BLUE WATERS-GEO

MAPPING AND MODELING THE WORLD

July 14, 2020

Blue Waters Python Provisioning

Roland Haas, Senior Research Programmer
rhaas@illinois.edu

Enhanced Python Provisioning

Why is it needed?
Reduces the level of effort by staff and users, improves user experience, and improves performance.

Who does it impact and when?
Users benefit by spending less time attempting to install Python packages and use optimally configured packages. The center benefits by reducing the number of service requests.

How does this affect science efforts?
This practice improves the effectiveness of science teams and allows for complete workflows to be run on the system.

Usage cases of Python for HPC by task

- preparing your input deck
  - create input files based on physical parameters
  - create directory structures
  - submit simulations
  - mostly string handling and scripting
- process simulation results
  - combine data from checkpoints
  - interactively explore data
  - distill scientific results from data
  - produce plots and other representation of results
  - mostly serial but possible bag-of-task parallelism
- orchestrate simulations
  - set up data for multi-stage simulations
  - check success of each step
  - start MPI parallel simulation code
- glue code for binary code
  - Python handles simulation and analysis infrastructure tasks
  - most lines of code are Python
  - most execution time is in compiled code
- Python for science code
  - no custom compiled code
  - Python code does actual science calculations

Design goals of an HPC python stack

- Provide multiple Python versions
  - 2.7
  - 3.X
- Maintainable by center staff
  - Consistent set of Python modules
  - Track which modules are used
- Extensible by users
- Low impact on system even when used by many independent jobs
- Use optimized numerical libraries, code compiled for cluster CPU where appropriate
  - LAPACK, BLAS
  - PETSc
  - TensorFlow, PyTorch
  - HDF5
  - MPI
- Provide majority of Python modules used on system
  - Domain specific packages
  - Hard to compile packages

Challenges providing an HPC Python environment

Package management

- Many Python packages in use
  - Deep dependency stacks
  - Too many to map 1:1 to environment modules
- Mix of Python and compiled code dependencies
- Integrate with HPC libraries

IO load

- Python module system very metadata-IO heavy
  - Bulk size is small
- Mostly open / stat calls
  - No coordination between Python processes in job
- Runtime loading of libraries and modules

