Python for High Performance Computing

Exascale Computing Project 2nd Annual Meeting

William Scullin (ALCF), Matt Belhorn (OLCF),
and Rollin Thomas (NERSC)

Knoxville, Tennessee
February 5, 2018

Agenda

- Introductions, Agenda, and Logistics
- Why this tutorial and why Python?
- Using Python at NERSC, ALCF, and OLCF
- Hands on 1: Logging into NERSC's Cori and setting up an environment
- Performance Basics: Threading Issues ● NumPy ● Extending Python
- Python Parallelism: MPI with mpi4py ● Caveats
- I/O Overview
- Hands-on 2: Using mpi4py
- Profiling Python code
- Debugging Overview
- Hands-on with your code and Q&A
- Other Resources

2

Administrative and Logistical Notes

- **SAFETY: Please note emergency exit locations and watch for bags and cords to avoid tripping hazards**
- There is a shared Google Doc for asking questions and sharing code snippets: http://bit.ly/2018ecppythontutorialshareddoc
  we will add a copy to the repo with slides and materials after the tutorial
- We will be doing hands-on exercises on NERSC Cori. If needed, training accounts are available. (Thanks NERSC!) Please see Matt or William at the first hands on if you didn't get setup before the tutorial started.
- Slides and materials are available at:
  LINK HERE
- The materials are an amalgamation of a decade of community efforts by volunteers. Formatting might be wonky, but the information isn't.
- Please ask questions. We'll be watching the room and the Google Doc.

3

Tutorial Objectives

What we want to do:

- Briefly explain what ALCF, NERSC, and OLCF are doing to welcome and support Python users in HPC.
- Provide guidance and best practices for using Python in a HPC context.
- Reveal common stumbling blocks that impact performance.
- Introduce some great tools that now exist to support developers using Python in HPC settings.

What we assume:

- You know and use Python
- You have some familiarity with the Scientific Python Stack, or
- You know and use HPC resources and are curious about using Python in your own HPC work.

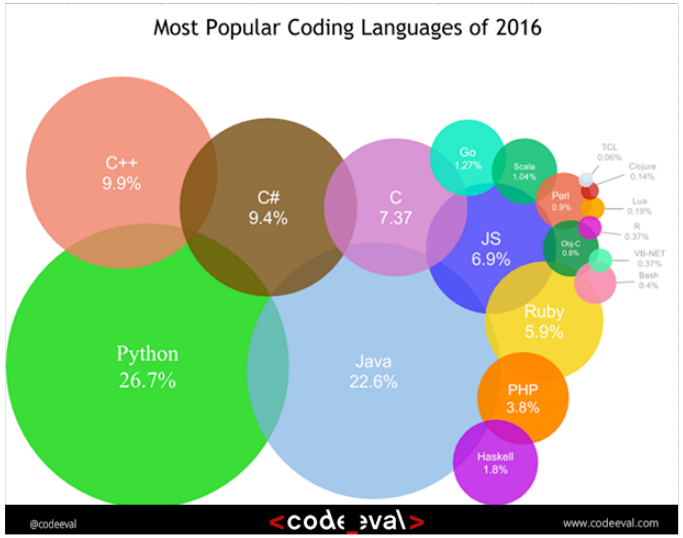
4

Why this tutorial? Why Python?

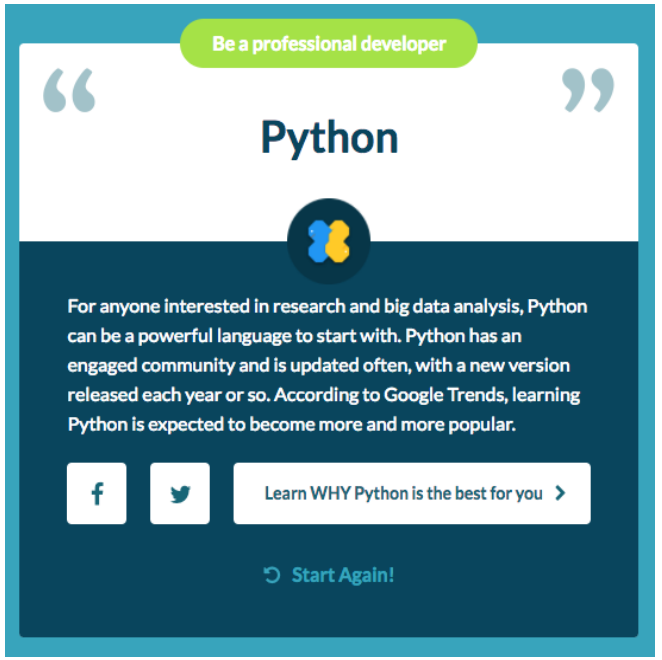
5

Why this tutorial?

- 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



codeval.com



bestprogramminglanguagefor.me

Aug 2016	Aug 2015	Change	Programming Language	Ratings	Change
1	1		Java	19.010%	-0.26%
2	2		C	11.303%	-3.43%
3	3		C++	5.800%	-1.94%
4	4		C#	4.907%	+0.07%
5	5		Python	4.404%	+0.34%
6	7	^	PHP	3.173%	+0.44%
7	9	^	JavaScript	2.705%	+0.54%
8	8		Visual Basic .NET	2.518%	-0.19%
9	10	^	Perl	2.511%	+0.39%
10	12	^	Assembly language	2.364%	+0.60%

http://www.tiobe.com/tiobe-index

Why Not Python?

- Performance is often a secondary concern for developers and distributions
  - Most Python developers aren't in HPC environments
  - Most Python 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
  - Sometimes Python isn't the right tool
- Existing investments in commercial high-productivity languages (e.g. Matlab)

Why is Python Popular?

Makes a great first impression:

- Clean, clear syntax.
- Multi-paradigm, interpreted.
- Duck typing, garbage collection.
- Built-ins mostly map to C/C++ equivalents.
- Excellent built-in documentation.

Keeps up with users' needs:

- Flexible, full-featured data structures.
- Extensive standard libraries.
- Portable.
- Fully open source with a strong development community.
- Commercial support available.
- Reusable open-source packages (PyPI).
- Package management tools.
- Good unit testing frameworks.
- Extensible with C/C++/Fortran for optimizing high-performance kernels.

```
from interface import Model

class BasicModel ( Model ) :

    def __init__( self, gaussian_process, training_data, upda
        self.gaussian_process = gaussian_process
        self.training_data    = training_data

        training_size       = len( self.training_data )
        self._input_diffs   = ( self.training_data.inputs[ No
                self.training_data.inputs[ :, None ] )

        self._gram          = numpy.zeros( ( training_size, t
        self._log_gram_det  = None
        self._inv_gram      = numpy.zeros_like( self._gram )
        self._residuals     = numpy.zeros( training_size )
        self._inv_gram_resp = numpy.zeros( training_size )

        if update :
            self._update()

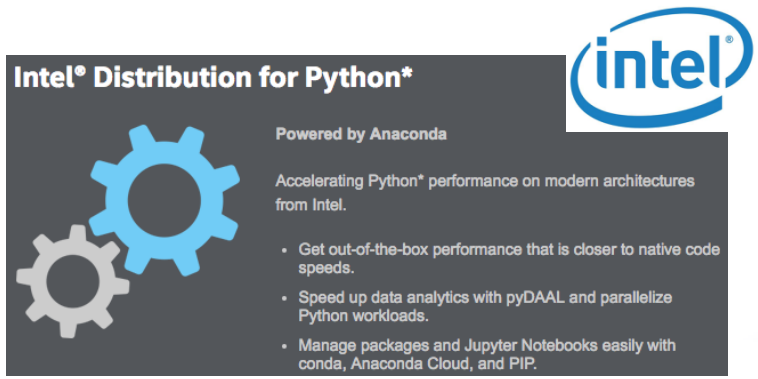
    @property
    def log_p( self ) :
        return -0.5 * ( numpy.dot( self._residuals, self._inv
                self._log_gram_det + len( self.training_data
                numpy.log( 2.0 * numpy.pi ) )

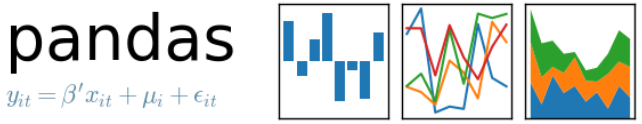
    @property
    def hyperparameters( self ) :
        deque = self.gaussian_process.mean_function.hyperpara
        deque.extend( self.gaussian_process.covariance_functi
        return deque

    @hyperparameters.setter
    def hyperparameters( self, iterable ) :
        deque = collections.deque( iterable )
        self.gaussian_process.mean_function.take_hyperparamet
```


8

The Scientific Python Stack





Primary Uses:

- **Script workflows for both data analysis and simulations**
- **Perform exploratory, interactive data analytics & viz**

9

Choosing between 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

Python at the HPC Center

Observation: ***High productivity*** has driven the growth of Python in the sciences.

