Python Best Practices in HPC

Roland Haas (NCSA)
Email: rhaas@illinois.edu

ILLINOIS
NCSA | National Center for
Supercomputing Applications

Why use Python in HPC?

- 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

Pros and cons of using Python in your science project

- 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`
  - `mpi4py, h5py, netcdf`

- 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

Usage cases of Python for HPC by task

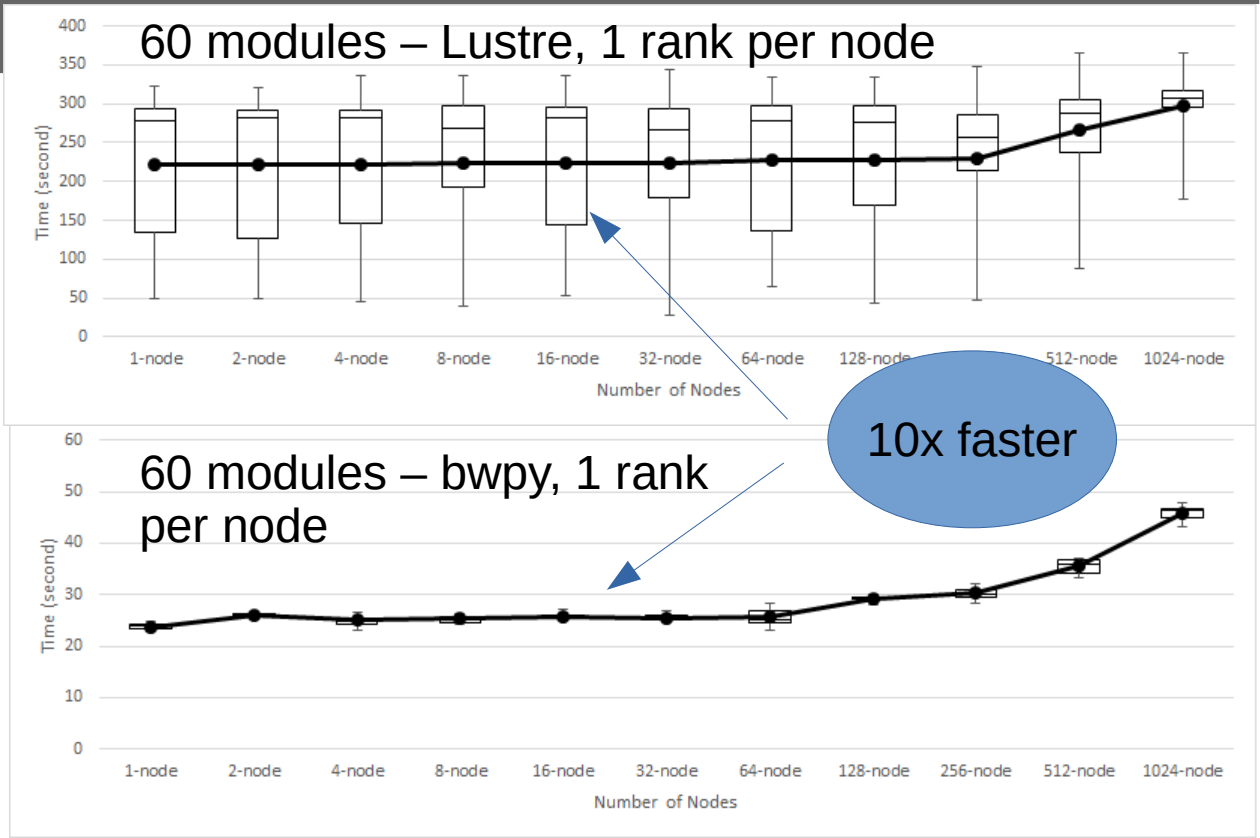
- preparing your input deck
  - create input files based on physical parameters
  - create directory structures
  - submit simulations
  - mostly string handling and scripting
- process simulation results
  - combine data from checkpoints
  - interactively explore data
  - distill scientific results 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 in simulation binary
  - Python handles simulation 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 or public packages do actual science calculations

Python startup time issues

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

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

- has 1600 `open` & `stat` calls
  - per MPI rank, hitting a 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



- solved in BWPY for provided modules
- for you own modules
  - install to `/dev/shm/$USER` on login node
  - tar up `/dev/shm/$USER`
  - extract **tarball** to `/dev/shm/$USER` on compute nodes, put first in `$PYTHONPATH`

Workflows in python

- for simple bag-of-tasks workflows, use `mpi4py`'s `MPICommExecutor` (see BWPY presentation)
  - do **not** use 1000 `aprun -n1 python`
- Python workflows in Blue Waters webinars series:
  - Parsl, modern, pure python, standalone
  - Pegasus, very mature, builds on HTCondor
- IO challenge
  - no file system likes millions of tiny files. Lustre is no exception
  - store temporary files in `/dev/shm` on compute nodes
  - pre-stage files in the background using Globus, has a python interface

MPICommExecutor

```
from mpi4py import MPI
from mpi4py.futures import MPICommExecutor

def sqr(x): return x*x
data = range(21)
with MPICommExecutor(root=0) as executor:
  if executor is not None: # on root
    squared = executor.map(sqr, data)
    print(squared)
```

