

PYTHON FOR HPC: BEST PRACTICES



WILLIAM SCULLIN

Assistant Computational Scientist
Leadership Computing Facility
Argonne National Laboratory

May 4th, 2017

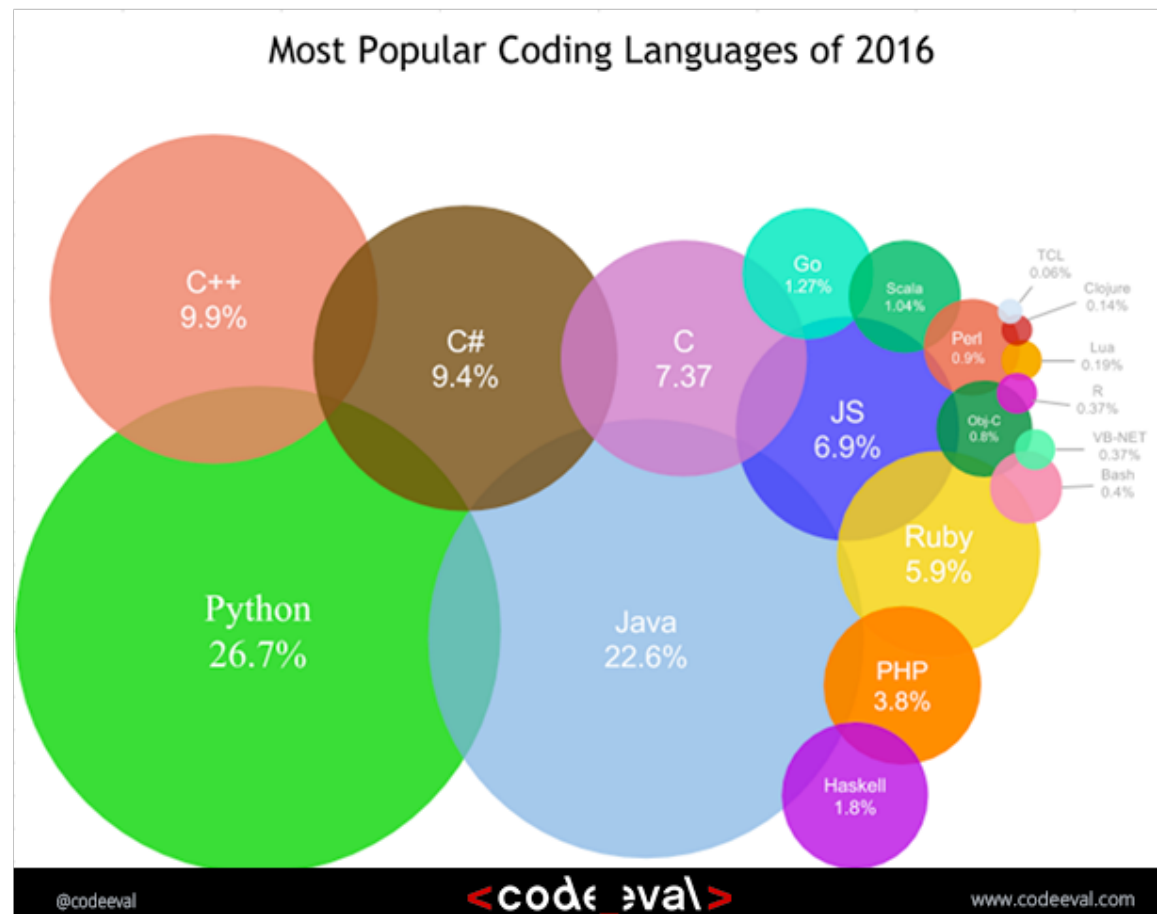
ALCF Computational Performance Workshop

**“PEOPLE ARE DOING HIGH PERFORMANCE
COMPUTING WITH PYTHON... HOW DO WE STOP
THEM?”**

- SENIOR PERFORMANCE ENGINEER

WHY THIS TALK?