...Not ***high performance*** (so much).

But supporting Python is no longer optional at HPC centers like NERSC, ALCF, OLCF.

Maximizing Python performance on these systems can be *(ok, is)* challenging:

- Interpreted, dynamic languages can be difficult to optimize.
- Python's global interpreter lock (GIL) has consequences for parallelism.
- Language design and implementation choices made without considering realities of HPC.

11

PyFR: Gordon Bell & SC16 Best Paper Finalist

Towards Green Aviation with Python at Petascale

Peter Vincent*, Freddie Witherden†, Brian Vermeire‡, Jin Seok Park§ and Arvind Iyer¶
Department of Aeronautics, Imperial College London, London, United Kingdom

- Demonstrated use of Python in a high-end HPC context for simulation of real-world flow problems at up to 13.7 DP-PFLOP/s.
- Detailed how a single Python codebase can target multiple platforms, including heterogeneous systems, using an innovative runtime code-generation paradigm.
- Achieved 58% computational efficiency for an unstructured mesh fluid dynamics simulation.

- Performance portability enabled by Python: CFD from a single code base supporting CPU and GPU architectures, a few x1000 lines of code.
- **Python can be at the highest levels of performance in supercomputing.**

[http://sc16.supercomputing.org/2016/08/23/finalists-compete-prestigious-acm-gordon-bell-prize-high-performance-computing/]
[http://sc16.supercomputing.org/2016/09/21/sc16-announces-best-paper-nominees]

12

Basic Guidelines for Python in HPC

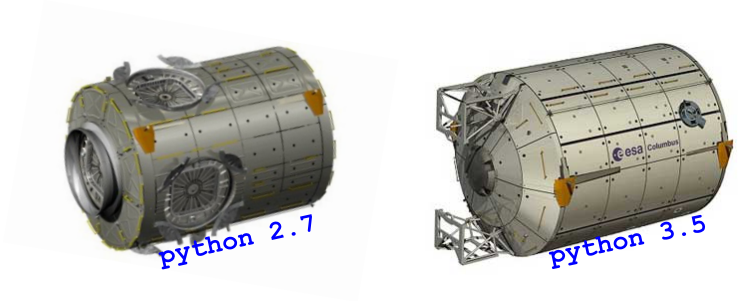
- Identify and exploit parallelism at the core, node, and cluster levels.
- Control your environment and check for correctness.
- Understand and apply numpy array syntax and its broadcasting rules (skipped here):
  https://docs.scipy.org/doc/numpy/reference/arrays.html
  https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html
- Use mpi4py appropriately.
- Use community solvers.
- Measure your codes' performance using profiling tools.
- Develop tests.
- Be part of the community.

13

Practical Matters:
Using Python at NERSC, ALCF, and OLCF

14

Python at NERSC, ALCF, & OLCF



Where do users commonly find Python on HPC systems?

- Environment Modules
  http://modules.sourceforge.net
  "The Environment Modules package provides for the dynamic modification of a user's environment via modulefiles."

```
module avail python
module load python
module swap python/2.7 python/3.5
module help…
```

- Installing their own Python (many options).
- System Python (e.g. `/usr/bin/python`)
  use system Python at your own risk.

15

Python Builds and Distributions

Centers may build, install Python & packages from:

- source
- package managers like Spack*
  https://spack.readthedocs.io/en/latest/
- using distributions like:
  - Anaconda
    https://docs.continuum.io/anaconda/
  - Intel Python
    https://software.intel.com/en-us/distribution-for-python
  - Enthought Canopy
    https://www.enthought.com/product/canopy/
- or all of the above.

Centers also let users set up their own!

- Packages depending on MPI should always be built against system vendor-provided libraries.
- Anaconda distribution comes with Intel MKL built-in.
- Intel distribution heavily leverages Anaconda tools.

[* Spack is also featured here at the ECP 2nd Annual Meeting!]

Customizing and Controlling Your Environment I: Virtualenv

User-controlled isolated python environments

- Almost universally used outside of science
- Site packages root under your control
- Activated venvs preclude other python interpreters
- Semi-conflicts with environment modules
  - Setup environment modules prior to activation

```
$ virtualenv -p python2.7 /path/to/my_env
$ . /path/to/my_env/bin/activate
(my_env)$ pip install --trusted-host \
                pypi.python.org -U pip
(my_env)$ CC=cc MPICC=cc pip install -v \
                  --no-binary :all: mpi4py
(my_env)$ deactivate
```


17

Customizing and Controlling Your Environment I: Virtualenv (cont'd)

Using your packages with an external interpreter:

- Install your own packages in your venv
- Use them with external python within your python scripts
- Mix-and-match with center-provided packages

```
#!/usr/bin/env python2.7
activate_this = '/path/to/env/bin/activate_this.py'
execfile(activate_this, dict(__file__=activate_this))
```

N.B.: Packages installed in the venv will supercede versions installed at the site level.

18

Customizing and Controlling Your Environment II: Conda

Anaconda provides the conda tool for creating a Conda “environment”:

- https://conda.io/docs/index.html
- Create, update, share environments.
- Incompatible with virtualenv, replaces it.
- Many pre-built packages organized in custom “channels.”
- Leverage your center’s Anaconda install to create custom environments with the conda tool.

Conda also works with your own Anaconda/“Miniconda” installation:

```
wget https://repo.continuum.io/miniconda/Miniconda2-latest-Linux-x86_64.sh
/bin/bash Miniconda2-latest-Linux-x86_64.sh -b -p $PREFIX
source $PREFIX/bin/activate
conda install basemap yt…
```

As well as with your own Intel Python installation:

```
conda create -c intel -n idp intelpython2_core python=2
source activate idp
```


19

Python at NERSC

NERSC-built:

```
module load python[/2.7.9]
python_base/2.7.9
numpy/1.9.2
scipy/0.15.1
matplotlib/1.4.3
ipython/3.1.0
```

[default]

Anaconda:

```
module load python/2.7-anaconda
module load python/3.5-anaconda
```

NERSC-built:

None

[default]

Anaconda:

```
module load [python/2.7-anaconda]
module load python/3.5-anaconda
```

Conda env via module (either system)

```
module load python/2.7-anaconda
conda create -n myenv numpy...
source activate myenv
```

[http://www.nersc.gov/users/data-analytics/data-analytics/python/]

