Learning to Predict and Improve Build Successes in Package Ecosystems

CASC
Center for Applied Scientific Computing

4-16-2024

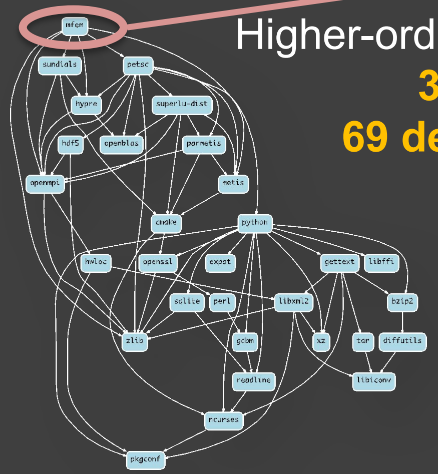
Harshitha Menon*, Daniel Nichols*, Abhinav Bhatele, Todd Gamblin

LLNL-PRES-856510
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

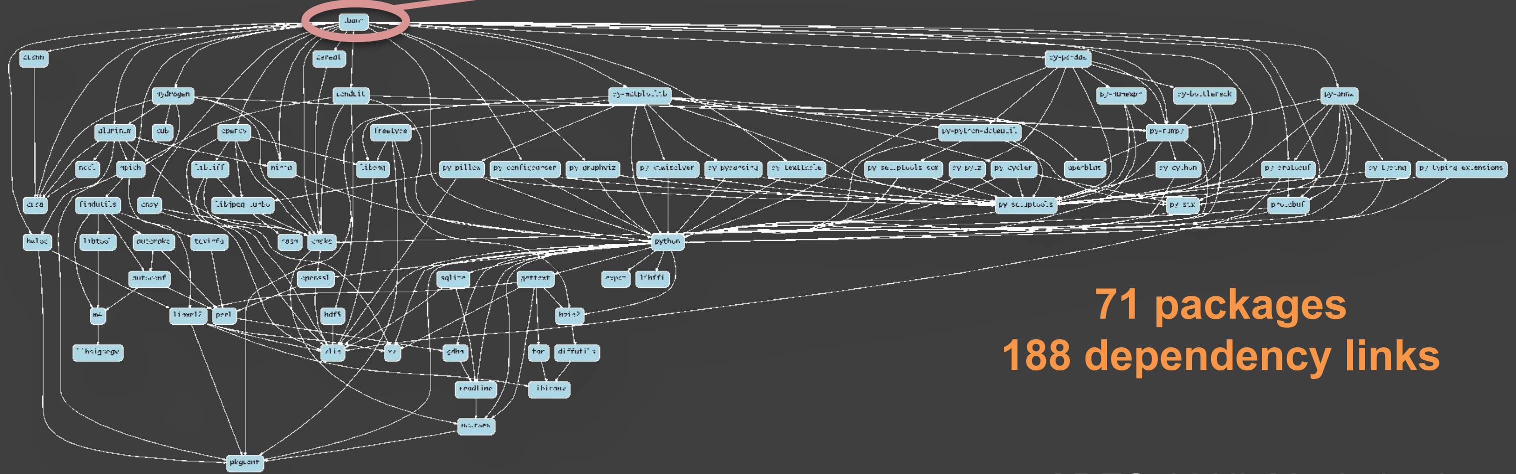
Lawrence Livermore National Laboratory

Codes have tens or hundreds of dependency libraries

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

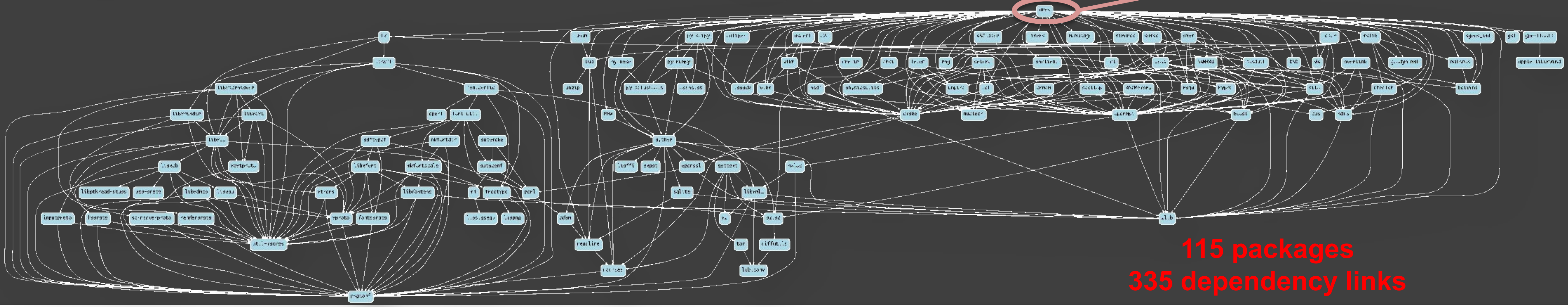


LBANN: Neural Nets for HPC



71 packages
188 dependency links

ARES: LLNL Multi-physics



115 packages
335 dependency links

Transitive dependency requirements can cause cascading errors

```
    umpire                                umpire
   /      \        upgrade               /      \
camp@1.0   blt@1.0    --->     camp@1.1   blt@1.1
   \      /                              \      /
