvapich2-2.3-intel-19.0.4
  buildable: False
```

~ 100 lines, 20 pinned system dependencies configured per machine



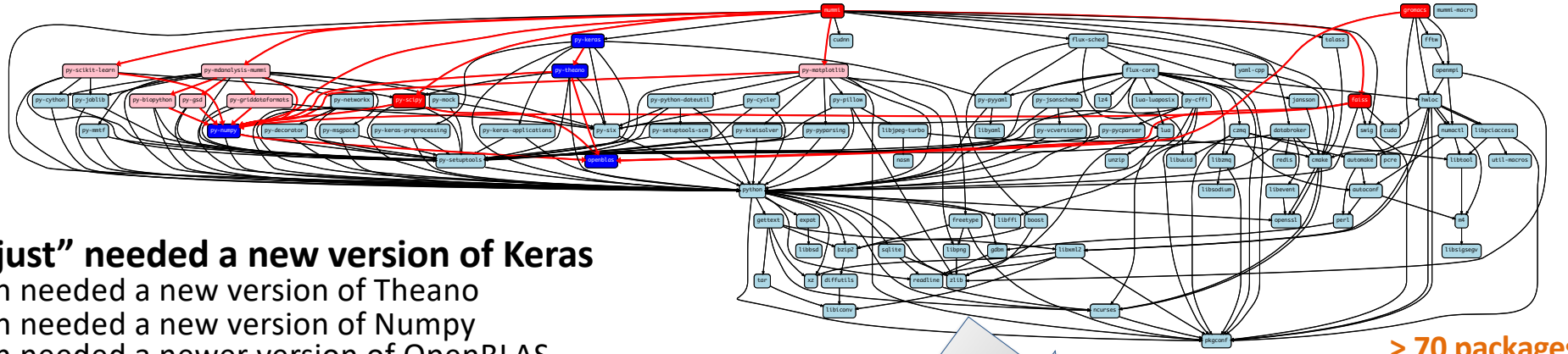
- LLNL's MuMMI c… including cance…
- When standi…



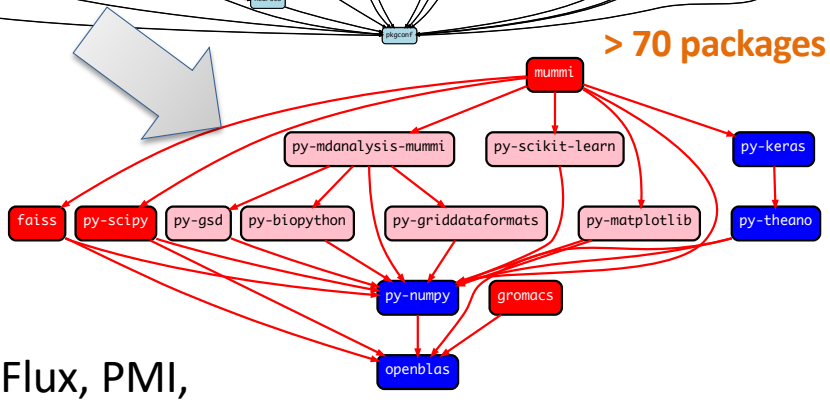
- We use system dependencies (MPI, compilers) to:
  - get the best performance from the machine.