Parsl

```
from parsl import App, DataFlowKernel
import parsl.configs.local as lc
dfk = DataFlowKernel(lc.localThreads)

@App('python', dfk)
def sqr(x): return x*x
data = range(21)

squared = map(sqr, data)
print([i.result() for i in squared])
```

Numerical computations using python

- `numpy` the de-facto standard way to handle numerical arrays in python
  - N-dimensional arrays of integer, real and complex numbers
  - linear algebra (BLAS, LAPACK), FFT, random numbers
  - linkages to C/C++/Fortran
- `scipy` provides higher level functions
  - optimization
  - integration
  - interpolation
  - signal and image processing
  - ODE solvers

- both `numpy` and `scipy` leverage BLAS, LAPACK, FFT, FITPACK
  - sub-optimal performance if those are incorrectly build
  - BWPY does “the right thing”
  - pip does not (usually)
- `PyTrilinos, petsc4py, Elemental, SLEPc` build on these

```
import numpy as np
A = np.random.random((1000,1000))
b = np.random.random((1000,))
c = A*b

pip:  0.02s
BWPY: 0.004s
```

5x faster

Computing in python code

- How CPython works
  - compile script to bytecode
  - execute one line of byte code after the other
- CPython is designed for maintainability, not speed
  - no look ahead
  - no parallelism (threads, vectorization)
  - hard to change this due to duck typing
- Alternatives
  - pypy
  - numba
  - Cython

- Not all are equally well suited for all tasks
  - pypy does not deal well with `numpy`

```
import numpy as np
a = np.zeros(10000)
for i in range(10000):
  a[i] = np.sqrt(i)
```

is 2x slower in pypy than CPython (uses numpy-pypy)

```
a = list()
for i in range(1000):
  a.append(str(i))
```

is 10x faster in pypy than CPython

Numba and Cython

- Numba is a just-in-time compiler for numerical operations in Cpython
  - needs (simple) annotations
  - deals well with `numpy`

```
import numpy as np
from numba import jit

@jit
def my_sqrt():
  a = np.zeros(10000)
  for i in range(10000):
    a[i] = np.sqrt(i)
```

12x faster than plain CPython

- Cython compiles python-like code to C, designed to link C extensions to python
  - load result as module
  - do threading and parallelization in C code

```
from libc.math cimport sqrt

def my_sqrt():
  cdef int i
  cdef double a[10000]
  for i in range(10000):
    a[i] = sqrt(i)
```

481x faster than plan CPython

Calling compiled code (the easy way)

- numpy has convenience code to link to Fortran code
  - very easy to use (much easier than C)

```
SUBROUTINE FIB(A,N)
INTEGER N
REAL*8 A(N)
DO I=1,N
   IF (I.EQ.1) THEN
      A(I) = 0.0D0
   ELSEIF (I.EQ.2) THEN
      A(I) = 1.0D0
   ELSE
      A(I) = A(I-1) + A(I-2)
   ENDIF
ENDDO
END SUBROUTINE
```

```
$ python -m numpy.f2py -m myfib \
  -c fib.f90
```

```
import numpy
import myfib

a = numpy.zeros(8, 'float64')
myfib.fib(a)
print(a)
```

For C code, you may even want to write a Fortran wrapper

from http://scipy-lectures.org

More on using compiled modules

- Cython: https://scipy-lectures.org/advanced/interfacing_with_c/interfacing_with_c.html#id13
- f2py (very easy!): https://docs.scipy.org/doc/numpy/user/c-info.python-as-glue.html#f2py
- SWIG: http://swig.org/Doc1.3/Python.html
- Boost – interferes with HDF5 on BW
- Ctypes: https://scipy-lectures.org/advanced/interfacing_with_c/interfacing_with_c.html#id6
- Numpy bindings in C/C++: https://dfm.io/posts/python-c-extensions/

Code profiling

- Profile you code to find out where it spends most time. Assuming that it *must* be your innermost loop is dangerous...
- object code profilers like CrayPat profile the python interpreter, but not your python code
- Python comes with a built in profiler in the `cProfile` module
- included in BWPY
  - default is function level granularity
  - add extra profiling modules and analysis tools in a virtualenv
- can be as simple as `python -m cProfile loop.py`

- output profile using `-o` switch for in depth analysis
  - `pstats` module lets you read it

```
python -o prof.dat -m cProfile \
  loop.py

import pstats
p = pstats.Stats('prof.dat')
p.sort_stats('cumulative').\
  print_stats(5)
```

- install `line_profiler` for line-by-line usage
  - annotate functions to profile using `@profile`
  - run `kernprof -l script.py`

Code profiling example

```
$ python -m cProfile loop.py
 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1    0.019    0.019    0.477    0.477 test-profile.py:1(<module>)
      1    0.334    0.334    0.457    0.457 test-profile.py:1(loop)
      1    0.000    0.000    0.477    0.477 {built-in method builtins.exec}
1000000    0.124    0.000    0.124    0.000 {method 'append' of 'list' objects}
      1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profil...
```

```
@profile
def loop():
  a = []
  for i in range(1000000):
    a.append(i)
```

```
$ virtualenv --system-site-packages $PWD
$ pip install line_profiler
$ kernprof -l loop.py
$ python -m line_profiler loop.py.lprof
Line #      Hits         Time  Per Hit   % Time  Line Contents
     1                                           @profile
     2                                           def loop():
     3         1          5.0      5.0      0.0    a = []
     4   1000001     957889.0      1.0     44.3    for i in range(1000000):
     5   1000000    1206173.0      1.2     55.7      a.append(i)
```

Questions?

