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

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

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

![Graph showing module usage distribution](image-url)
Challenges of using Python in an HPC environment

- Python startup and the import statement are very metadata intensive
- 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

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

- 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**
- **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**
- **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
<table>
<thead>
<tr>
<th>name</th>
<th>installs</th>
<th>Initial target audience</th>
<th>Extensible by user</th>
<th>Ease of use for staff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conda</td>
<td>Binary only</td>
<td>Laptops, single user</td>
<td>Yes</td>
<td>Very easy if it works</td>
</tr>
<tr>
<td>pip</td>
<td>From source, mostly Python, some compiled code</td>
<td>End users, Python</td>
<td>Yes</td>
<td>Easy if it works</td>
</tr>
<tr>
<td>EasyBuild / Spack</td>
<td>From source, mostly compiled code, Python as well</td>
<td>HPC staff</td>
<td>Yes</td>
<td>Medium</td>
</tr>
<tr>
<td>Gentoo ebuilds</td>
<td>From source</td>
<td>Laptop, system wide</td>
<td>Via pip</td>
<td>Hard</td>
</tr>
<tr>
<td>name</td>
<td># of packages</td>
<td>Package description language</td>
<td>Site-local package repository</td>
<td>Ease for staff to add new packages</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------------</td>
<td>------------------------------</td>
<td>------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>Conda</td>
<td>684 (Linux, x86_64)</td>
<td>Yaml files</td>
<td>Yes</td>
<td>Hard</td>
</tr>
<tr>
<td>pip</td>
<td>244,876 (number of projects)</td>
<td>Python</td>
<td>No</td>
<td>N/A</td>
</tr>
<tr>
<td>EasyBuild / Spack</td>
<td>1978 / 4347</td>
<td>Home-grown (Python) / Python</td>
<td>Yes / Yes</td>
<td>Easy</td>
</tr>
<tr>
<td>Gentoo ebuilds</td>
<td>18,852</td>
<td>Home-grown (bash)</td>
<td>Yes</td>
<td>Medium</td>
</tr>
</tbody>
</table>