  - avoid long build times

Di Natale et al. *A Massively Parallel Infrastructure for Adaptive Multiscale Simulations: Modeling RAS Initiation Pathway for Cancer*. In **Supercomputing 2019** (SC '19). 2019 …**st paper.**

Transitive dependency requirements can cause cascading issues

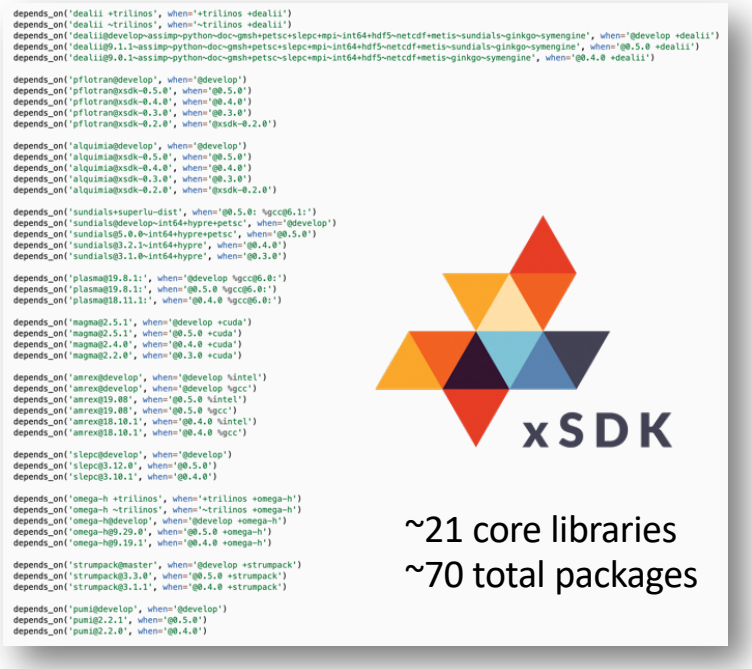


- **Team "just" needed a new version of Keras**
  - which needed a new version of Theano
  - which needed a new version of Numpy
  - which needed a newer version of OpenBLAS
  - But the team was using the system OpenBLAS, which was too old
  - Team had to build several versions of OpenBLAS before they 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

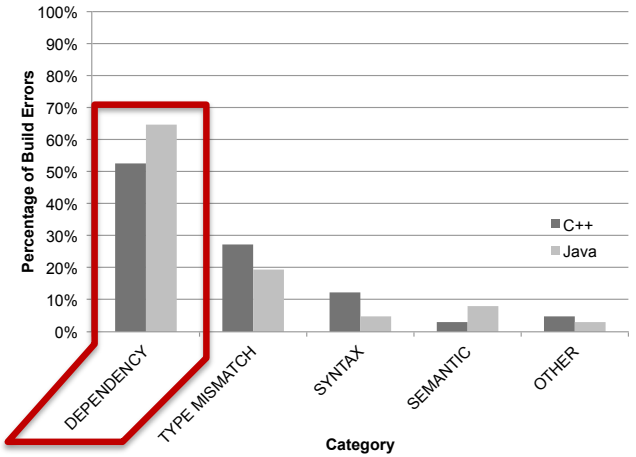
- Teams *really* like to lock versions down for testing:
  - **Axom** team tests on several systems and pins about 20 package versions to consistent (at the time) values
  - **Serac** team pins more system versions than this
  - **xSDK** and **E4S** stacks from ECP pin specific versions for each package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



Pinned versions and options from xSDK

Even outside of HPC, dependencies are the most frequent cause of build errors and software release delays

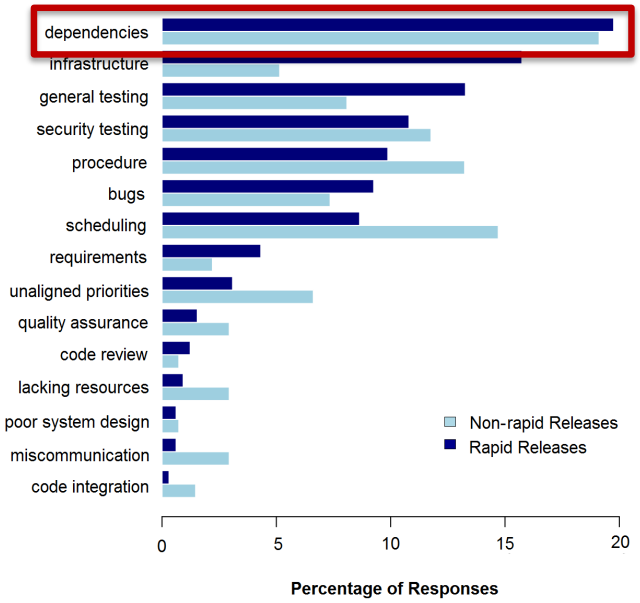
Types of errors in 26.6 million builds at Google



"our study clearly shows that better tools to resolve dependency errors have the greatest potential payoff"

Survey of 26.6M builds by 18K developers at Google. Seo et al., *Programmers' Build Errors: A Case Study (at Google).* ICSE 2014.

Factors perceived to cause release delays among 491 developers at ING



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.