20

Python at ALCF

- Every system we run is a cross-compile environment except Cooley
  - pip/distutils/setuptools/anaconda don't play well with cross-compiling
- Blue Gene/Q Python is manually maintained
  - Instructions on use are available in: `/soft/cobalt/examples/python`
  - Modules built on request
- Theta offers Python either as:
  - Intel Python - managed and used via Conda
    - We prefer users to install their own environments
    - Users will need to set up their environment to use the Cray MPICH compatibility ABI and strictly build with the Intel MPI wrappers: http://docs.cray.com/books/S-2544-704/S-2544-704.pdf
  - ALCF Python managed via Spack and loadable via modules:
    `module load alcfpython/2.7.13-20170513`
    - A module that loads modules for NumPy, SciPy, MKL, h5py, mpi4py...
    - We build and rebuild alcfpython via Spack to emphasize performance and Cray compatibility
    - Use of virtualenv is recommended - do not mix conda and virtualenv!!!
    - We'll build any package with a Spack spec on request

21

Python at OLCF

Provided interpreters:

```
module load python[/2.7.9]
             python/3.5.1
```

Major Provided Packages:

```
python_numpy/1.9.2
python_scipy/0.15.1
python_matplotlib/1.2.1
python_ipython/3.0.0
python_mpi4py/1.3.1
python_h5py/2.6.0
python_netcdf4/1.1.7
```

Anaconda:

- Prefer to build your own
- Generally interferes with Tcl Environment Modules

Custom package install paths:

- Prefer NFS project space `/ccs/proj/${PROJECTID}`
- Take care with user site-packages, `${HOME}`
- Avoid `/lustre/atlas`

Further site-specific information on the OLCF Website

[https://www.olcf.ornl.gov/training-event/2016-olcf-user-conference-call-olcf-python-best-practices]

22

Hands-on Exercise 1: Setting up an environment on Cori

Instructions:

1. Ensure that you have an account.
2. Log in and verify you are in group ntrain:

```
ssh train${N}@cori.nersc.gov
train15@cori10:~> groups
train15 ntrain training
train15@cori10:~> salloc -N 1 -q regular -t 20 -C haswell \
--reservation=ecp_python -A ntrain
salloc: Granted job allocation 10011248
train15@nid00681:~> module load python/3.6-anaconda-4.4
train15@nid00681:~> which python
/usr/common/software/python/3.6-anaconda-4.4/bin/python
```

3.Create an environment

```
conda create -n myenv numpy python=3
[installation messages]
source activate myenv
```

4. Install mpi4py

```
env MPICC=$(which cc) pip install mpi4py
```

5. Check out the examples

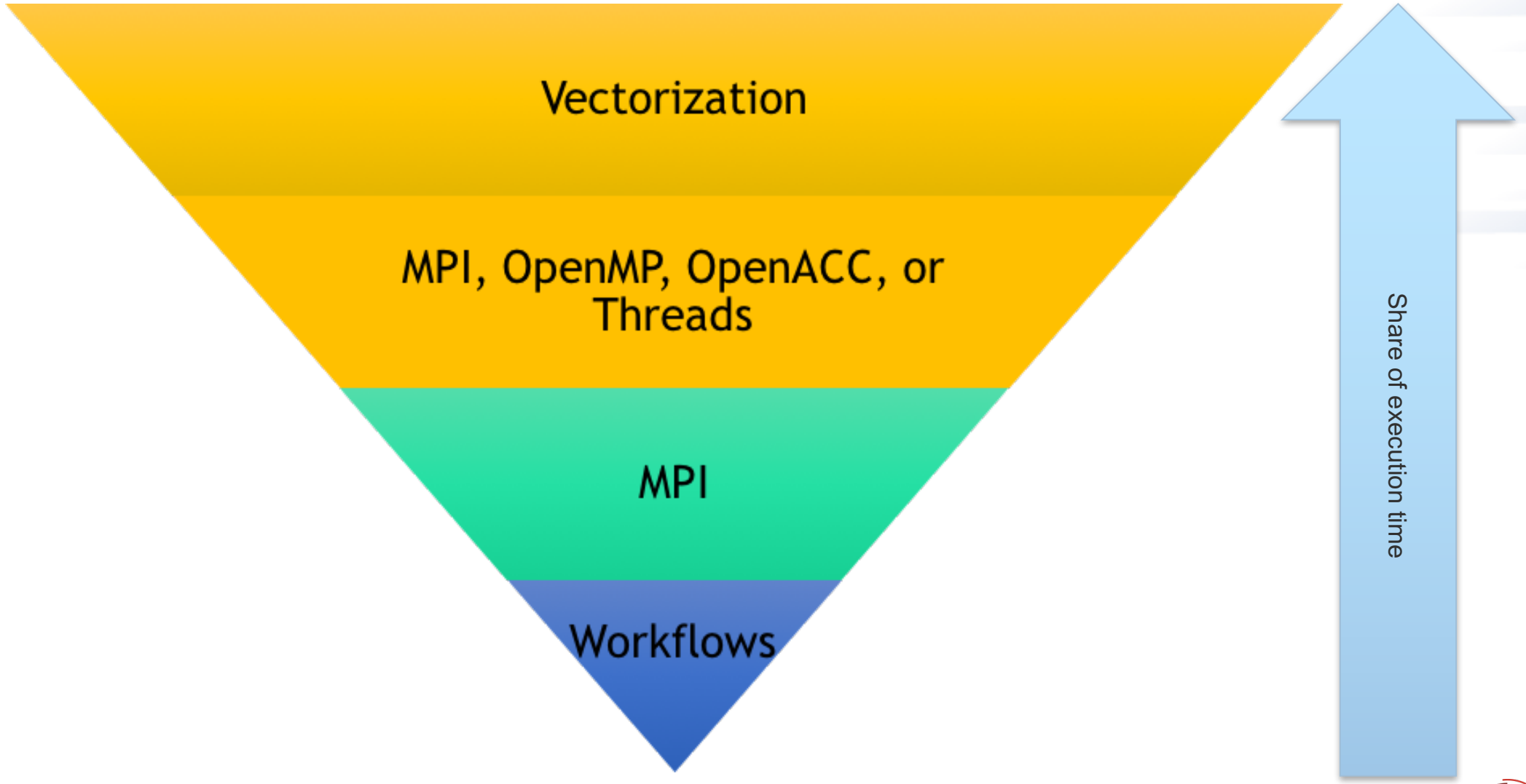
```
git clone https://github.com/wscullin/ecp_python_tutorial.git
```


23

Performance Basics:
Python Internals and Threading
NumPy
Extending Python

24

Structuring a HPC Python code



25

How does CPython work?

Let's look a a function to calculate the area of a circle as Python sees it.

```
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? (Part 2)

Let's use that to calculate an array of circle areas:

```
>>> 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
>>>
```

Can we improve things in pure Python?

List comprehensions seem to simplify things…

```
>>> 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
>>>
```

Takeaways on CPython

- CPython is a **Read–Eval–Print Loop** (**REPL**) environment.
- There is no look-ahead to enable optimizations.
- There is no automatic parallelism.
- Everything is evaluated piece-wise and sequentially.
- CPython was written for safety and ease of maintenance, not performance:

  Russell Power and Alex Rubinsteyn wrote in their paper "How fast can we make interpreted Python?": "In the general absence of type information, almost every instruction must be treated as INVOKE_ARBITRARY_METHOD."

- While you can improve pure Python performance through language features running in CPython, it won't deliver the efficiency of compiled code.

29

Parallelism & 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
- 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

30

NumPy and SciPy

NumPy should almost always be your first stop for performance improvement. Compiled from C and FORTRAN 77, it provides:

- Numerical data types that ease working with C/C++/Fortran
- 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, OpenBLAS, 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

How NumPy is built matters:

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']
```

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
>>>
```

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
```

Optimized build: 119.68859601020813s pip install: 499.9269280433655s

Checking your NumPy Configuration:

Check your configuration for the use of optimized libraries:

```
>>> import numpy as np
>>> np.__config__.show()
```

NumPy's distutils can give insight into compilers and options used:

```
>>> import numpy
>>> import numpy.distutils
>>> np_config_vars = numpy.distutils.unixccompiler.sysconfig.get_config_vars()
>>> # np_config_vars is a dict with configuration values
>>> import pprint
>>> # pprint is a pretty printer and not required, just recommended
>>> pprint.pprint(np_config_vars)
{'AC_APPLE_UNIVERSAL_BUILD': 0,
 'AIX_GENUINE_CPLUSPLUS': 0,
 'AR': 'ar',
 'ARCH': 'x86_64',
 'ARFLAGS': 'rc',
...
```

NumPy Data Types

NumPy covers all the same numeric data types available in C/C++ and Fortran as variants of int, float, and complex:

- all available signed and unsigned as applicable
- available in standard lengths
- floats are double precision by default
- generally available with names similar to C or Fortran
  - ie: long double is `longdouble`
- generally compatible with Python data types
- follow endianness of the platform – conversion routines are offered
- `longdouble` follows the compiler / platform's definition of long double

NumPy also offers the ability to create structured datatypes.