    cmake@3.19                          cmake@3.21.0
```

- blt@1.0 requires cmake >= 3.18, but is incompatible with cmake@3.21.0 due to an unknown bug
- camp@1.0 depends on cmake@3.19 or higher, but camp@1.1 depends on cmake@3.21 or higher
- The umpire developers want to use camp@1.1 for its new features
- Upgrading camp to 1.1 pushes cmake to the latest 3.21.0 will cause the build to fail
- We need to use blt@1.1 to make this work.

Package maintainers have to build several versions to find a working configuration

Developers integrate large software stacks manually

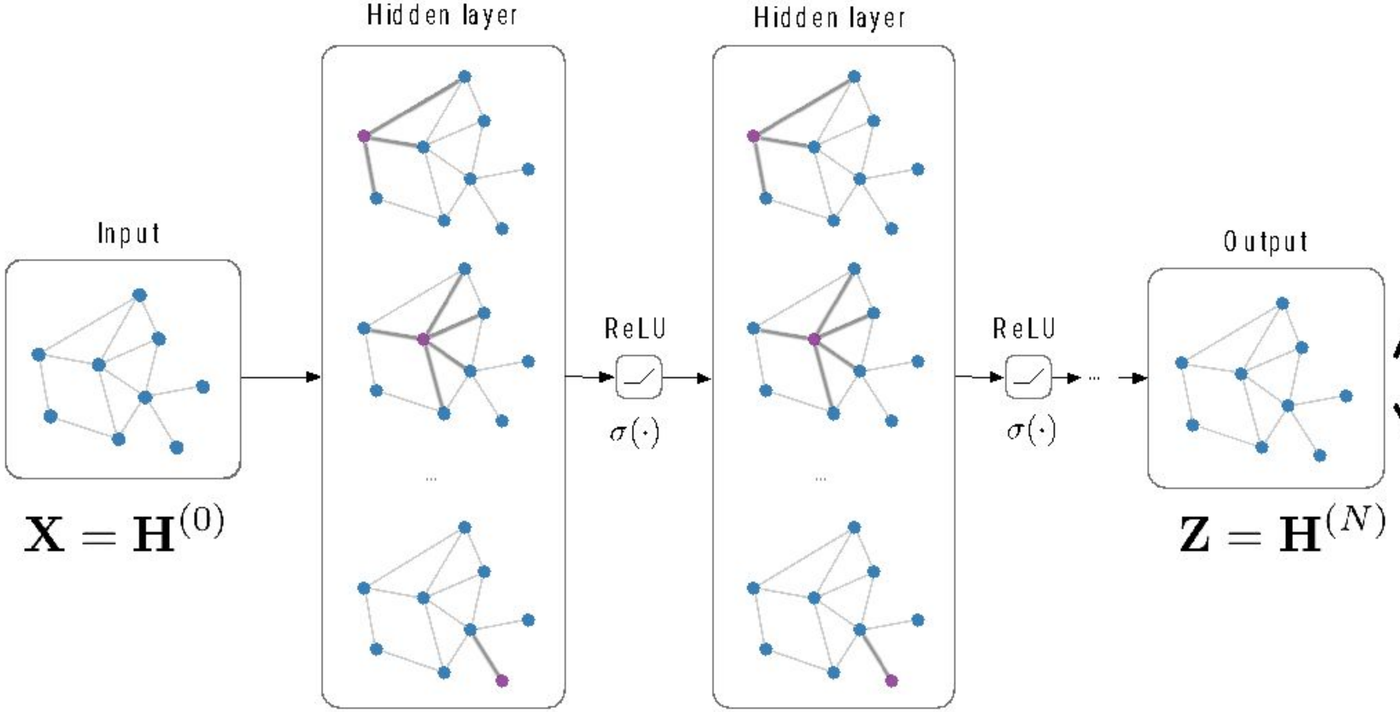
- Package managers rely on developers to specify constraints
- Finding compatible set of versions for packages is hard
- In HPC, there are many more parameters to adjust
  - Version, compiler, ABI, build options, microarchitecture, GPU capability, etc.
- We solve this problem repeatedly by trial and error
  - Incompatibilities are not known in advance

