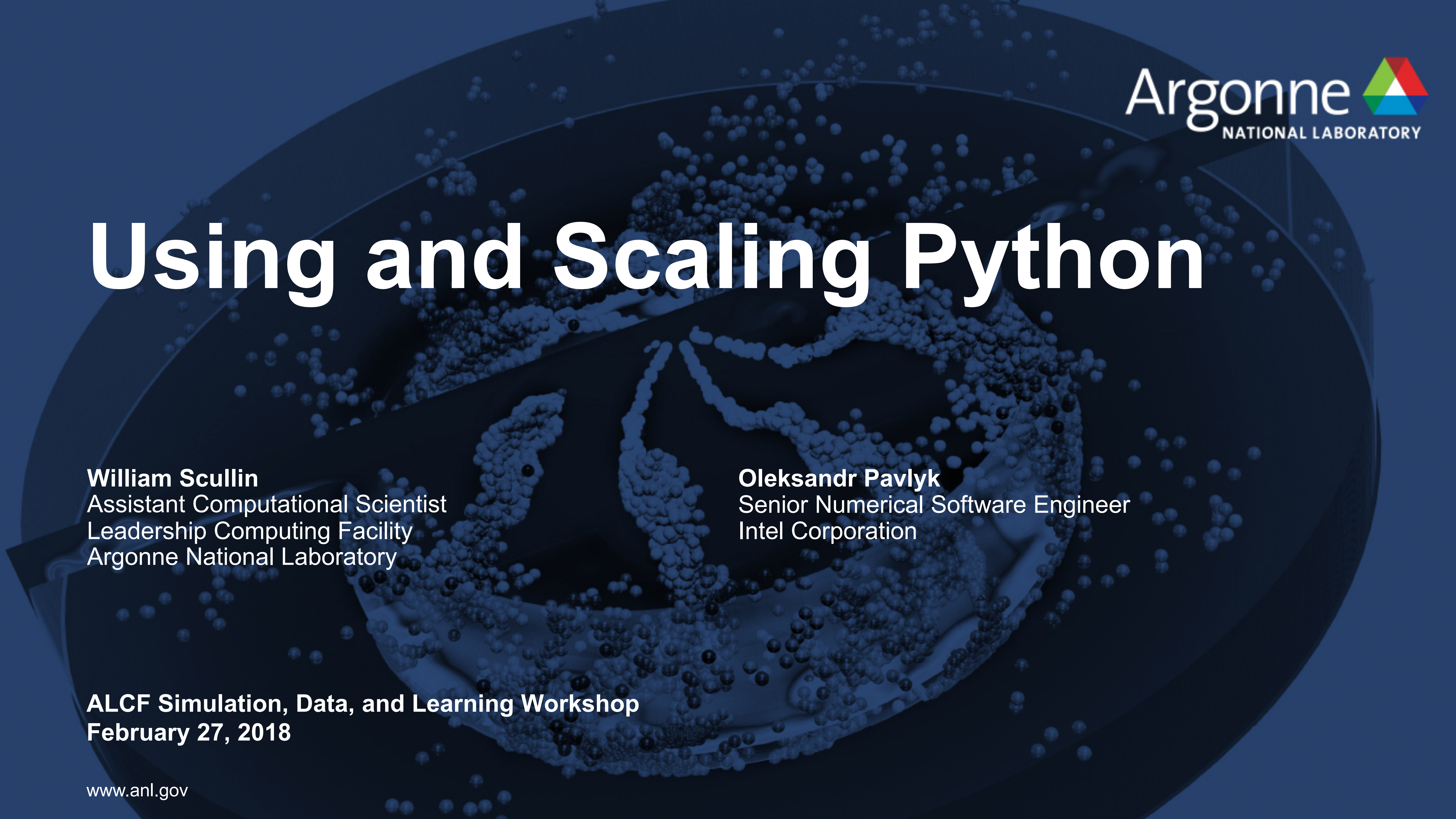


# Using and Scaling Python

**William Scullin**
Assistant Computational Scientist
Leadership Computing Facility
Argonne National Laboratory

**Oleksandr Pavlyk**
Senior Numerical Software Engineer
Intel Corporation

