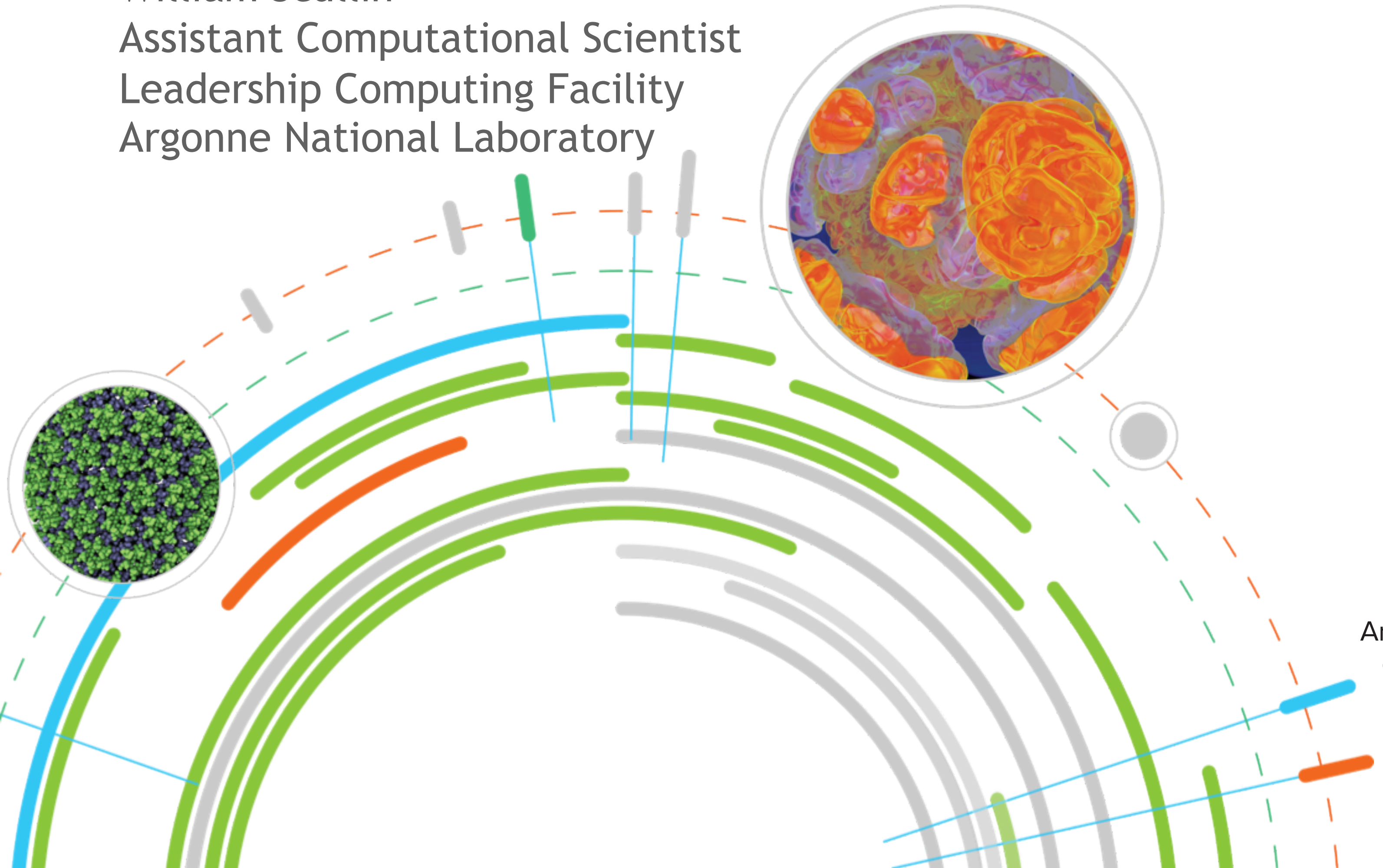
Advanced Scaling with Python

Using MPI

William Scullin
Assistant Computational Scientist
Leadership Computing Facility
Argonne National Laboratory