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070 and ACI-1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications.

ILLINOIS

NCSA | National Center for Supercomputing Applications

Refeferences and extra material

- This presentation is heavily based on William Scullin's presentations: https://www.alcf.anl.gov/files/Scullin-Pavlyk _SDL2018_Python.pdf
- https://github.com/bccp/nbodykit, https://wiki.fysik.dtu.dk/gpaw/
- https://bluewaters.ncsa.illinois.edu/webinars/workflows
- https://cython.org/, https://www.pypy.org/, https://numba.pydata.org/
- https://bluewaters.ncsa.illinois.edu/python, https://bluewaters.ncsa.illinois.edu/Python-profiling

Python usage by science problem

- data science, machine learning
  - Python is the dominant language
  - Lots of support, often not much scalability beyond single nodes
- image and data analysis
  - often HTC-like workflow
  - Python workflow managers avoid having to learn a new language
  - extensive image and data processing libraries for python
- “true” HPC workloads
  - Python as glue code, e.g. nbodytoolkit, GPAW
  - most code in python, C / Fortran code does heavy lifting

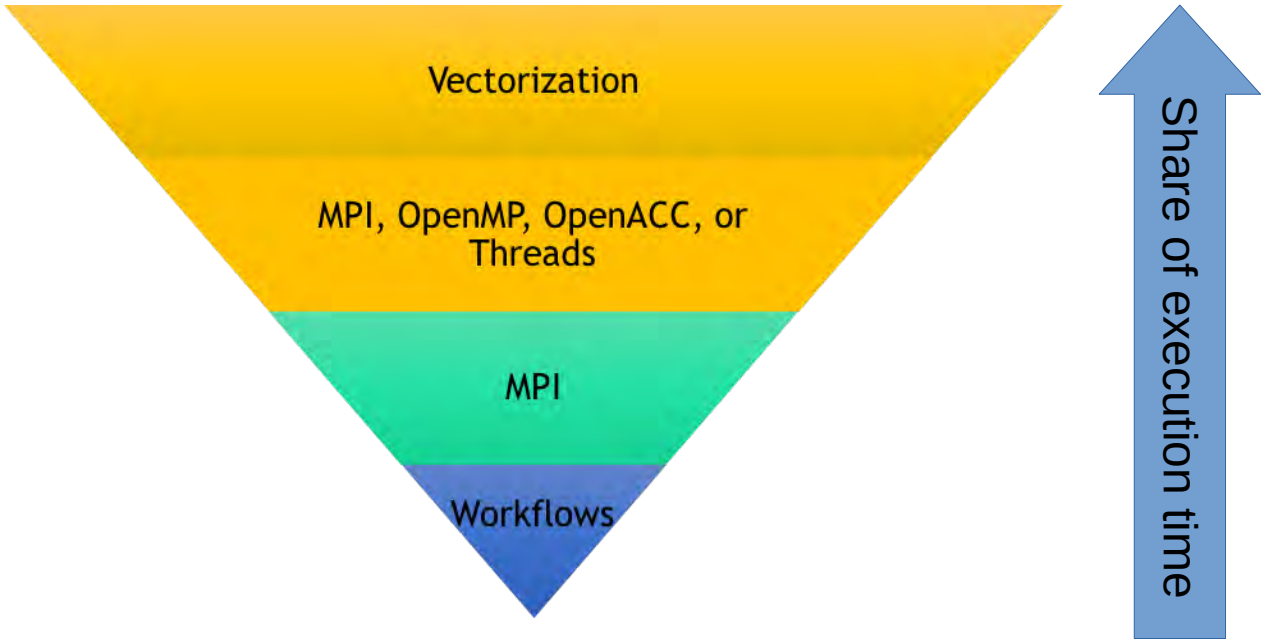


image (C) William Scullin

Cython and numpy

- Cython lets you call C code passing numpy arrays

```
void cos_doubles(double * in_array, double * out_array, int size){
    int i;
    for(i=0;i<size;i++){
        out_array[i] = cos(in_array[i]);
    }
}
```

```
cdef extern from "cos_doubles.h":
    void cos_doubles (double * in, double * out, int size)

# create the wrapper code, with numpy type annotations
def cos_doubles_func(np.ndarray[double, ndim=1, mode="c"],
                     np.ndarray[double, ndim=1, mode="c"]):
    cos_doubles(<double*> np.PyArray_DATA(in_array),
                <double*> np.PyArray_DATA(out_array),
                in_array.shape[0])
```