- Python is popular
- It's becoming the de facto language for data science
- It's behind a large number of scientific workflows
- It's not uncommon for prototyping or even implementing production software
- We tend to make a lot of mistakes



WHY PYTHON?

- If you like a programming paradigm, it's supported
- Most functions map to what you know already
- Easy to combine with other languages
- Easy to keep code readable and maintainable
- Lets you do just about anything without changing languages
- The price is right - no license management
- Code portability
- Fully Open Source
- Very low learning curve
- Commercial support options are available
- Comes with a highly enthusiastic and helpful community

WHY NOT PYTHON?

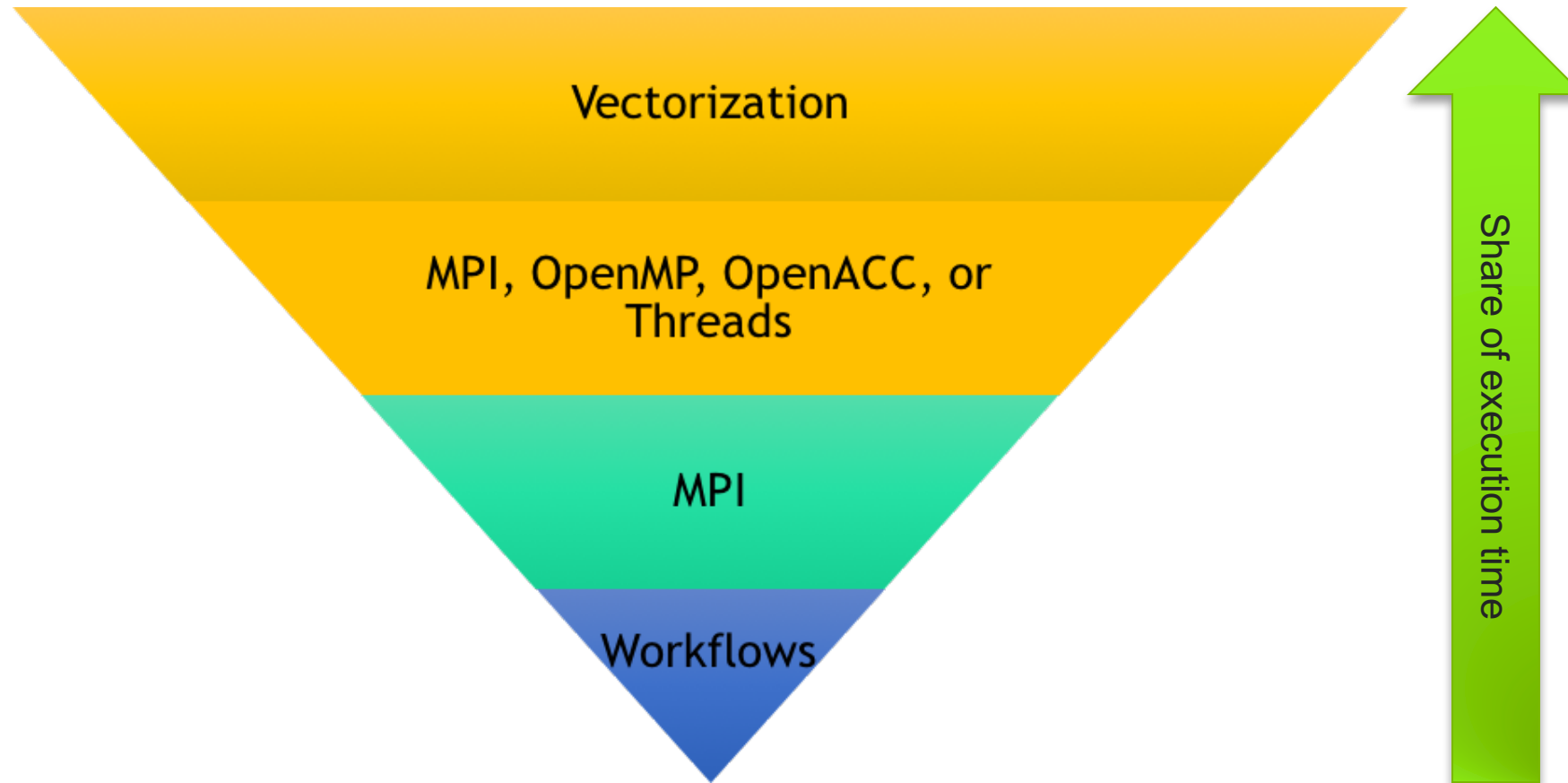
- Performance is often a secondary concern for developers and distributions
 - Most developers aren't in HPC environments
 - Most developers aren't in science environments
 - Many tools were designed to work best in generic environments
- Language maintainers favor consistency over compatibility
 - Backwards compatibility is seldom guaranteed
- Low learning curve
 - It's easy to develop a code base that works, but won't scale