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

Three main ways to deal with dependencies have emerged in the past 10-20 years

	Bundled Distribution	Semantic Versioning	Live at Head
Examples	Linux distributions (Red Hat, Debian) E4S, xSDK, Anaconda Spack with locked versions	Spack NPM, Cargo, Go Most language dependency managers	Google, Facebook, Twitter
Idea	Curate a large set of mutually compatible dependencies	Use uniform version convention, Solve for compatible set	Everything in one repository, Developers test changes with all *dependents*
Pros	Stability (if software is included)	Frequent updates Only relies on local information Works in theory	Frequent updates Stability, consistency All changes tested
Cons	**Infrequent updates** **High packaging/curation effort** **Lack of flexibility**	**Versions are coarse** **Developers over-constrain/over-promise** **Errors start to dominate at scale**	**Doesn't scale beyond a single organization** **High computational cost of testing** **Lack of flexibility (typically just one target env.)**

- All of the approaches have serious drawbacks
- Need a way to guarantee stability, frequent updates, and version/config flexibility

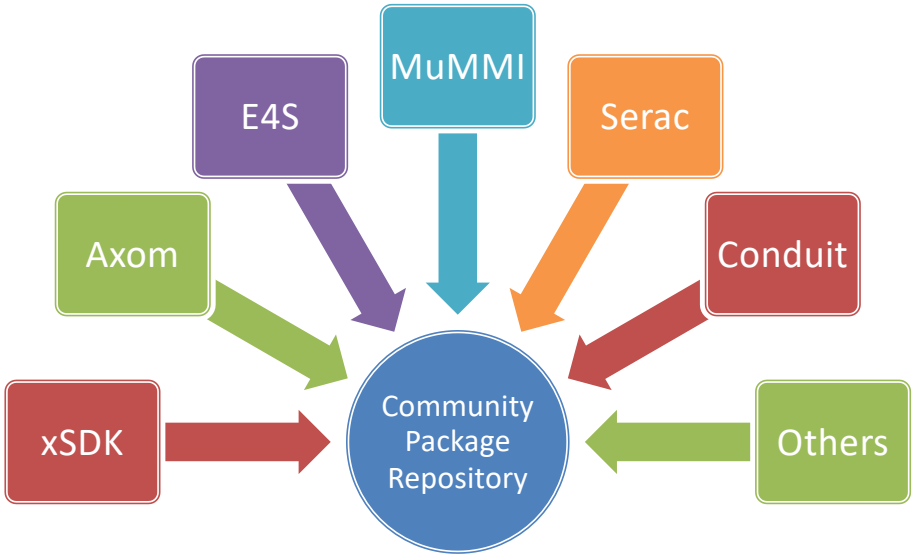
Taxonomy c/o T. Winters et al. *Software Engineering at Go[illegible]*. 2020.

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



If all projects add restricted versions, conflicts will eventually arise that prevent all packages from building.

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

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

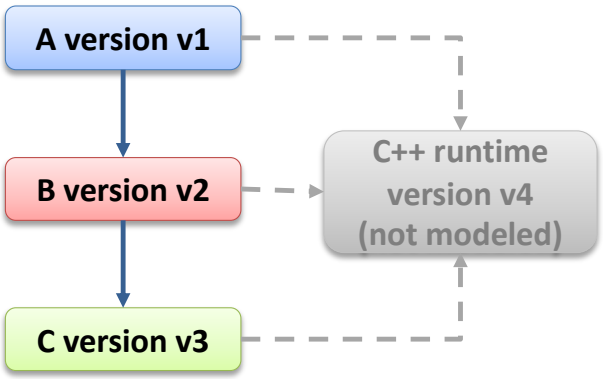
We're kicking off the BUILD project at LLNL to address some of these issues

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

BUILD: Binary Understanding and Integration Logic for Dependencies

What's missing from current package ecosystems?