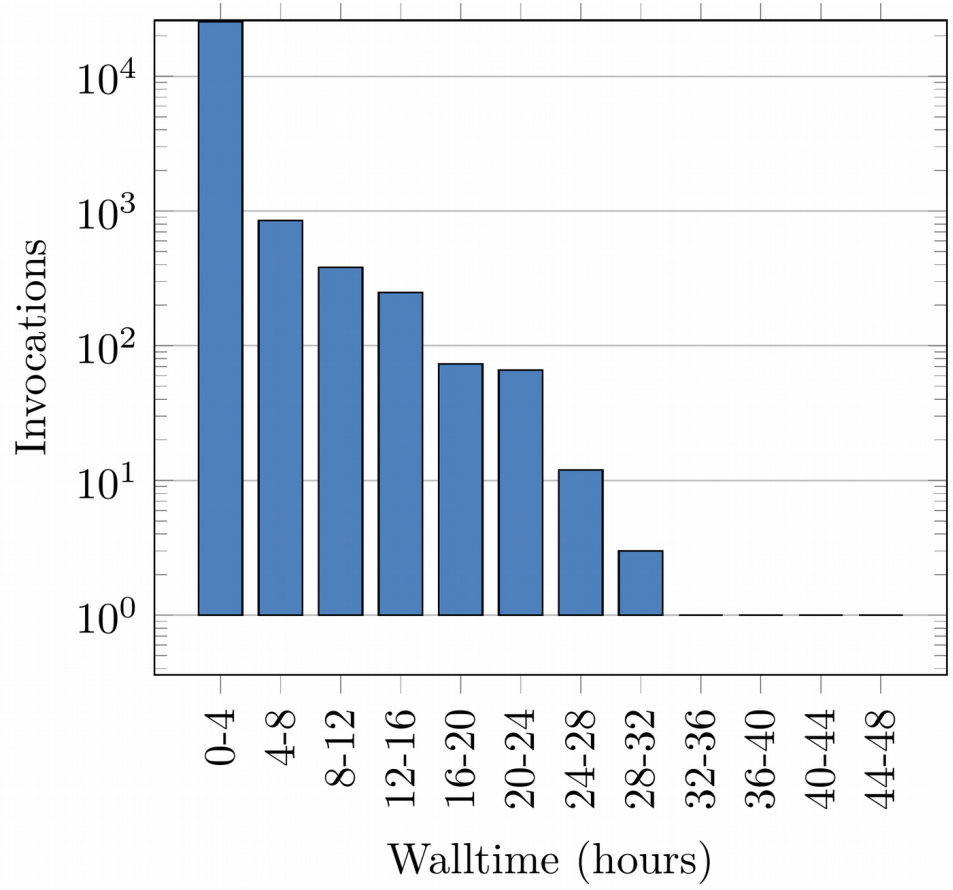
Python package management solutions

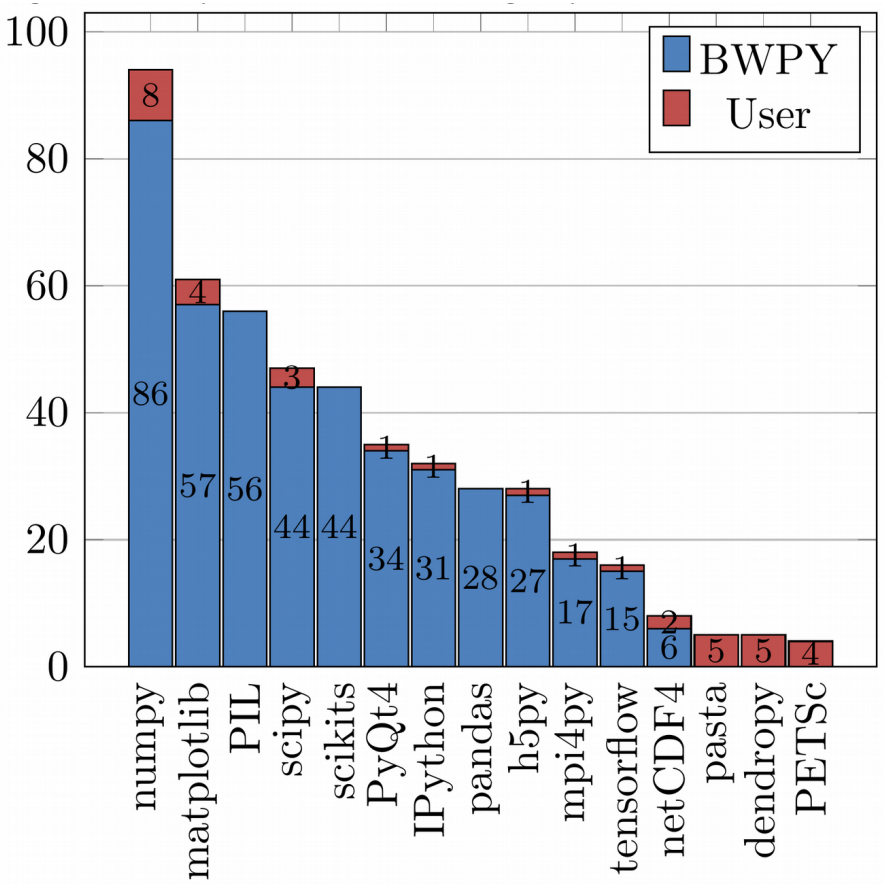
- "monolithic" Python module for users
  - Use package manager to select packages in module
  - Allow user extensions
- Python only vs. compiled code dependencies
- Most common
  - Conda
  - Pip
  - EasyBuild
  - Spack
- Less common
  - Gentoo ebuilds

name	installs	Initial target audience	# of packages	Ease of use for staff
Conda	Binary only	Laptops, single user	684 (Linux, x86_64)	Very easy if it works
pip	From source, mostly Python, some compiled code	End users, Python	244,876 (number of projects)	Easy if it works
EasyBuild / Spack	From source, mostly compiled code, Python as well	HPC staff	1978 / 4347	Medium
Gentoo ebuilds	From source	Laptop, system wide	18,852	Hard

Tracking use of Python modules

- Track which modules are used
- Track how many node-hours are used
- Using hook in `sitecustomize.py`





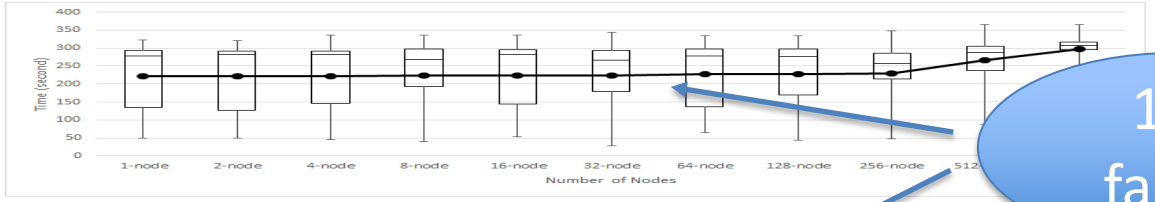
Challenges of using Python in an HPC environment