Argonne Leadership Computing Facility

Argonne NATIONAL LABORATORY

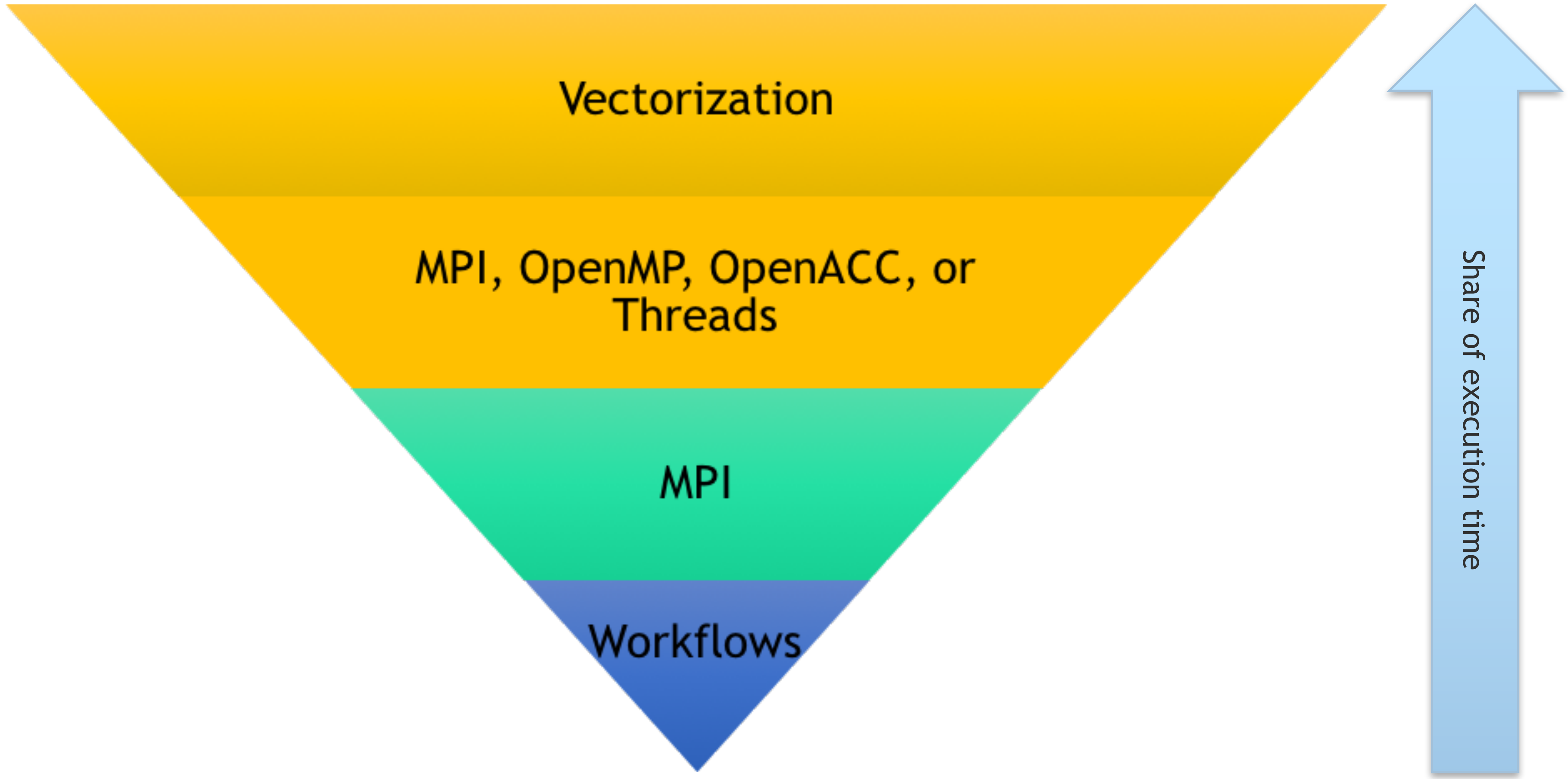
Why MPI?

- It is 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

Why not MPI?

- Generally unsupported outside HPC contexts
  - Packages provided with a distribution may be highly un-tuned
  - Commercial cloud services generally don't have fast interconnects
- Mixing programming paradigms can be messy
  - MPI applications are generally synchronous – you only compute as fast as the slowest process
  - Generally projects use MPI+X where X is node-local
  - Mixing threading paradigms is generally a recipe for disaster
- There can be a steep learning curve
  - Simple to learn, but difficult to master
  - APIs aren't generally taught in CS programs
  - Best tools for debugging and profiling are generally commercial
  - Performance gains aren't automatic

Where does MPI fit in?



Python and MPI

- Python was originally developed as a system scripting language for the Amoeba distributed operating system
- 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
  - You can mix bindings, libraries, and languages – just watch the MPI you link and your data types
- Argonne has required vendor support as part of machine acquisitions since at least 2003.
- Python 3 is the future – and the future is here
  - All major libraries now work under Python 3.5
  - Python 3's loader and internals are more I/O intensive which presents challenges for scaling