PYTHON 2 OR 3?

Python was originally developed as a system scripting language for the Amoeba distributed operating system and has been developing ever since, with many backwards-incompatible changes made in the name of progress without too much delay on adoption. However, the changes from Python 2 to Python 3 were sufficiently radical that adoption has been slow going. That said:

- Python 3 is the future – and the future is here
- All major libraries now work under Python 3.5+
- Almost all popular tools work with Python 3.5+
- Python 3's loader and more of the interpreter's internals are written in Python
 - This makes loading more I/O intensive which presents challenges for scaling
 - It also makes it easier to write alternative interpreters that can be faster than CPython

WHERE DO WE WANT TO SPEND OUR TIME?



HOW DOES CPYTHON WORK?

```
Python 2.7.13 (default, Apr 23 2017, 16:50:50)
[GCC 4.2.1 Compatible Apple LLVM 7.3.0 (clang-703.0.31)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
[>>> def area_circle(r):
...     pi=3.14159
...     area=pi*r**2
...     return area
...
[>>> import dis
[>>> dis.dis(area_circle.func_code)
2          0 LOAD_CONST          1 (3.14159)
          3 STORE_FAST          1 (pi)

3          6 LOAD_FAST           1 (pi)
          9 LOAD_FAST           0 (r)
         12 LOAD_CONST          2 (2)
         15 BINARY_POWER
         16 BINARY_MULTIPLY
         17 STORE_FAST          2 (area)

4          20 LOAD_FAST           2 (area)
         23 RETURN_VALUE

>>> █
```


HOW DOES CPYTHON WORK?

```
>>> def area_circles(R):
...     A=[]
...     for r in R:
...         A.append(area_circle(r))
...     return A
...
>>> dis.dis(area_circles.func_code)
2          0 BUILD_LIST          0
          3 STORE_FAST          1 (A)

3          6 SETUP_LOOP          33 (to 42)
          9 LOAD_FAST              0 (R)
         12 GET_ITER
>>      13 FOR_ITER              25 (to 41)
         16 STORE_FAST          2 (r)

4          19 LOAD_FAST          1 (A)
         22 LOAD_ATTR            0 (append)
         25 LOAD_GLOBAL         1 (area_circle)
         28 LOAD_FAST            2 (r)
         31 CALL_FUNCTION        1
         34 CALL_FUNCTION        1
         37 POP_TOP
         38 JUMP_ABSOLUTE        13
>>      41 POP_BLOCK

5      >>  42 LOAD_FAST          1 (A)
         45 RETURN_VALUE

>>>
```