Goal

Use historical build data to understand build incompatibilities and predict build outcomes with high accuracy.

- ***RQ1*** – Can a GNN predict the build outcomes of various package configurations with high accuracy?
- ***RQ2*** – Can self-supervised pre-training be utilized to reduce the need for expensive build data to train the model?
- ***RQ3*** – How can predicted build outcomes be utilized to select better package configurations and increase the likelihood of a successful build?

Graph Neural Network (GNN)



$$\mathbf{H}^{(l+1)} = \sigma\left(\hat{\mathbf{A}}\mathbf{H}^{(l)}\mathbf{W}^{(l)}\right)$$

Node classification:

$\mathrm{softmax}(z_n)$

e.g. Kipf & Welling (ICLR 2017)

Graph classification:

$\mathrm{softmax}(\quad_n z_n)$

e.g. Duvenaud et al. (NIPS 2015)

Link prediction:

$p(A_{ij}) = \sigma(z_i^T z_j)$

Kipf & Welling (NIPS BDL 2016)

"Graph Auto-Encoders"

* figure from Thomas Kipf, University of Amsterdam

Graph Neural Networks (GNN) for Build Prediction

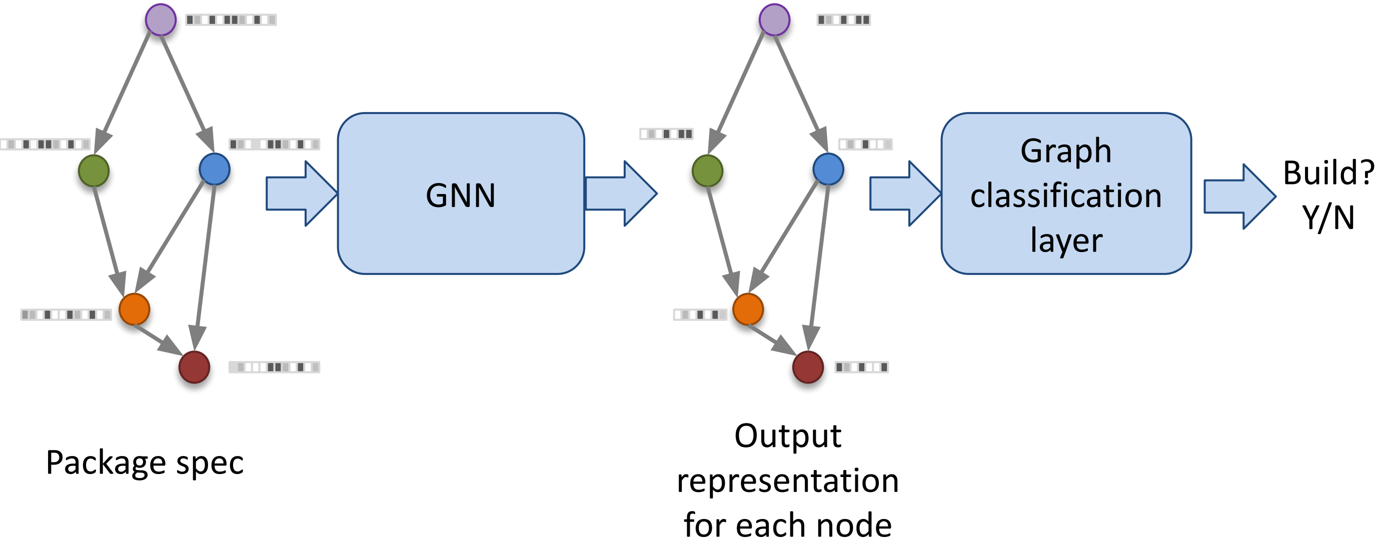
Why GNN?