Parallelism and Python: A Word on the GIL

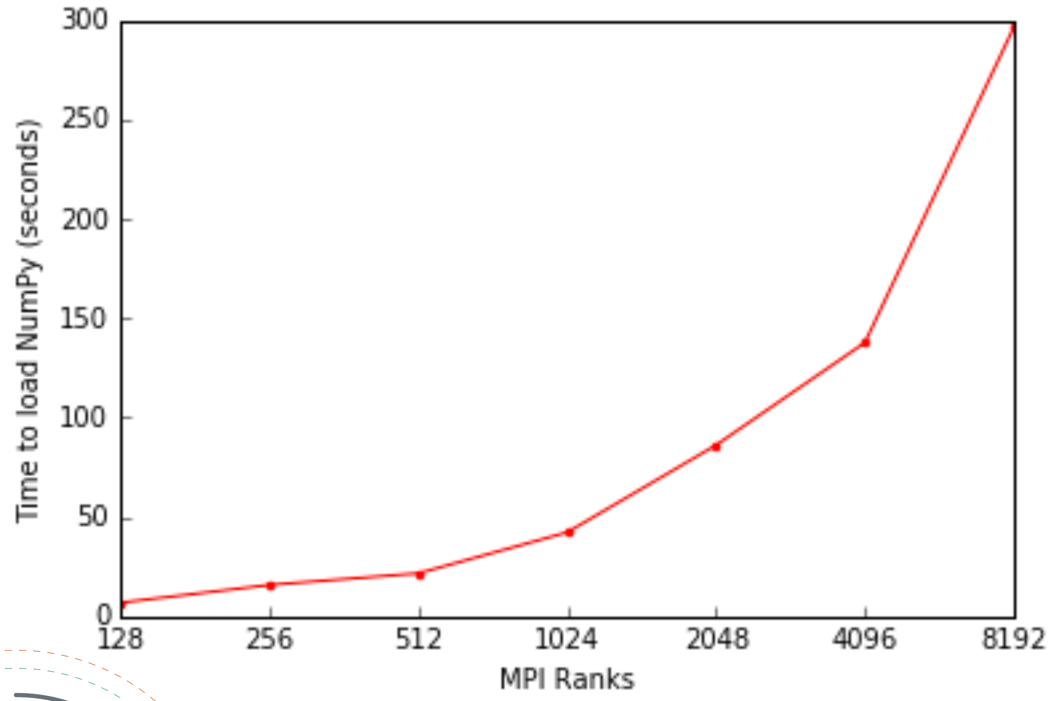
To keep memory coherent, Python only allows a single thread to run in the interpreter's memory space at once. This is enforced by the Global Interpreter Lock, or GIL.

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 threads embedded in compiled modules

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

Parallelism and Python: 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: communicators

- The two default communicators exist at startup:
  - `COMM_WORLD`
  - `COMM_SELF`
- For safety, duplicate communicators before use in or with libraries
- Only break from the standard are methods:
  `Is_inter()` and `Is_intra()`

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)
```

Implementation Example: Basics

- Meant to show the result of various collectives on various datatypes
- Meant to be run on no fewer than 8 ranks due to the scatter of the `dict`

Implementation Example: Monte Carlo Calculation of Pi

Algorithim:

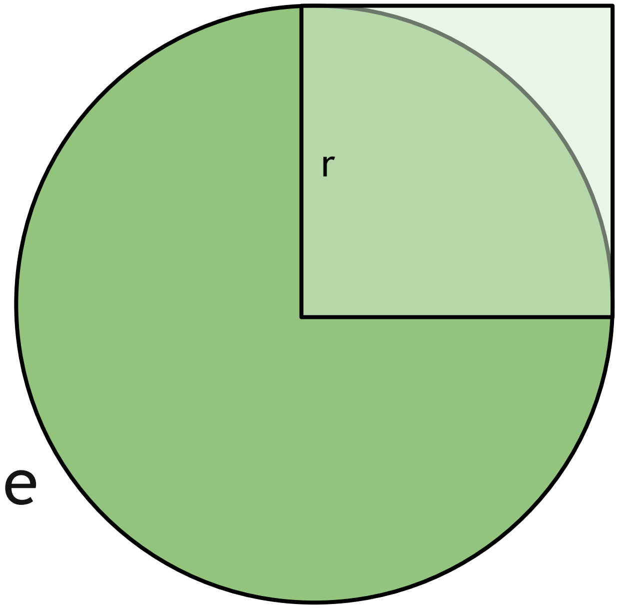
- Generate random points inside a square with sides length (r)
- Identify fraction (f) that fall inside a circle with radius equal to box width (r)
- Use area formulas for quarter of a circle and square to determine Pi:

$$A_{quartercircle} = \frac{\pi r^2}{4}$$

$$A_{square} = r^2$$

$$\frac{A_{quartercircle}}{A_{square}} = f = \frac{\pi}{4}$$

$$\pi = 4f$$



- The accuracy should be proportional to the the number of points tested

Questions?

Acknowledgments

- This work is supported in part by the resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.
- Extended thanks go to:
  - Continuum Analytics, Inc.
  - CSC
  - Enthought Inc.
  - Intel
  - Northwestern University
  - NumFocus
  - Lisandro Dalcin for his work on mpi4py and tolerating questions
  - the members of the Chicago Python User's Group (ChiPy) for allowing me to ramble on about science and HPC