HOW DOES CPYTHON WORK?

```
[>>> def area_circles_lc(R):
[...     return [area_circle(r) for r in R]
[...
[>>> dis.dis(area_circles_lc.func_code)
 2          0 BUILD_LIST          0
          3 LOAD_FAST            0 (R)
          6 GET_ITER
    >>     7 FOR_ITER            18 (to 28)
          10 STORE_FAST           1 (r)
          13 LOAD_GLOBAL         0 (area_circle)
          16 LOAD_FAST           1 (r)
          19 CALL_FUNCTION        1
          22 LIST_APPEND         2
          25 JUMP_ABSOLUTE       7
    >>     28 RETURN_VALUE

>>> █
```

THREADS AND PYTHON: A WORD ON THE GIL

To keep memory coherent, Python only allows a single thread to run in the interpreter's memory space at once. This is enforced by the Global Interpreter Lock, or GIL.

The GIL isn't all bad. It:

- Is mostly sidestepped for I/O (files and sockets)
- Makes writing modules in C much easier
- Makes maintaining the interpreter much easier
- Makes for any easy topic of conversation
- Encourages the development of other paradigms for parallelism
- Is almost **entirely irrelevant in the HPC space** as it neither impacts MPI or threading within compiled modules

For the gory details, see David Beazley's talk on the GIL: <https://www.youtube.com/watch?v=fwzPF2JLoeU>

NUMPY AND SCIPY

NumPy should almost always be your first stop for performance improvement. It provides:

- N-dimensional homogeneous arrays (ndarray)
- Universal functions (ufunc)
- built-in linear algebra, FFT, PRNGs
- Tools for integrating with C/C++/Fortran
- Heavy lifting done by optimized C/Fortran libraries such as Intel's MKL or IBM's ESSL

SciPy extends NumPy with common scientific computing tools

- optimization
- additional linear algebra
- integration
- interpolation
- FFT
- signal and image processing
- ODE solvers

Problems arise when NumPy isn't well built...



NUMPY AND SCIPY

Optimized and built with MKL via Spack

```
[wscullin@thetalogin6 ~]$ python
Python 2.7.13 (default, May 2 2017, 20:30:06)
[GCC Intel(R) C++ gcc 4.9.4 mode] on linux2
Type "help", "copyright", "credits" or "license" for more information.
readline: /etc/inputrc: line 19: term: unknown variable name
>>> import numpy as np
>>> np.__config__.show()
lapack_opt_info:
  libraries = ['mkl_rt', 'pthread']
  library_dirs = ['/projects/datascience/soft/builds/spack/packages/opt/linux/mkl/lib/intel64']
  define_macros = [(('SCIPY_MKL_H', None), ('HAVE_CBLAS', None))]
  include_dirs = ['/projects/datascience/soft/builds/spack/packages/opt/linux/mkl', '/projects/
datascience/soft/builds/spack/packages/opt/linux/mkl/include', '/projects/datascience/soft/
builds/spack/packages/opt/linux/mkl/lib']
blas_opt_info:
  libraries = ['mkl_rt', 'pthread']
  library_dirs = ['/projects/datascience/soft/builds/spack/packages/opt/linux/mkl/lib/intel64']
  define_macros = [(('SCIPY_MKL_H', None), ('HAVE_CBLAS', None))]
  include_dirs = ['/projects/datascience/soft/builds/spack/packages/opt/linux/mkl', '/projects/
datascience/soft/builds/spack/packages/opt/linux/mkl/include', '/projects/datascience/soft/
builds/spack/packages/opt/linux/mkl/lib']
lapack_mkl_info:
  libraries = ['mkl_rt', 'pthread']
  library_dirs = ['/projects/datascience/soft/builds/spack/packages/opt/linux/mkl/lib/intel64']
  define_macros = [(('SCIPY_MKL_H', None), ('HAVE_CBLAS', None))]
  include_dirs = ['/projects/datascience/soft/builds/spack/packages/opt/linux/mkl', '/projects/
datascience/soft/builds/spack/packages/opt/linux/mkl/include', '/projects/datascience/soft/
builds/spack/packages/opt/linux/mkl/lib']
blas_mkl_info:
  libraries = ['mkl_rt', 'pthread']
  library_dirs = ['/projects/datascience/soft/builds/spack/packages/opt/linux/mkl/lib/intel64']
  define_macros = [(('SCIPY_MKL_H', None), ('HAVE_CBLAS', None))]
  include_dirs = ['/projects/datascience/soft/builds/spack/packages/opt/linux/mkl', '/projects/
datascience/soft/builds/spack/packages/opt/linux/mkl/include', '/projects/datascience/soft/
builds/spack/packages/opt/linux/mkl/lib']
```