- GNNs are highly effective for analyzing graph structured data.
- Build outcome depends on the relationship between packages in a dependency graph

Problem Definition

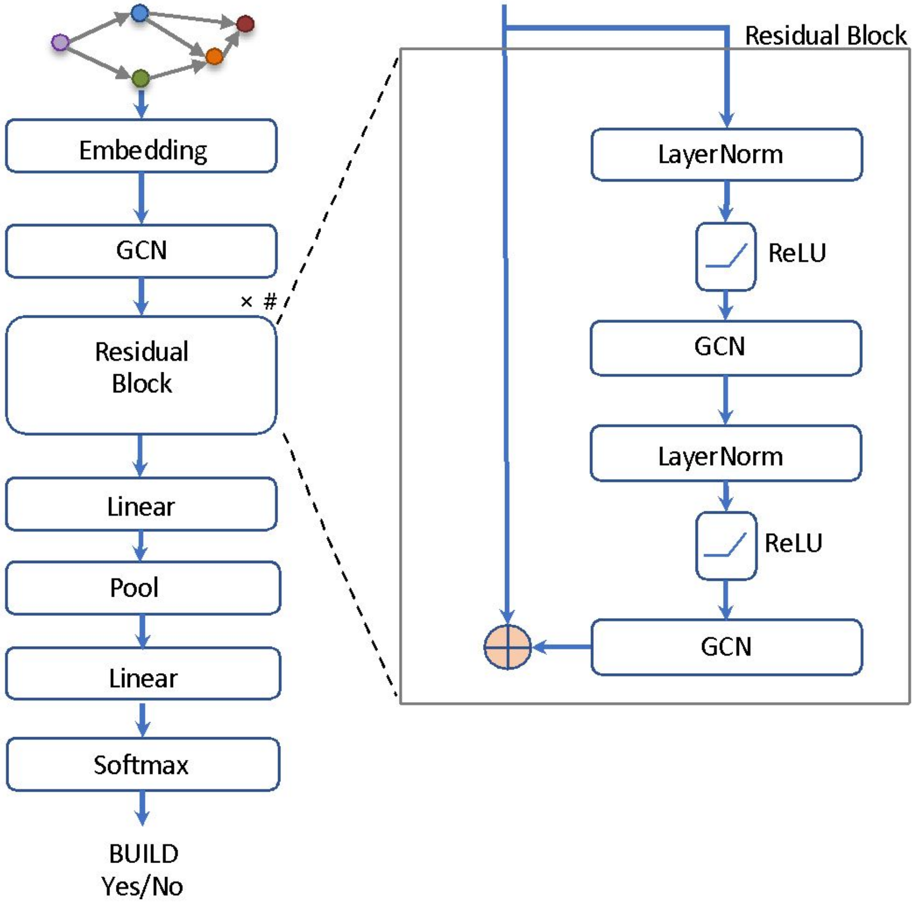
- The package dependency graph is a directed acyclic graph (DAG)
- Graph is represented as $G = (V, E)$, where V is the set of nodes representing the packages and E is the set of edges capturing the dependencies.
- We cast the build success prediction problem as a supervised learning problem
- **Goal**: learn a model to predict the build outcome

Overview of build outcome prediction using GNN



- Each configuration is represented as a graph
- Node features incorporate information about packages (which package and version)
- Layers GCN
- Final layer does a global pooling to predict whether this configuration builds or not.