If it can be done in C/C++/Fortran, it can be done in NumPy.

Creating NumPy Arrays

```
# Initialize with Python lists: array with 2 rows, 4 cols
>>> import numpy as np
>>> np.array([[1,2,3,4],[8,7,6,5]])
array([[1, 2, 3, 4],
       [8, 7, 6, 5]])

# Make an array of evenly spaced numbers over an interval
>>> np.linspace(1,100,10)
array([   1.,   12.,   23.,   34.,   45.,   56.,   67.,   78.,
    89.,  100.])

# Create an array and pre-populate with zeros
>>> np.zeros((2,5))
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
```

Slicing NumPy Arrays (Part 1)

```
>>> a = np.array([[1,2,3,4],[9,8,7,6],[1,6,5,4]])
>>> arow = a[0,:] # get slice referencing row zero
>>> arow
array([1, 2, 3, 4])

>>> cols = a[:,[0,2]] # get slice referencing columns 0 and 2
>>> cols
array([[1, 3],
       [9, 7],
       [1, 5]])
```

Slicing NumPy Arrays (Part 2)

```
# NOTE: arow & cols are NOT copies, they point to the original data
>>> arow
array([1, 2, 3, 4])
>>> arow[:] = 0
>>> arow
array([0, 0, 0, 0])

>>> a
array([[0, 0, 0, 0],
       [9, 8, 7, 6],
       [1, 6, 5, 4]])

# Explicitly copy data
>>> copyrow = arow.copy()
```

Broadcasting with universal functions (ufuncs)

Applies operations to many elements with a single call – with compiled code

```
>>> a = np.array(([1,2,3,4],[8,7,6,5]))
>>> a
  array([[1, 2, 3, 4],
         [8, 7, 6, 5]])
```

Rule 1: Dimensions of one may be prepended to either array to match the array with the greatest number of dimensions

```
>>> a + 1 # add 1 to each element in array
array([[2, 3, 4, 5],
       [9, 8, 7, 6]])
```

Rule 2: Arrays may be repeated along dimensions of length 1 to match the size of a larger array

```
>>> a + np.array(([1],[10])) # add 1 to 1st row, 10 to 2nd row
array([[ 2, 3, 4, 5],
       [18, 17, 16, 15]])

>>> a**([2],[3]) # raise 1st row to power 2, 2nd to 3
array([[ 1,   4,   9, 16],
       [512, 343, 216, 125]])
```

When NumPy isn’t enough

- Building blocks like NumPy and SciPy are already built with great vectorizations and thread support via the libraries they link with: BLAS/LAPACK, MKL, FFTW
- Don’t re-implement solvers in pure Python or even NumPy - many of your favorite libraries and packages already have Python bindings:
  - PyTrilinos
  - petsc4py
  - Elemental
  - SLEPc
- Where bindings for a library aren’t available, it’s often easy to generate them

39

Developing Your Own Bindings and Compiled Modules

While not an exhaustive, common options for using pre-compiled, vectorized, threaded, GIL-free code for speed from Python include:

- Cython – create C code from Python or a Python-like language
- F2PY – wrap Fortran code
- PyBind11 – “seamless operability between C++11 and Python”
- swig – generate bindings for just about anything
- Boost.Python – “seamless operability between C++ and Python”
- ctypes – built-in Python FFI for interfacing C **an option of last resort**
- Writing bindings in C/C++ http://dan.iel.fm/posts/python-c-extensions/

40

Developing Your Own Modules: Cython

- Cython is a meant to make writing C extensions easy
- Naive usage can offer x12 speedups
- Builds on Python syntax
- Translates .pyx files to C which compiles
- Provides interfaces for using functionality from OpenMP, CPython, libc, libc++, NumPy, and more
- Works best when you can statically type variables
- Lets you turn off the GIL
- Provides annotations to guide development

41

Developing Your Own Modules: Cython

Using `cython -a ${sourcefile}.{pyx,py}`, we can get guidance on where a module built with Cython would have to interact with CPython and lose performance:

Generated by Cython 0.25.2

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

Raw output: calcpipy.c

```
+01: import random
 02:
+03: def calcpi_py(samples):
 04:     """serially calculate Pi using only standard library functions"""
+05:     inside = 0
+06:     random.seed(0)
+07:     for i in range(int(samples)):
+08:         x = random.random()
+09:         y = random.random()
+10:         if (x*x)+(y*y) < 1:
+11:             inside += 1
+12:     return (4.0 * inside)/samples
```

Generated by Cython 0.25.2

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

Raw output: calcpi.c

```
+01: cdef extern from "stdlib.h":
 02:     cpdef long random() nogil
 03:     cpdef void srandom(unsigned int) nogil
 04:     cpdef const long RAND_MAX
 05:
+06: cdef double randdbl() nogil:
 07:       cdef double r
+08:       r = random()
+09:       r = r/RAND_MAX
+10:       return r
 11:
+12: cpdef double calcpi(const int samples):
 13:     """serially calculate Pi using Cython library functions"""
 14:     cdef int inside, i
 15:     cdef double x, y
 16:
+17:     inside = 0
 18:
+19:     srandom(0)
+20:     for i in range(samples):
+21:         x = randdbl()
+22:         y = randdbl()
+23:         if (x*x)+(y*y) < 1:
+24:             inside += 1
+25:     return (4.0 * inside)/samples
 26:
```


42

Developing Your Own Modules: f2py

f2py comes with NumPy and can be used to rapidly generate wrappers for Fortran code

```
$cat calcpi.f90

    subroutine calcpi(samples, pi)
      REAL,INTENT(OUT) :: pi
      INTEGER, INTENT(IN) :: samples
      REAL :: x, y
      INTEGER :: i, inside

      inside = 0

      do i = 1, samples

        call random_number(x)
        call random_number(y)

        if ( x**2 + y**2 <= 1.0D+00 ) then
          inside = inside + 1
        end if

       end do
       pi = 4.0 * REAL(inside) / REAL(samples)
    end subroutine

$f2py --fcompiler=gfortran -m calcpi_fortran -c calcpi.f90
$...
$python -c "import calcpi_fortran; print calcpi_fortran.calcpi(1000000)"
3.14163589478
```


43

Other Tools for Performance

There are a handful of projects that seek to improve performance of pure Python code. Two noteworthy options are:

- Numba – a Python JIT
  - Sponsored by Continuum (now Anaconda, Inc.)
  - Can target CPUs and GPUs
  - Relies on decorators
- PyPy – an alternative to CPython
  - Not yet 100% compatible with CPython and all modules
  - No code changes required

44

Python Parallelism

45

Overview of Parallel and Distributed Programming Options

threading

- useful for certain concurrency issues, not really usable for parallel computing due to the GIL

subprocess

- relatively low level control for spawning and managing processes, think popen

multiprocessing - multiple Python instances (processes)

- basic multiple process parallelism through forked interpreters
- **N.B. Does not mix well with OpenMP, MPI, or shared memory tools**

MPI

- mpi4py exposes your full local MPI API within Python
- as scalable as your local MPI

GPU (OpenCL & CUDA)

- PyOpenCL and PyCUDA provide low and high level abstraction for highly parallel computations on GPUs

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

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
- ALCF 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

mpi4py: why mpi4py?