- Python startup and the import statement are very metadata intensive

```
python3 -c 'import numpy'
```

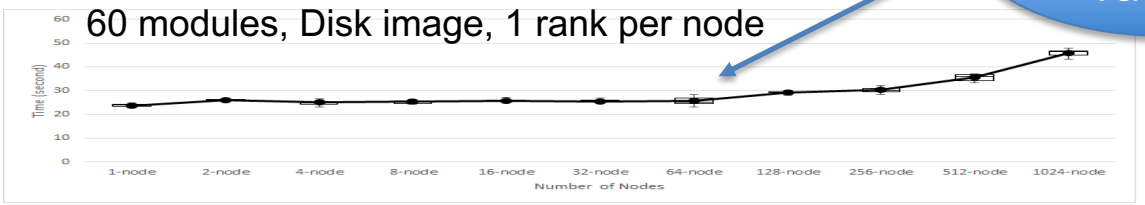
- has 1600 open & stat calls
  - per MPI rank, hitting a (potentially) single metadata server
- e.g. a 1ms response time, 1024 ranks → 1,600s startup time
  - makes shared file system slow for every user on the system

60 modules, Lustre, 1 rank per node



10x faster

60 modules, Disk image, 1 rank per node



- Reduce metadata server load
  - solved on Blue Waters using BWPY disk image wrapper (on GitHub)
  - Similar solution possible using containers
- Overall improvement of responsiveness

Extra slides

Best Practice: Enhanced Python Provisioning

How is this Best Practice different from the common practice?

- Typical Python installations are composed of base packages plus the “top 6” packages such as: numpy, scipy, matplotlib, mpi4py, and pycuda.

Is it measurable, and if so, how?

- Monitor number of Python service requests (tickets) compared to the number of packages available.

Example of use

- Over 300 Python packages are available, including mpi4py, pycuda, h5py, netcdf4-python, machine learning, geo-inf packages and pandas. The core numerical libraries are linked against Intel’s MKL, and the stack is built for running on both login and compute nodes. Python tickets reduced significantly.
- Anaconda provides similar package breadth and flexibility geared towards binary packages.

Pros and cons of using Python in science projects

- Very low learning curve
  - for you
  - for your students
- Quick turnaround while developing
  - fully open source
  - no licensing costs
  - encourages sharing code
- large number of scientific packages:
  - numpy, scipy
  - PyTrilinos, petsc4py, Elemental, SLEPc
  - Pytorch, Tensorflow, GDAL
  - mpi4py, h5py, netcdf
  - matplotlib

- Very low learning curve
  - low quality code possible
- not initially designed for HPC
  - most developers aren't scientists
  - Python itself is not very fast
- Large startup costs, hard on cluster IO subsystem
- not always backwards compatible, even between minor versions
- duck-typing makes code validation hard, errors only detected at runtime

Why use Python in an HPC environment?

- everybody (else) is already using it
  - including your students, whether you like it or not ...
  - large body of documentation available on the web
- Python's design principles:
  - Beautiful is better than ugly.
  - Explicit is better than implicit.
  - Simple is better than complex.
  - Readability counts.

  make for code well suited to scientific projects
- Python was originally designed to be usable as a glue language
  - highly extensible
  - can bind to many compiled languages: C, C++, Fortran

name	installs	Initial target audience	Extensible by user	Ease of use for staff
Conda	Binary only	Laptops, single user	Yes	Very easy if it works
pip	From source, mostly Python, some compiled code	End users, Python	Yes	Easy if it works
EasyBuild / Spack	From source, mostly compiled code, Python as well	HPC staff	Yes	Medium
Gentoo ebuilds	From source	Laptop, system wide	Via pip	Hard

name	# of packages	Package description language	Site-local package repository	Ease for staff to add new packages
Conda	684 (Linux, x86_64)	Yaml files	Yes	Hard
pip	244,876 (number of projects)	Python	No	N/A
EasyBuild / Spack	1978 / 4347	Home-grown (Python) / Python	Yes / Yes	Easy
Gentoo ebuilds	18,852	Home-grown (bash)	Yes	Medium