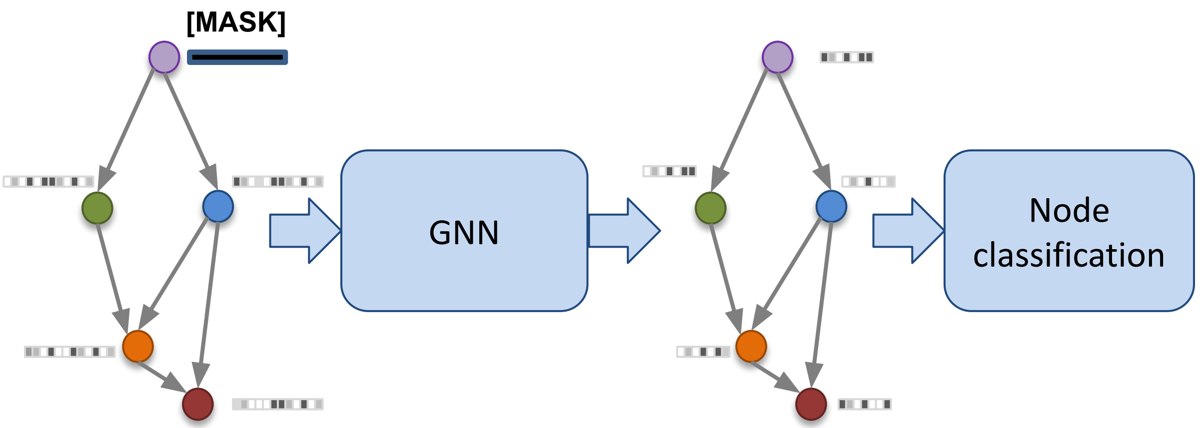
BuildCheck GNN Architecture

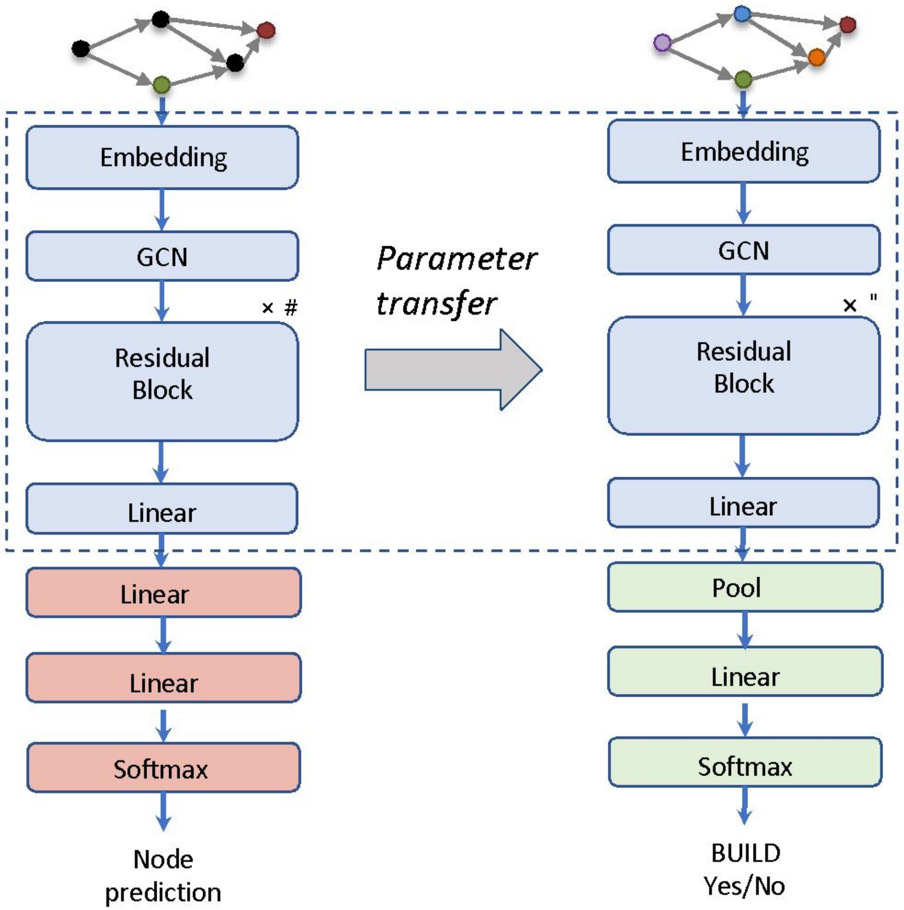


Main Components

- Multiple Graph Convolutional layers.
- Residual block: aids in training deeper networks.
- Embedding layer: maps package information into a continuous vector space
- Pool layer: computes the average of all node features and creates a representation of the entire graph

Self-Supervised Pre-training Task for Learning Node Embeddings for Downstream tasks





Pre-trained package dependency model can be used for build prediction with fine-tuning

Evaluation on Extreme Scale Scientific Software Stack (E4S)