The test on a KNL system:

```
>>> import timeit
>>> sum([timeit.timeit('import numpy as np; np.random.random((100,100))*np.random.random((100))') for i in range(100)])/100.0
```

119.68859601020813s

499.9269280433655s

Installed via pip

```
Python 2.7.5 (default, Nov 6 2016, 00:28:07)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-11)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> np.__config__.show()
lapack_info:
  NOT AVAILABLE
lapack_opt_info:
  NOT AVAILABLE
openblas_lapack_info:
  NOT AVAILABLE
blas_info:
  NOT AVAILABLE
atlas_3_10_blas_threads_info:
  NOT AVAILABLE
atlas_threads_info:
  NOT AVAILABLE
blas_src_info:
  NOT AVAILABLE
atlas_3_10_threads_info:
  NOT AVAILABLE
atlas_blas_info:
  NOT AVAILABLE
atlas_3_10_blas_info:
  NOT AVAILABLE
lapack_src_info:
  NOT AVAILABLE
atlas_blas_threads_info:
  NOT AVAILABLE
openblas_info:
  NOT AVAILABLE
blas_mkl_info:
  NOT AVAILABLE
blas_opt_info:
  NOT AVAILABLE
blis_info:
  NOT AVAILABLE
atlas_info:
  NOT AVAILABLE
atlas_3_10_info:
  NOT AVAILABLE
lapack_mkl_info:
  NOT AVAILABLE
>>> █
```

A WORD FROM OUR SPONSORS: CANNED PYTHON

At this point in history, there are few reasons for the average user to manually cobble together a Python stack for themselves on an x86_64 system. **All options are relatively equivalent** with unique advantages and disadvantages to weigh.

We will be making two options available on Theta:

- The Intel Python distribution
- Optimized builds of Python built with LLNL/Spack via modules

You may also wish to consider a commercial distribution:

- Continuum Analytics Anaconda
- Enthought Canopy

Both Intel Python and Continuum Analytics Anaconda build on the Conda package and environment manager. Enthought Canopy relies on virtualenv for environment management.

Think of Conda as being like rpm or deb packages – easy to install binary packages, though managing dependencies becomes potentially problematic.

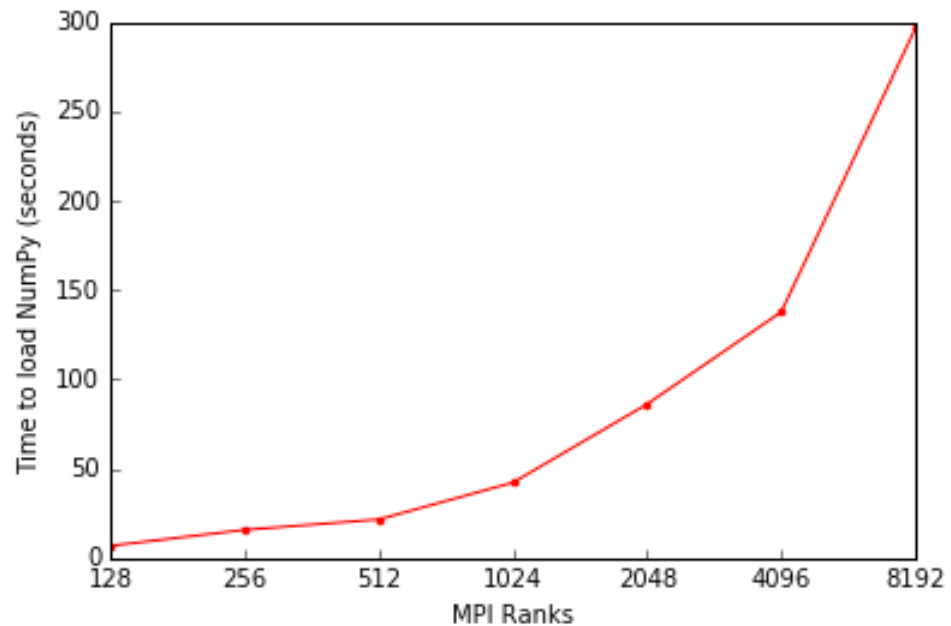
Think of LLNL/Spack+virtualenv as being like BSD or MacPorts – highly customizable, highly transparent, but potentially a lot of time spent compiling.