- Pythonic wrapping of the system's native MPI
- provides almost all MPI-1,MPI-2 and common MPI-3 features
- very well maintained
- distributed with major Python distributions
- portable and scalable
  - requires only: NumPy, Cython (build only), 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/

50

mpi4py: running

- mpi4py jobs are launched like other MPI binaries:
  - `mpiexec -np ${RANKS} python ${PATH_TO_SCRIPT}`
  - Just running python ${PATH_TO_SCRIPT} should always work for a single-rank case
- an independent Python interpreter launches per rank
  - no automatic shared memory, files, or state
  - crashing an interpreter crashes MPI globally – like any other language
  - it is possible to embed a Python interpreter in a C/C++ program and use MPI from there

51

mpi4py: startup

Importing and MPI initialization:

- importing mpi4py allows you to set runtime configuration options (e.g. automatic initialization, thread_level) via mpi4py.rc()
- importing the MPI submodule calls MPI_Init() by default
  - 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

52

mpi4py: shutdown

- `MPI_Finalize()` will automatically run at interpreter exit
- use `Is_finalized()` to test for finalization when uncertain if a module called `MPI_Finalize()`
- calling `Finalize()` more than once exits the interpreter with an error and may crash C/C++/Fortran modules

53

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 (serialized)
  - 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 if it can only be represented in C as PyObject

mpi4py: communicators

- Three default communicators exist at startup:
  - `COMM_WORLD`
  - `COMM_SELF`
  - `COMM_NULL`
- Methods are bound to communicators e.g.: MPI.`COMM_WORLD.Get_size()`
- Only break from the standard calls are the additional methods: `Is_inter()` and `Is_intra()`
- For safety, duplicate communicators before use in or with libraries and modules you didn't write

57

mpi4py: collectives and operations

- Collectives operating on Python objects are naïve
  - Iterable types generally get cast to a list – this has some interesting side-effects
  - Operators work about the way one would expect with Python types
- Collective reduction operations on Python objects are mostly 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

58

mpi4py: crashing

If you crash or have trouble running simple codes:

- Remember: 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
    - `mpi4py.get_config()` will show you the contents of `mpi.cfg` used at build time and is generally of limited utility
- Ensure Python, mpi4py, and your code are available on all nodes and libraries and paths are correct
- Try running with a single rank
- Rebuild binary modules with debugging symbols
- The default error handler is `MPI.ERRORS_RETURN` which allows the use of Python exception handling, but can allow for silent death in C/C++/Fortan MPI code.
    - Use `MPI.{Comm|Win|File}.Set_errhandler()` to set `MPI.ERRORS_ARE_FATAL` on any communicator, memory window, or file you pass into C/C++/Fortan MPI code.
    - Use `MPI.{Comm|Win|File}.Get_errhandler()` to check the error handler on any communicator, memory window, or file passed from C/C++/Fortan MPI code.

59

Parallel I/O and h5py

- General Python I/O isn't MPI-safe
  - As in any other language, reads are safe though there may be locking issues
  - Likewise, if you must use Python I/O, write a file per MPI rank or thread
- mpi4py provides full support for MPI-IO
  - As a wrapper, the level of support depends on your underlaying MPI

60

Parallel I/O and h5py

- h5py 2.2.0 and later support parallel I/O
- Requires mpi4py and the `mpi` used to compile hdf5, mpi4py, and h5py must be the same.
- Beware pre-packaged h5py and hdf5 – it's frequently serial
- Confirm h5py's MPI support before using:

```
>>> import h5py
>>> h5py.get_config().mpi
True
```

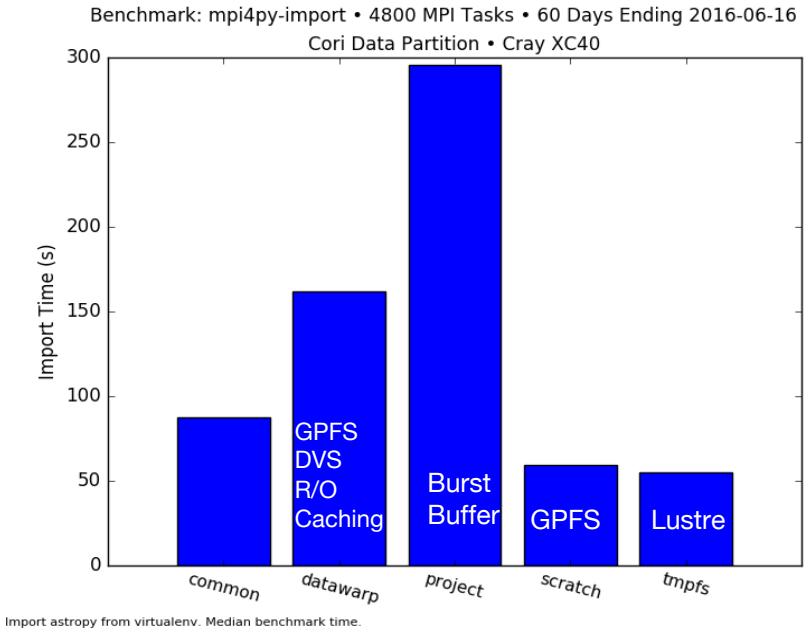
- While setup is sometime trouble, it is as easy to use as:

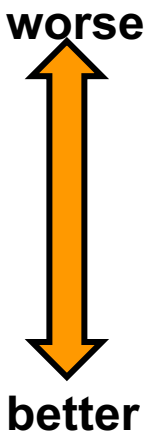
```
f = h5py.File('myfile.hdf5', 'w',
        driver='mpio', comm=MPI.COMM_WORLD)
```

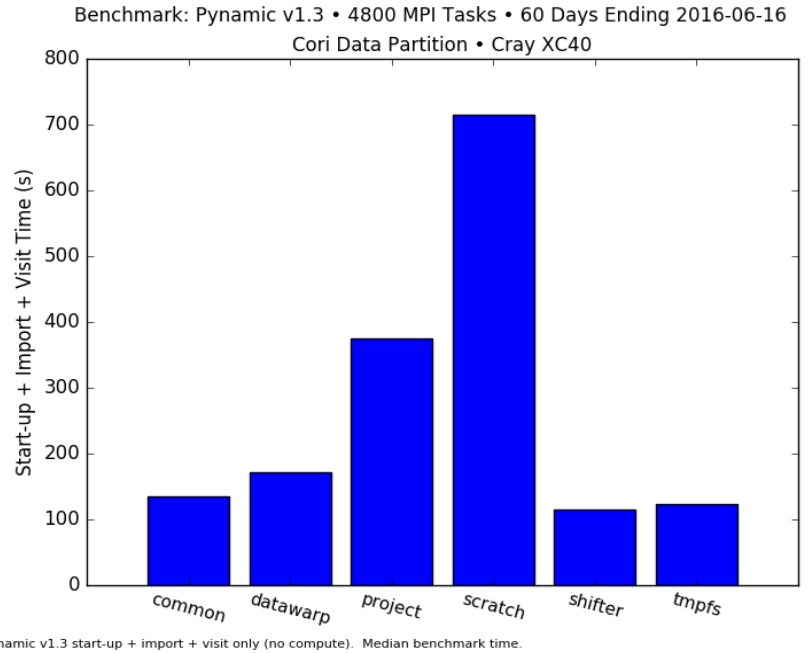
- All changes to file structure or metadata of a file must be performed on all ranks with an open file

61

Issues Affecting Python at Scale







- Python's "import" statement is file metadata intensive (.py, .pyc, .so open/stat calls).
- Becomes more severe as the number of Python processes trying to access files increases.
- Result: Very slow times to just start Python applications at larger concurrency (MPI).
- Storage local to compute nodes, use of containers (**Shifter**) helps fix:
  - Eliminates metadata calls off the compute nodes.
  - In containers, paths to .so libraries can be cached via ldconfig.
- Other approaches:
  - Ship software stack to compute nodes (e.g., python-mpi-bcast).
  - Install software to read-only/cache-enabled file systems.
  - See also Spindle or collfs (Scalable Shared Library Loading).