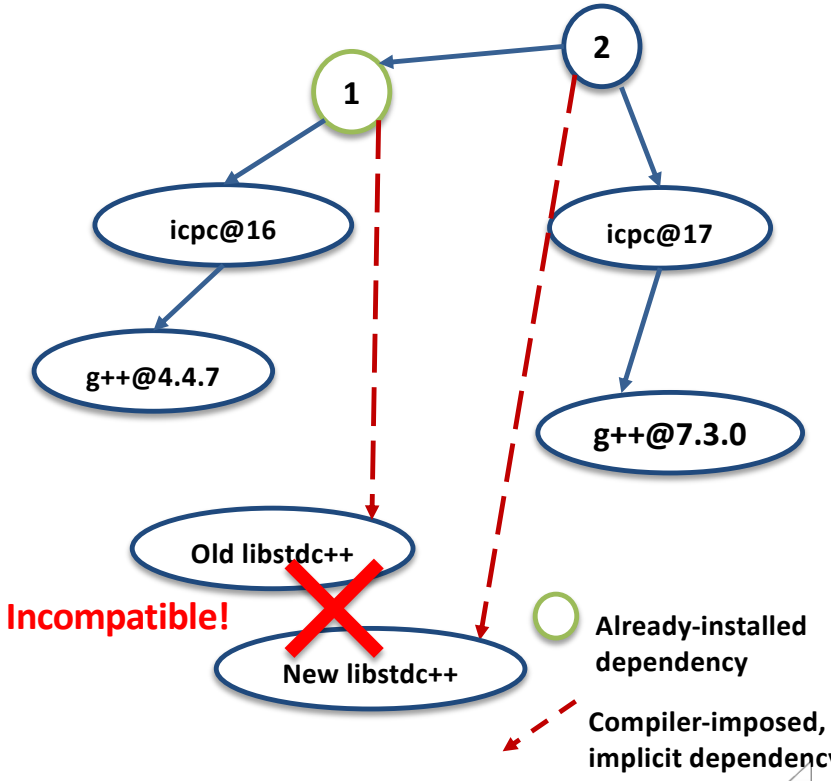
Current model is coarse



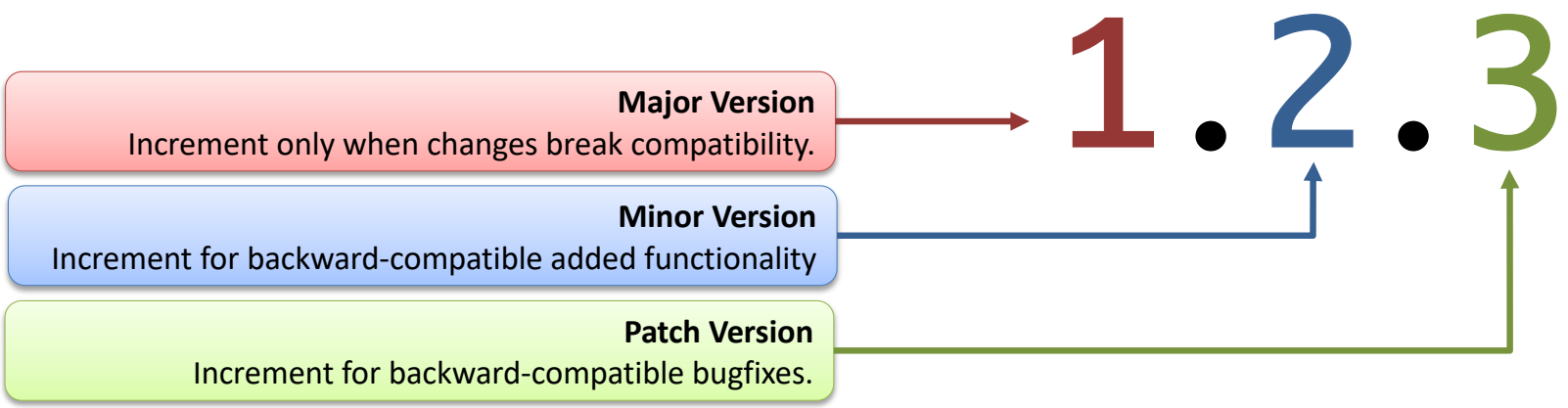
- Humans define rules like these:
  - A version 1.0 depends on B version 2.0
  - B version 2.0 depends on C version 3.0 or 4.0
- Humans update rules each time there is a new release
- Specification is incomplete
  - Runtime libraries, compilers, etc. are not modeled
  - Not clear whether updates to C 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.