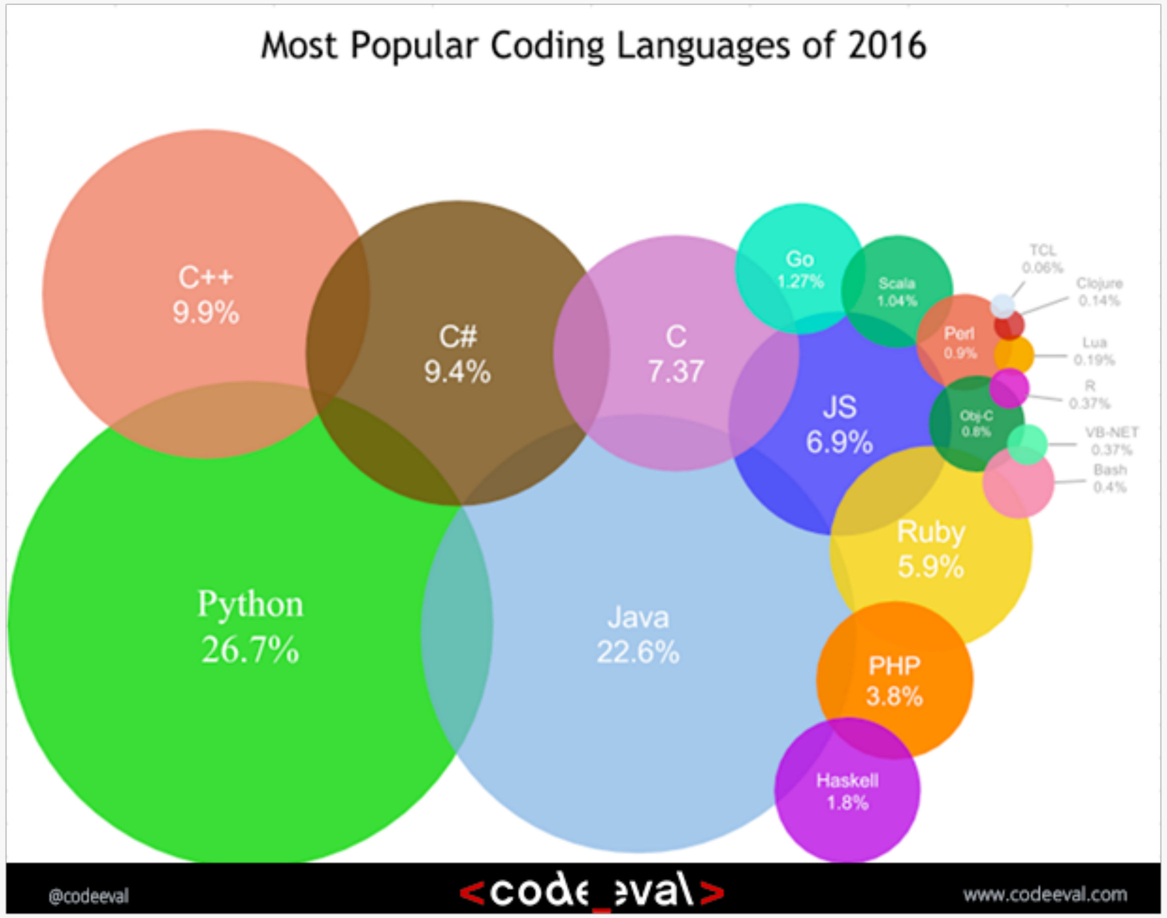
**ALCF Simulation, Data, and Learning Workshop**
**February 27, 2018**

“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 scientific software
- We tend to see a lot of practices and mistages that strongly impact the performance of user Python codes