- Evaluated on 367 unique packages in E4S ecosystem
  - different programming languages such as C/C++, FORTRAN, Python, Lua, and others.
  - With tens and hundreds of dependencies
- We explored 45,837 unique build configurations
- Utilize Spack for managing software packages and creating the dataset

ECP
EXASCALE COMPUTING PROJECT
E4S

Spack enables Software Distribution for HPC

- Spack is a flexible package manager which automates the build and installation of scientific software
- Packages are parameterized, so that users can easily tweak and tune configurations

```
$ spack install hdf5@1.10.5
$ spack install hdf5@1.10.5 %clang@6.0
$ spack install hdf5@1.10.5 +threadssafe
$ 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
```

- Spack specs can constrain versions of dependencies
- Spack concretizer solves the version constraints to ensure consistent builds

github.com/spack/spack

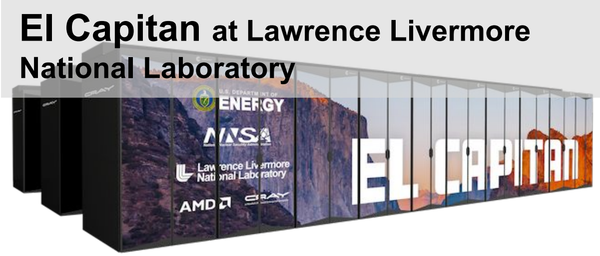
Spack is critical for DoE's Exascale Computing Project mission to create robust exascale software ecosystem

Building Software on High Performance Computing (HPC) Systems

- We want to build software from source for performance
  - Use fast compilers
  - Use vendor provided libraries
  - Need to use the host GPU

- Often need to build multiple variants of the same package
  - On new machines, first time builds