62

Hands-on Exercise 2: Using mpi4py

Instructions:

1. Grab a node:
   ```
   salloc -N 1 -q regular -t 240 -C haswell -A ntrain --reservation=ecp_python
   ```
2. Activate the environment setup earlier:
   ```
   source activate myenv
   ```
3. Change to the directory basics
   ```
   cd ~/ecp_python_tutorial/basics
   ```
4. Run basic_features.py on 8 ranks:
   ```
   srun -n 8 -c 1 python basic_features.py
   ```basic_features.py


63

# Hands-on Exercise 2: Using mpi4py (part 2)

Instructions:

5. Output should look like:

```
mpirun -n 8 $(which python) ./basic_features.py
###############################################################################
Rank 0 sees local_dict as {'a': 1, 'c': 3, 'b': 2, 'e': 5, 'd': 4, 'g': 'gee whiz', 'f': 6, 'h': ('hi', 'there')}
Rank 0 sees local_list_max as [0, 1, 2, 3]
Rank 0 sees local_list_sum as [0, 1, 2, 3]
Rank 0 sees local_string as This is a string.
Rank 0 sees local_tuple as (0, 0, 0, 0, 0, 0, 0, 0)
Rank 0 sees local_np_array as [0 1 2 3 4 5 6 7 8 9]
###############################################################################
Rank 6 sees local_dict as None
Rank 6 sees local_list_max as [0, 6, 12, 18]
Rank 6 sees local_list_sum as [0, 6, 12, 18]
Rank 6 sees local_string as This should be fun!
Rank 6 sees local_tuple as (6, 6, 6, 6, 6, 6, 6, 6)
Rank 6 sees local_np_array as [0 1 2 3 4 5 6 7 8 9]
###############################################################################

Running collective operations

###############################################################################
Rank 0 sees local_dict as a after scatter using None
Rank 0 sees local_list_max as [0, 7, 14, 21] after allreduce using max
Rank 0 sees local_list_sum as [0, 1, 2, 3, 0, 1, 2, 3, 0, 2, 4, 6, 0, 3, 6, 9, 0, 4, 8, 12, 0, 5, 10, 15, 0, 6, 12, 18, 0, 7, 14, 21] after reduce using sum
Rank 0 sees local_string as This is a string. after bcast using defaults
Rank 0 sees local_tuple as [0, 1, 2, 3, 4, 5, 6, 7] after alltoall using defaults
Rank 0 sees local_np_array as [ 0  8 16 24 32 40 48 56 64 72] after allreduce using sum
###############################################################################
Rank 6 sees local_dict as f after scatter using None
Rank 6 sees local_list_max as [0, 7, 14, 21] after allreduce using max
Rank 6 sees local_list_sum as None after reduce using sum
Rank 6 sees local_string as This is a string. after bcast using defaults
Rank 6 sees local_tuple as [0, 1, 2, 3, 4, 5, 6, 7] after alltoall using defaults
Rank 6 sees local_np_array as [ 0  8 16 24 32 40 48 56 64 72] after allreduce using sum
```


64

Hands-on Exercise 2: Using mpi4py

6. Why wasn't our output in order?
7. How might we scatter a dictionary?

8. Change to the directory pi
```
cd ~/ecp_python_tutorial/pi
```

9. Run builtins_mpi_pi.py on 1, 8, and 16 ranks:
```
srun -n 1 -c 1 python builtins_mpi_pi.py
srun -n 8 -c 1 python builtins_mpi_pi.py
srun -n 16 -c 1 python builtins_mpi_pi.py
```

10. Run threads_pi.py with 1, 8, and 16 threads with the same sample count:
```
./threads_pi.py 12000000 1
./threads_pi.py 12000000 8
./threads_pi.py 12000000 16
```
11. What does this tell us about native Python threads?

65

Basic Profiling: cProfile & SnakeViz

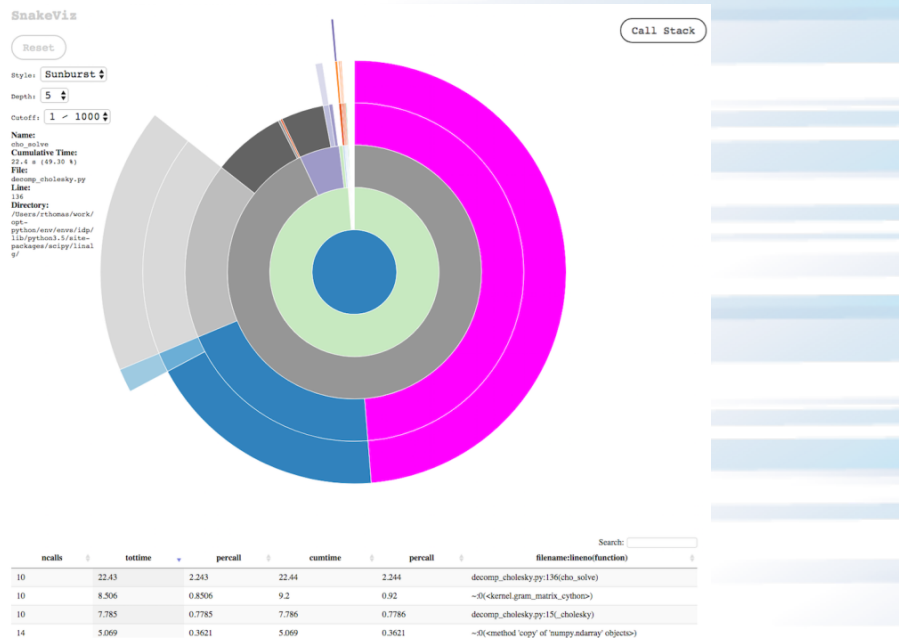
cProfile

Low-overhead profiler, from standard library.

Outputs statistics on what your code is doing:

- Number of function calls,
- Total time spend in functions,
- Time per function call, etc.

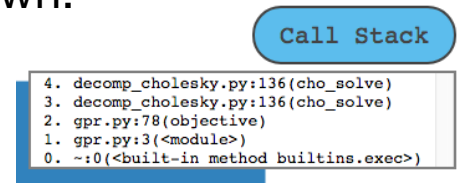
[https://docs.python.org/2/library/debug.html]
[https://docs.python.org/2/library/profile.html#module-cProfile]
[https://docs.python.org/2/library/profile.html#the-stats-class]

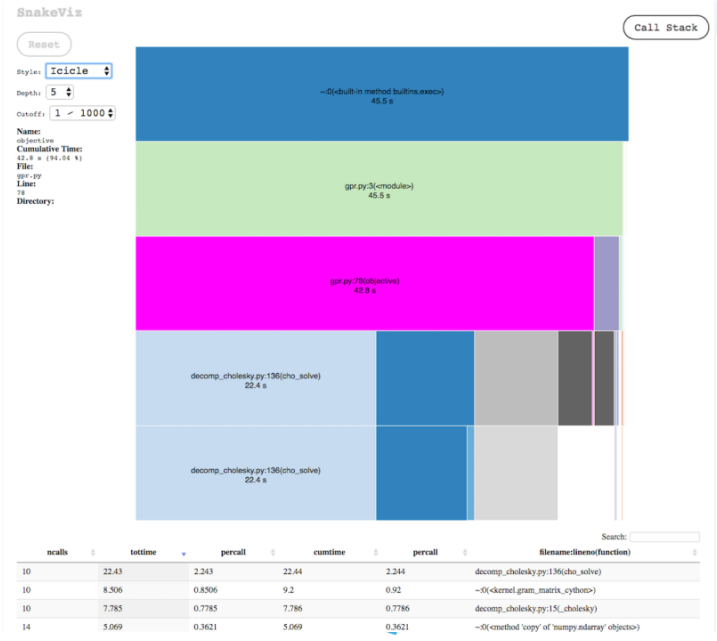


SnakeViz

Lets you visualize cProfile output in a browser:

- Statistics mentioned above.
- Visualize call stack & drill-down.






```
> python -m cProfile -o out.prof my-program.py
...
> snakeviz out.prof
snakeviz web server started on 127.0.0.1:8080...
```

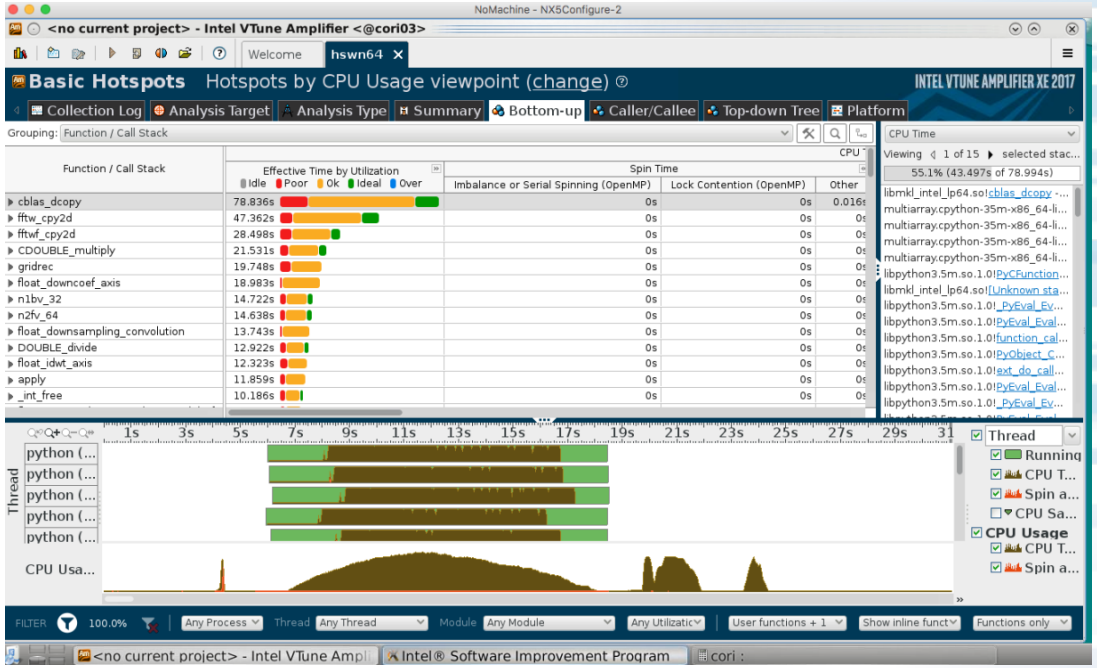
[https://jiffyclub.github.io/snakeviz/]

Intel VTune Works with Python Code

VTune Amplifier

Performance analysis profiler.
GUI and command-line interface.

Thread timelines.
Hotspot analysis.
Memory profiling.
Locks & waits.
Filter/zoom in timeline.



Run GUI (**`amplxe-gui`**) over NX!

[https://software.intel.com/en-us/node/628111]
[https://software.intel.com/en-us/videos/performance-analysis-of-python-applications-with-intel-vtune-amplifier]

Part of Intel Parallel Studio, may be available as a module, e.g. at NERSC:

```
> module load vtune
> salloc … --perf=vtune
…
> srun … amplxe-cl -collect hotspots python my-app.py
```

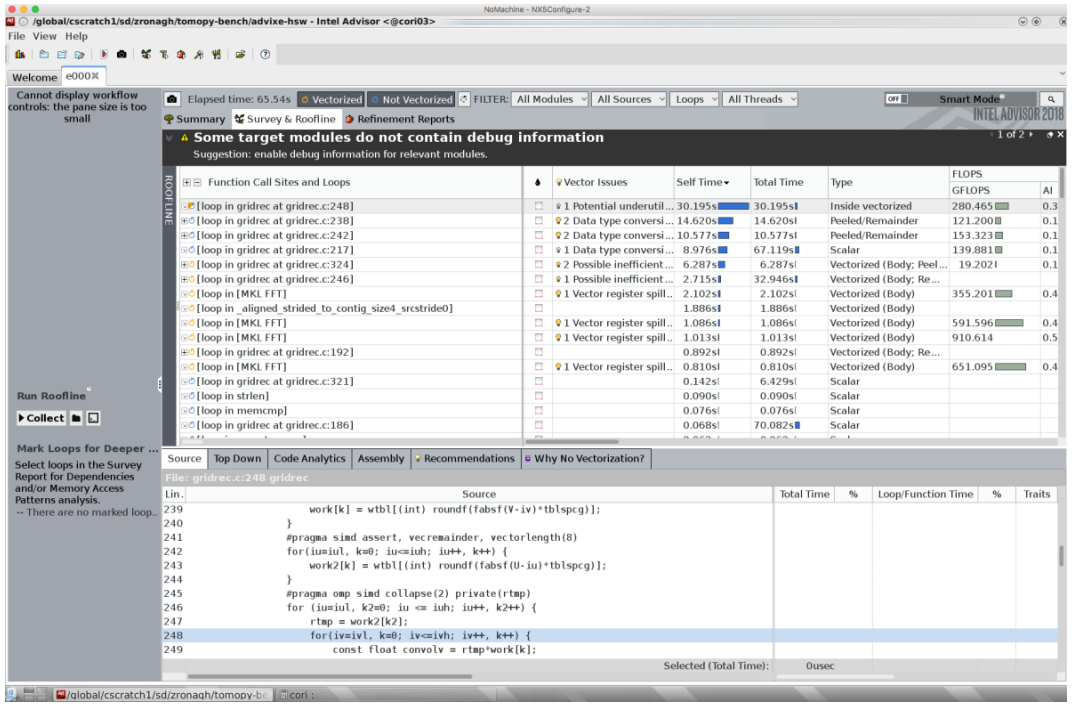
Best practice on KNL:

`-no-auto-finalize`, archive and **`-finalize`** on e.g. Haswell node
[e.g. http://www.nersc.gov/assets/Uploads/04-vtune.pdf]