WHY MPI?

- It is (still) the HPC paradigm for inter-process communications
 - Supported by every HPC center and vendor on the planet
 - APIs are stable, standardized, and portable across platforms and languages
 - We'll still be using it in 10 years...
- It makes full use of HPC interconnects and hardware
 - Abstracts aspects of the network that may be very system specific
 - Dask, Spark, Hadoop, and Protocol Buffers use sockets or files!
 - Vendors generally optimize MPI for their hardware and software
- Well-supported tools for development – even for Python
 - Debuggers now handle mixed language applications
 - Profilers are treating Python as a first-class citizen
 - Many parallel solver packages have well-developed Python interfaces
- Folks have been writing Python MPI bindings since at least 1996
 - David Beazley may have started this...
 - Other contenders: Pypar (Ole Nielsen), pyMPI (Patrick Miller, et al), Pydusa (Timothy H. Kaiser), and Boost MPI Python (Andreas Klöckner and Doug Gregor)
 - The community has mostly settled on mpi4py by Lisandro Dalcin

A BOTTLENECK AT THE START: LOADING PYTHON

When working in diskless environments or from shared file systems, keep track of how much time is spent in startup and module file loading. Parallel file systems are generally optimized for large, sequential reads and writes. NFS generally serializes metadata transactions. This load time can have substantial impact on total runtimes.



MPI4PY

- Pythonic wrapping of the system's native MPI
- provides almost all MPI-1,2 and common MPI-3 features
- very well maintained
- distributed with major Python distributions
- portable and scalable
 - requires only: NumPy, Cython, and an MPI
 - used to run a python application on 786,432 cores
 - capabilities only limited by the system MPI
- <http://mpi4py.readthedocs.io/en/stable/>

HOW MPI4PY WORKS...

- mpi4py jobs are launched like other MPI binaries:
 - `mpiexec -np ${RANKS} python ${PATH_TO_SCRIPT}`
- an independent Python interpreter launches per rank
 - no automatic shared memory, files, or state
 - crashing an interpreter does crash the MPI program
 - it is possible to embed an interpreter in a C/C++ program and launch an interpreter that way
- if you crash or have trouble with simple codes
 - CPython is a C binary and mpi4py is a binding
 - you will likely get core files and mangled stack traces
 - use `ld` or `otool` to check which MPI mpi4py is linked against
 - ensure Python, mpi4py, and your code are available on all nodes and libraries and paths are correct
 - try running with a single rank
 - rebuild with debugging symbols

MPI4PY STARTUP AND SHUTDOWN

- Importing and MPI initialization
 - importing `mpi4py` allows you to set runtime configuration options (e.g. automatic initialization, `thread_level`) via `mpi4py.rc()`
 - by default importing the MPI submodule calls `MPI_Init()`
 - calling `Init()` or `Init_thread()` more than once violates the MPI standard
 - This will lead to a Python exception or an abort in C/C++
 - use `Is_initialized()` to test for initialization
- `MPI_Finalize()` will automatically run at interpreter exit
 - there is generally no need to ever call `Finalize()`
 - use `Is_finalized()` to test for finalization if uncertain
 - calling `Finalize()` more than once exits the interpreter with an error and may crash C/C++/Fortran modules