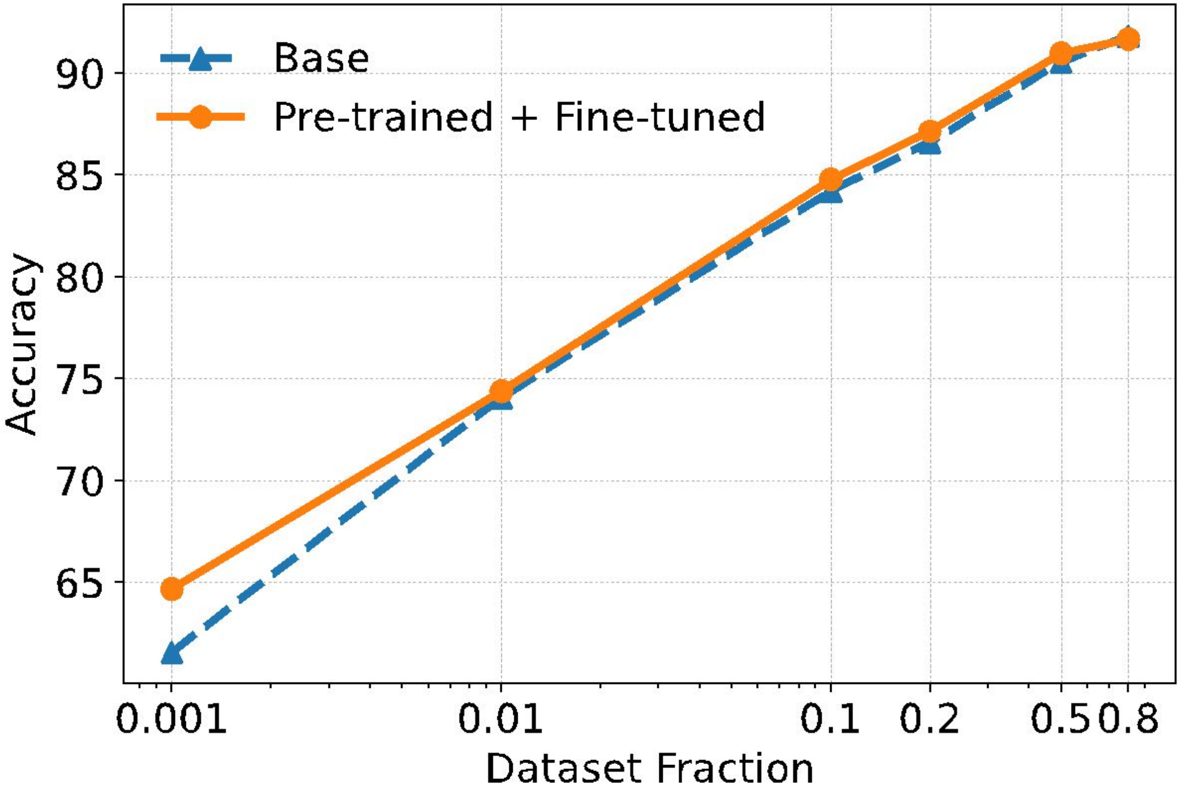
2 ExaFlops Peak
AMD Zen / Radeon



1.1 ExaFlops Peak
AMD Zen / Radeon

500 PetaFlops Peak
Fujitsu/ARM a64fx

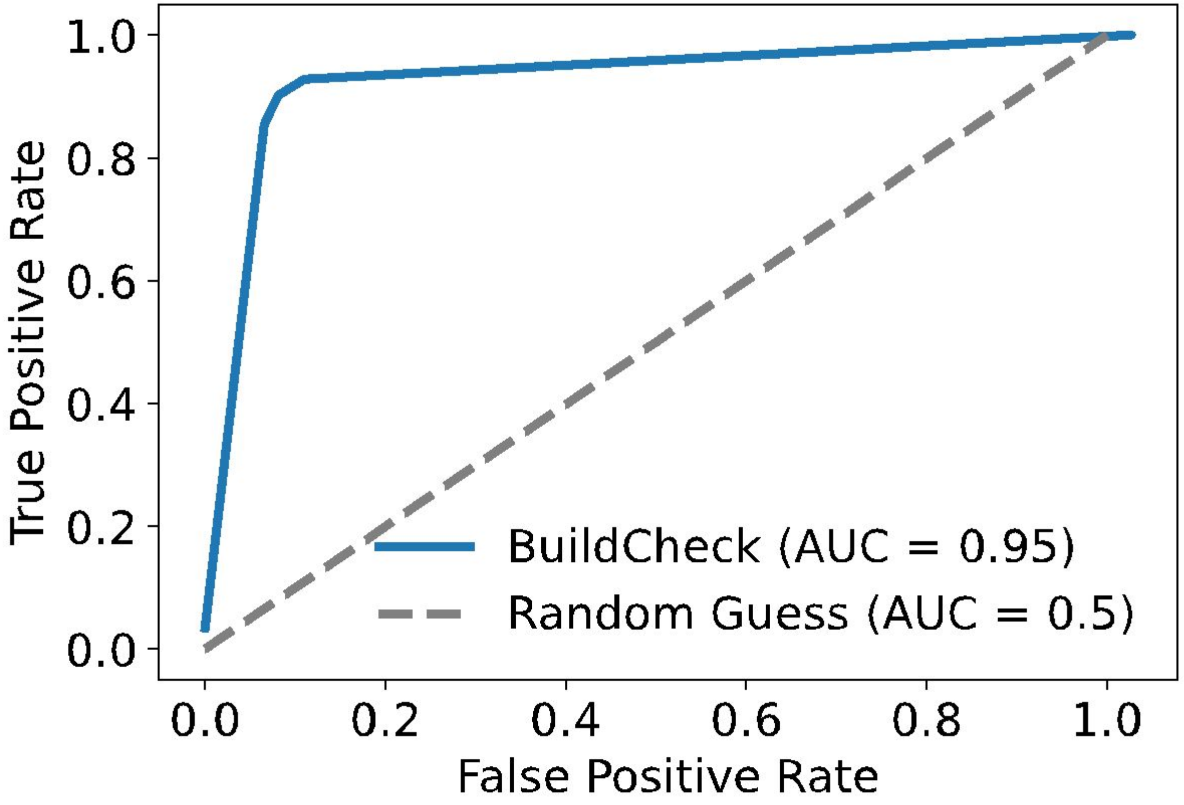
Evaluation



- Base model achieves an accuracy of 91%
- We can achieve better accuracy with little build data when we do self-supervised pre-training first