Intel Tools Screenshots of TomoPy analysis
courtesy Zahra Ronaghi, NERSC

67

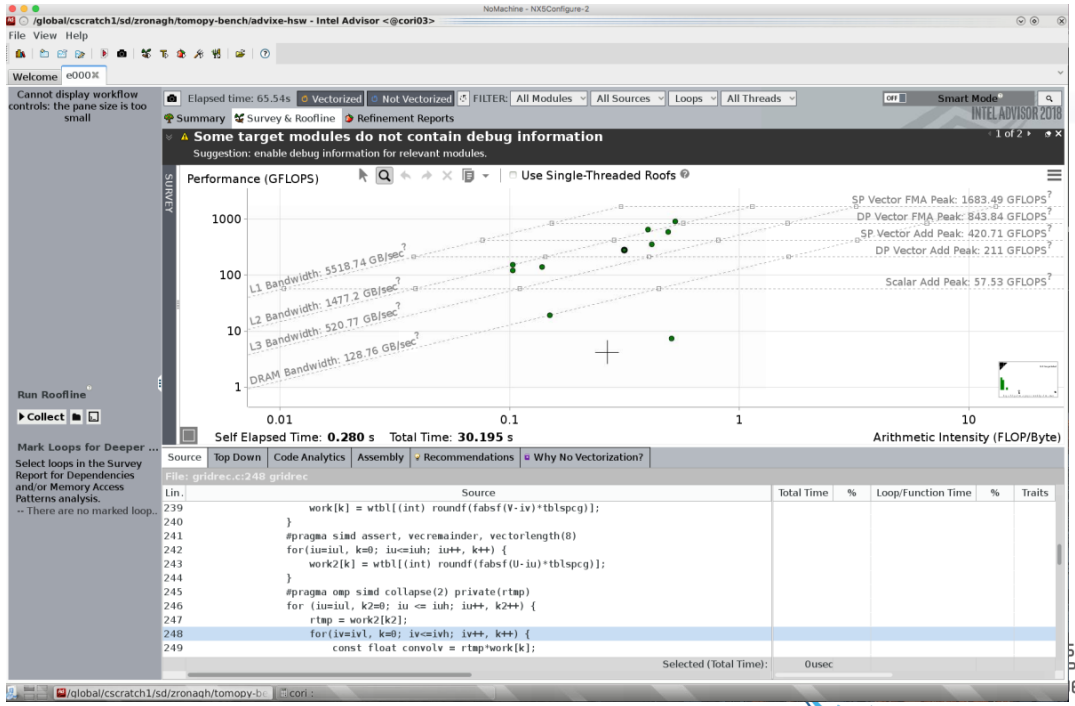
Intel Advisor Works with Python Code



Roofline analysis*:
Performance of code in relation to hardware limits.
Memory bandwidth or compute bound?

[* Roofline: An upcoming IDEAS Webinar topic.]

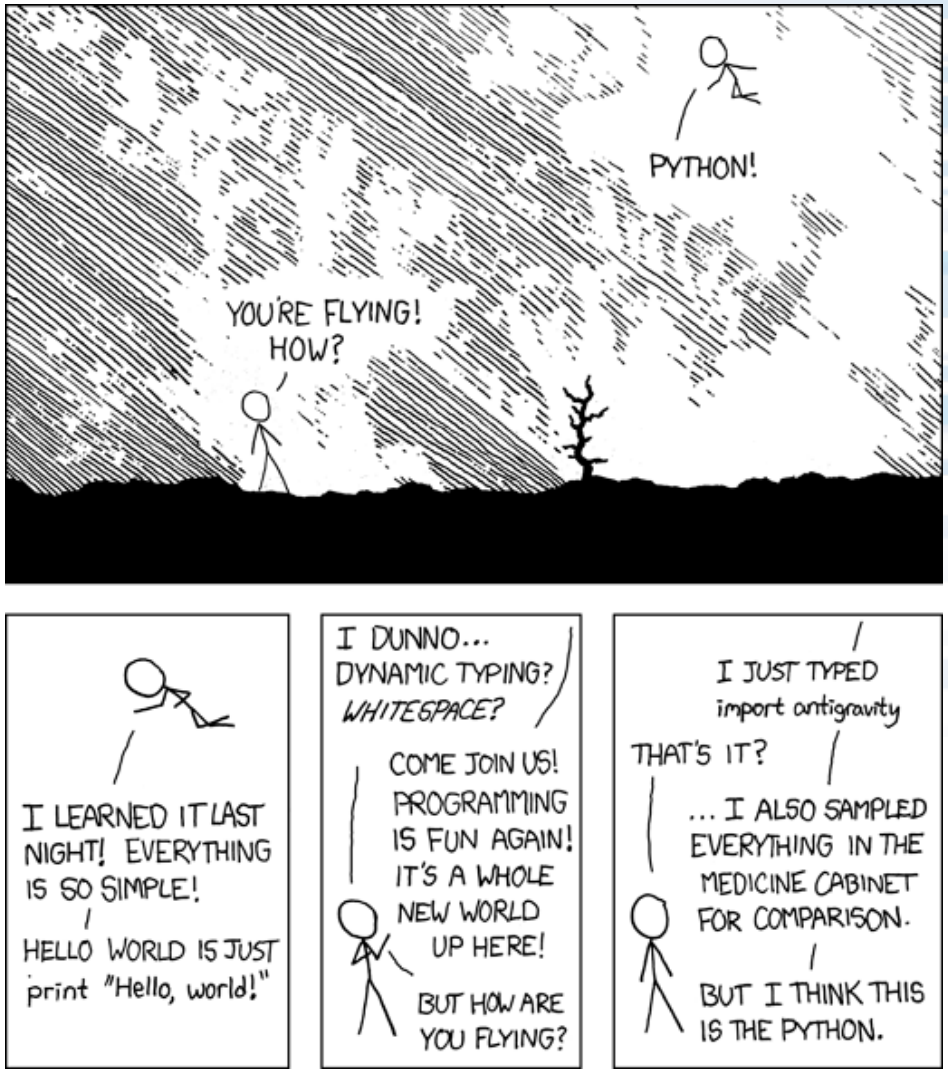
What should I do next? When do I stop?
Suggests optimizations for your C extensions.
Point out vectorization opportunities.
Optimize use of threads.
Works with Python and C/C++/Fortran code.



68

Getting Started with Python Resources

- https://www.python.org/about/gettingstarted/
- https://wiki.python.org/moin/BeginnersGuide
- https://www.codecademy.com/learn/python
- https://www.coursera.org/specializations/python
- https://software-carpentry.org/lessons/
- https://pymotw.com/



https://xkcd.com/353/

69

More Resources

Your NERSC and LCF Python contacts:

- NERSC: Rollin Thomas rcthomas@lbl.gov
- ALCF: William Scullin wscullin@alcf.anl.gov
- OLCF: Matt Belhorn belhornmp@ornl.gov

Documentation:

- NERSC: http://www.nersc.gov/users/data-analytics/data-analytics/python/
- OLCF: https://www.olcf.ornl.gov/training-event/2016-olcf-user-conference-call-olcf-python-best-practices/

Other presentations:

- ALCF Performance Workshop (May 2017):
  Python on HPC Best Practices http://www.alcf.anl.gov/files/scullin-python.pdf
- NERSC Intel Python Training Event (March 2017):
  Optimization Example http://www.nersc.gov/assets/Uploads/Intel-tomopy-Mar2017.pdf
  by Oleksandr Pavlyk (Intel)

70

Conclusion

- NERSC, ALCF, and OLCF recognize, welcome, and want to support new and experienced Python users in HPC.
- Using Python on our systems can be as easy as a module load, but can be customized by users.
- We have provided some guidance and best practices to help users improve Python performance in HPC context.
- Try out some of the profiling and performance analysis tools described here, and ask for help if you get stuck.
- While there are many challenges for Python in HPC, if users, staff, & vendors work together, there are many rewards.

71

Thank you!

Cross-Compiling on Cray XC30s with pip

Instruct Cray compiler wrappers to target the login node architecture so code will run everywhere

```
module unload craype-interlagos
module load craype-istanbul

virualenv --python=python2.7 "${VENV_NAME}"
source "${VENV_NAME}/bin/activate"

# If pip is badly out of date, the TLS certificates may not be trusted.
pip install --trusted-host pypi.python.org --upgrade pip
```

Set envvars needed to guide pip for cross-compiling and instruct it to build from source

```
CC=cc MPICC=cc pip install -v --no-binary :all: mpi4py
```

Set envvars needed for pip to use external dependencies. See package documentation.

```
HDF5_DIR="${CRAY_HDF5_DIR}/${PE_ENV}/${GNU_VERSION%.*}"
CC=cc HDF5_MPI="ON" HDF5_DIR="${HDF5_DIR}" pip install -v --no-binary :all: h5py
deactivate "${VENV_NAME}"
module unload craype-istanbul
module load craype-interlagos
```


73