MPI4PY AND PROGRAM STRUCTURE

Any code, even if after `MPI.Init()`, unless reserved to a given rank will run on all ranks:

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
mpisize = comm.Get_size()

if rank%2 == 0:
    print("Hello from an even rank: %d" %(rank))

comm.Barrier()

print("Goodbye from rank %d" %(rank))
```

MPI4PY AND DATATYPES

- Python objects, unless they conform to a C data type, are pickled
 - pickling and unpickling have significant compute overhead
 - overhead impacts both senders and receivers
 - pickling may also increase the memory size of an object
 - use the lowercase methods, eg: `recv()`, `send()`
- Picklable Python objects include:
 - `None`, `True`, and `False`
 - integers, long integers, floating point numbers, complex numbers
 - normal and Unicode strings
 - tuples, lists, sets, and dictionaries containing only picklable objects
 - functions defined at the top level of a module
 - built-in functions and classes defined at the top level of a module
 - instances of such classes whose `__dict__()` or the result of calling `__getstate__()` is picklable

MPI4PY AND DATATYPES

- Buffers, MPI datatypes, and NumPy objects aren't pickled
 - transmitted near the speed of C/C++
 - NumPy datatypes are autoconverted to MPI datatypes
 - buffers may need to be described as a 2/3-list/tuple
 - [data, MPI.DOUBLE] for a single double
 - [data,count,MPI.INT] for an array of integers
 - custom MPI datatypes are still possible
 - use the capitalized methods, eg: Recv(), Send()
- When in doubt, ask if what is being processed can be represented as memory buffer or only as PyObject

MPI4PY: COLLECTIVES AND OPERATIONS

- Collectives operating on Python objects are naive
- For the most part collective reduction operations on Python objects are serial
- Casing convention applies to methods:
 - lowercased methods will work for general Python objects (albeit slowly)
 - uppercase methods will work for NumPy/MPI data types at near C speed

MPI4PY: PARALLEL I/O

- All 30-something MPI-2 methods are supported
- conventional Python I/O is not MPI safe!
 - safe to read files, though there might be locking issues
 - write a separate file per rank if you must use Python I/O
- h5py 2.2.0 and later support parallel I/O
 - hdf5 must be built with parallel support
 - make sure your hdf5 matches your MPI
 - h5pcc must be present
 - check things with: `h5pcc -showconfig`
 - hdf5 and h5py from Anaconda are serial!
 - anything which modifies the structure or metadata of a file must be done collectively
 - Generally as simple as:

```
f = h5py.File('parallel_test.hdf5', 'w',  
              driver='mpio', comm=MPI.COMM_WORLD)
```


ENUMERATED ADMONISHMENTS

1. Benchmark as you develop
2. Profile
3. Ask if you can do an operation with NumPy or SciPy
4. Never mix forking and threading – ie: Python multiprocessing
5. Check the build configurations of your important Python modules
6. Beware of thread affinity:
`aprun -n ... -N ... -e KMP_AFFINITY=none -d ... -j ...`
7. Watch your data types
8. Avoid Python threading
9. Watch startup times carefully
10. Google – someone else has likely already implemented the solution you seek
11. Python distutils is always the wrong answer

Script	CPython		Pypy	
	Serial / 1 Rank	8 Ranks	Serial / 1 Rank	8 Ranks
builtins_mpi_pi	3.677074	1.065756	0.313690	0.127450
builtins_pyobj_mpi_pi	4.016020	1.092005	0.304663	0.110477
numba_mpi_pi	0.416354	0.424889	n/a	n/a
numpy_mpi_pi	0.666984	0.154728	1.962236	0.556384
threads	7.450568	n/a	0.344480	n/a