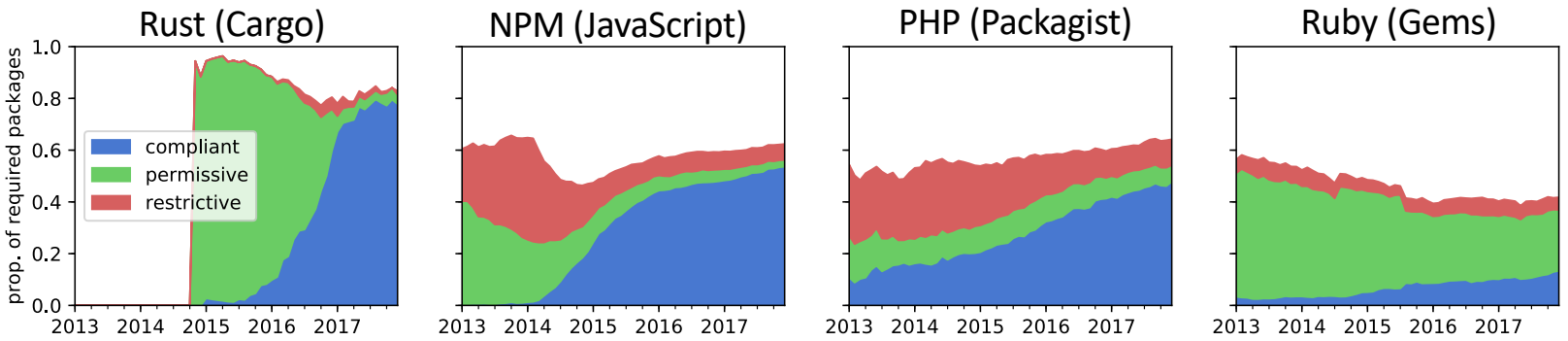
Semantic versioning is the de-facto standard for conveying compatibility information



- 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



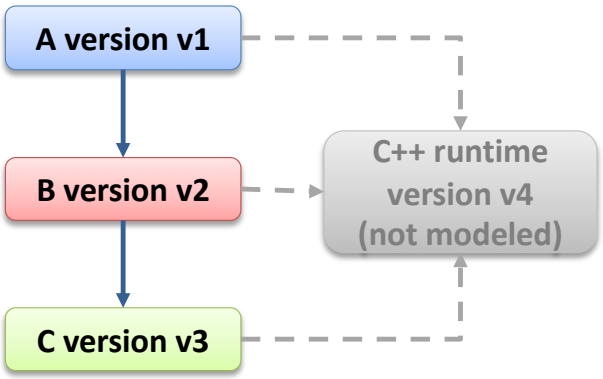
- Plots show version restrictions across 4 package ecosystems.
  - **Permissive** leaves room for lots of room for incorrect builds
  - **Restrictive** rules out builds that would work
  - Non-specialized (white) leaves everything open (no constraints)

- 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

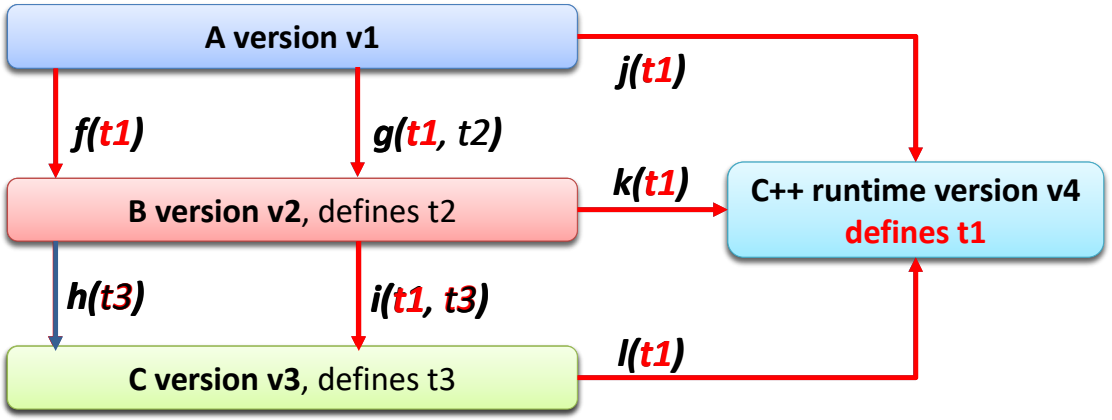
Decan et al. *What Do Package Dependencies Tell Us About Semantic Versioning?* IEEE Trans. Software Eng. [illegible] 2019.

What if the package manager could model more aspects of ABI?

Current model is coarse



Complete model represents *how* changes affect code



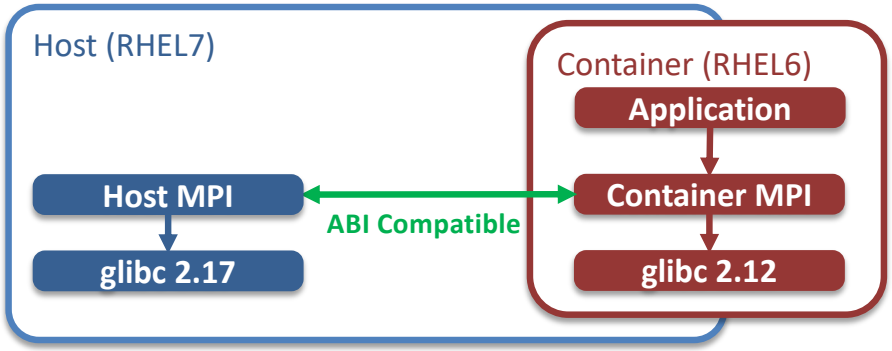
- Libraries at call granularity:
  - Entry calls
  - Exit calls
  - Data type definitions & usage
- Runtime libraries behind compilers
  - C++, OpenMP, glibc
  - GPU runtimes
- Changes in the graph
  - "If C changes, what needs to be rebuilt?"