Our model achieves an accuracy of 91% on E4S build dataset

Evaluation

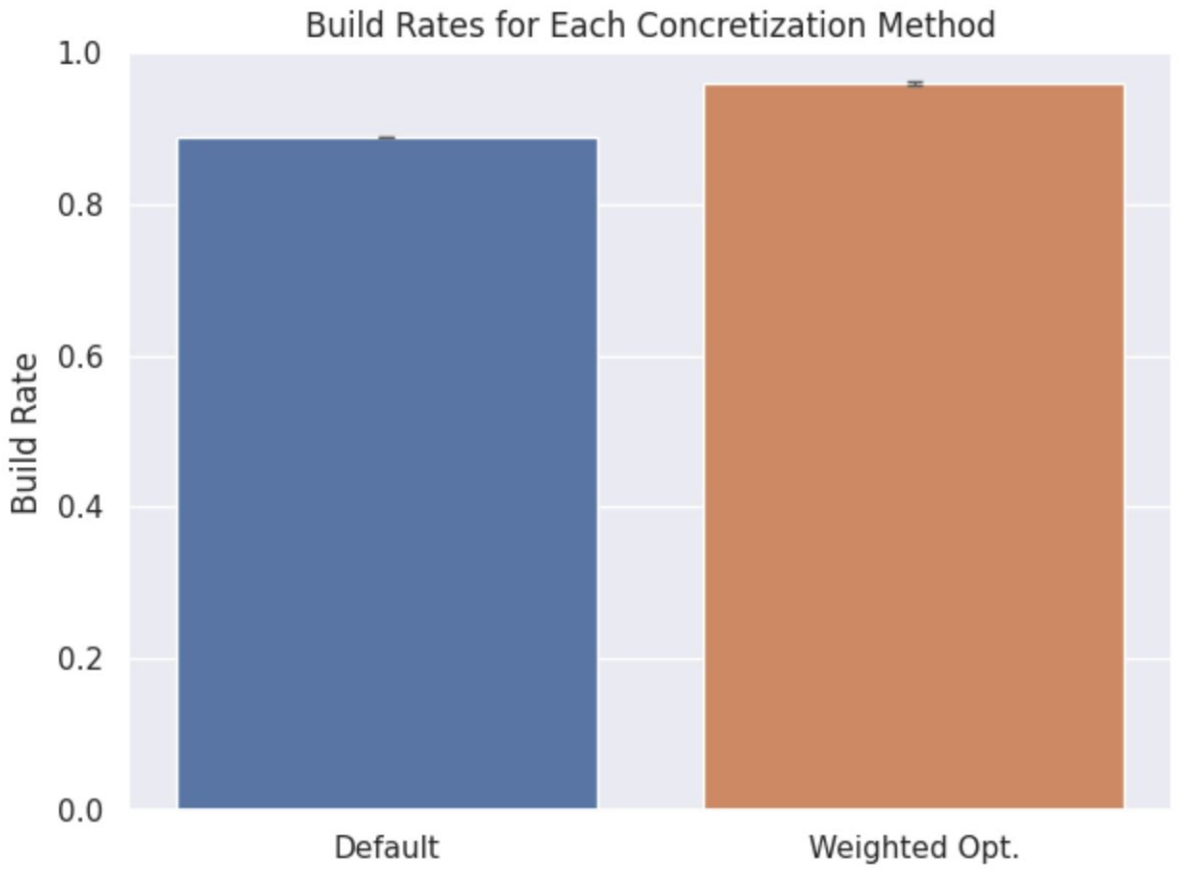


- False positives result in long, expensive builds
- False negatives result in not attempting builds that would succeed

Our model achieves an accuracy of 91% on E4S build dataset

Improving Builds in Spack with Predicted Build Outcomes

- Spack uses Answer Set Programming to solve dependency constraints
- Currently *prefers* most recent versions using *optimization*
- Change logic program to prefer *more probable parent-child pairs*
- Re-build E4S packages



Improves successful build ratio from 89% to 96%

Conclusion and Future Work

- ***RQ1*** – Demonstrated how to combine the capabilities of GNNs and advanced package management technologies to predict build outcomes with high accuracy
- ***RQ2*** – Demonstrated the effectiveness of self-supervised pre-training to reduce the amount of build data necessary for training
- ***RQ3*** – Improved the rate of successful package build using predicted build outcomes to guide version selection
- Our model can eliminate very expensive trial-and-error exercise to find working builds
- Model more build outcomes than success
- Incorporate probabilistic reasoning into Spack's solver

CASC
Center for Applied Scientific Computing

Lawrence Livermore National Laboratory

This work was prepared by LLNL under Contract DE-AC52-07NA27344 and was supported by the LLNL-LDRD Program under Project No. 21-SI-005.