  - We will model semantics of interfaces
  - "If h(t3) changes, is B still correct?

This model allows us to reason about compatibility

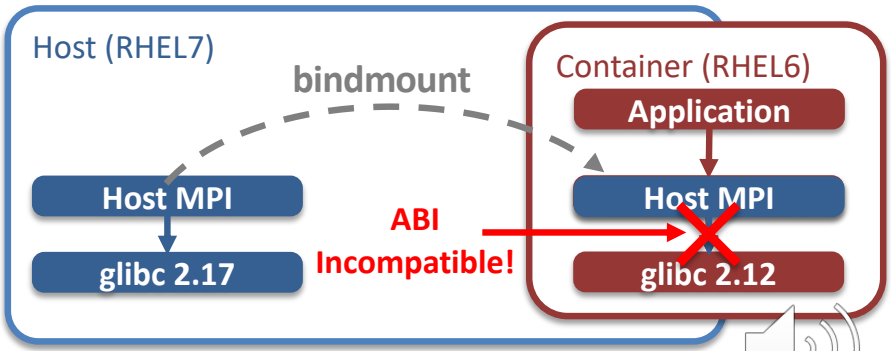
A common ABI issue: containers with hardware dependencies

- 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

Isolated container usage model



Attempt to use host MPI gone wrong



To reason about ABI, we need better binary tools

OS-provided libraries

Stripped proprietary binaries

Prebuilt packages

- 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

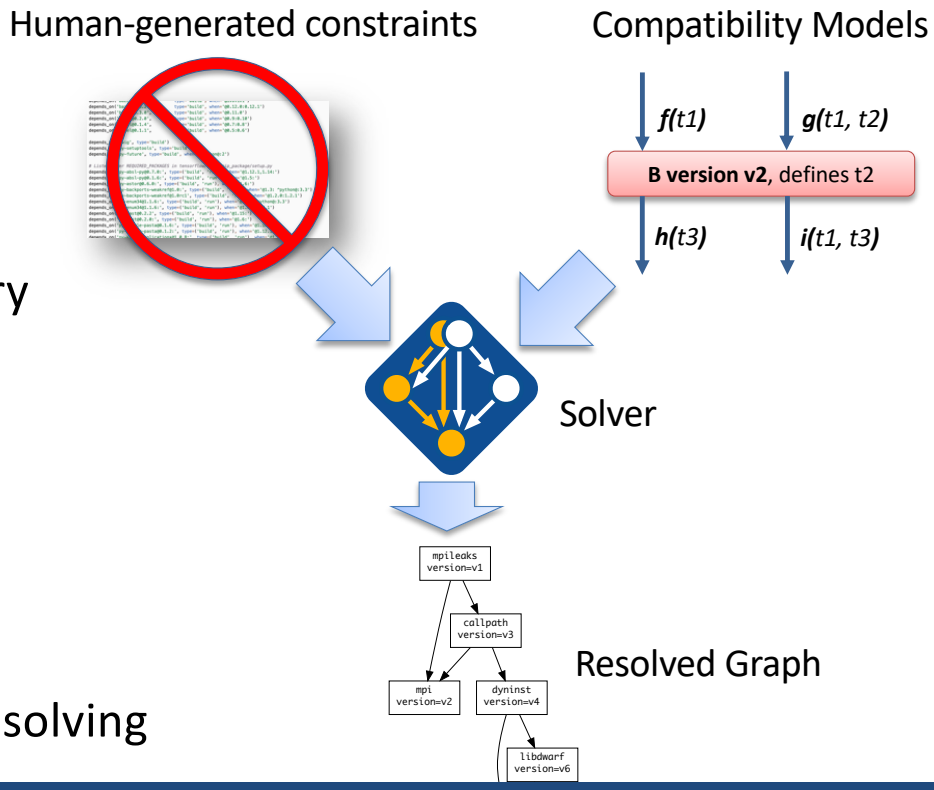
We need better debug information to rely on prebuilt packages.

With the right tools, we can extract more compatibility information from binaries

- 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**