## 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

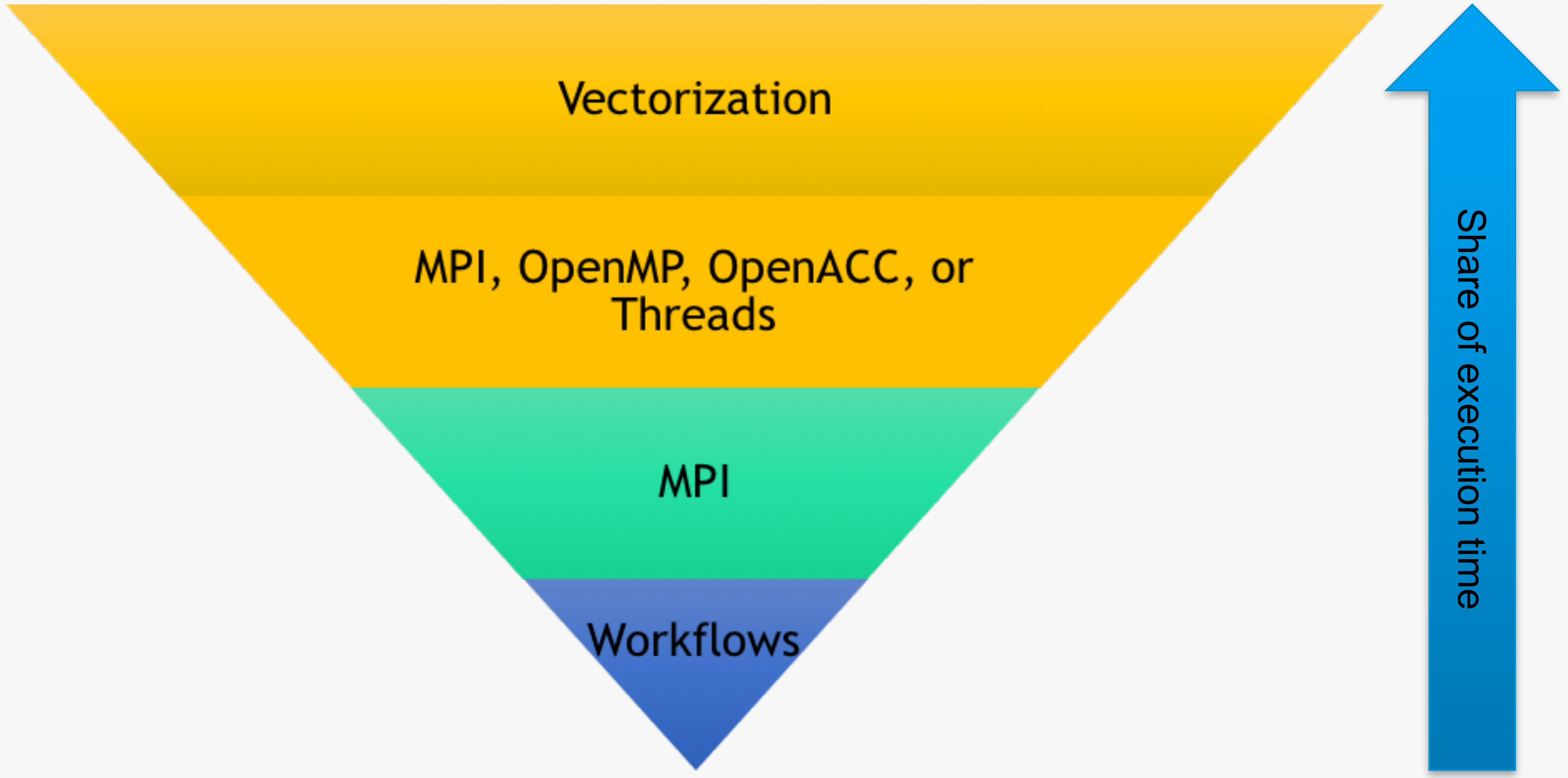
# Python at ALCF

- 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:

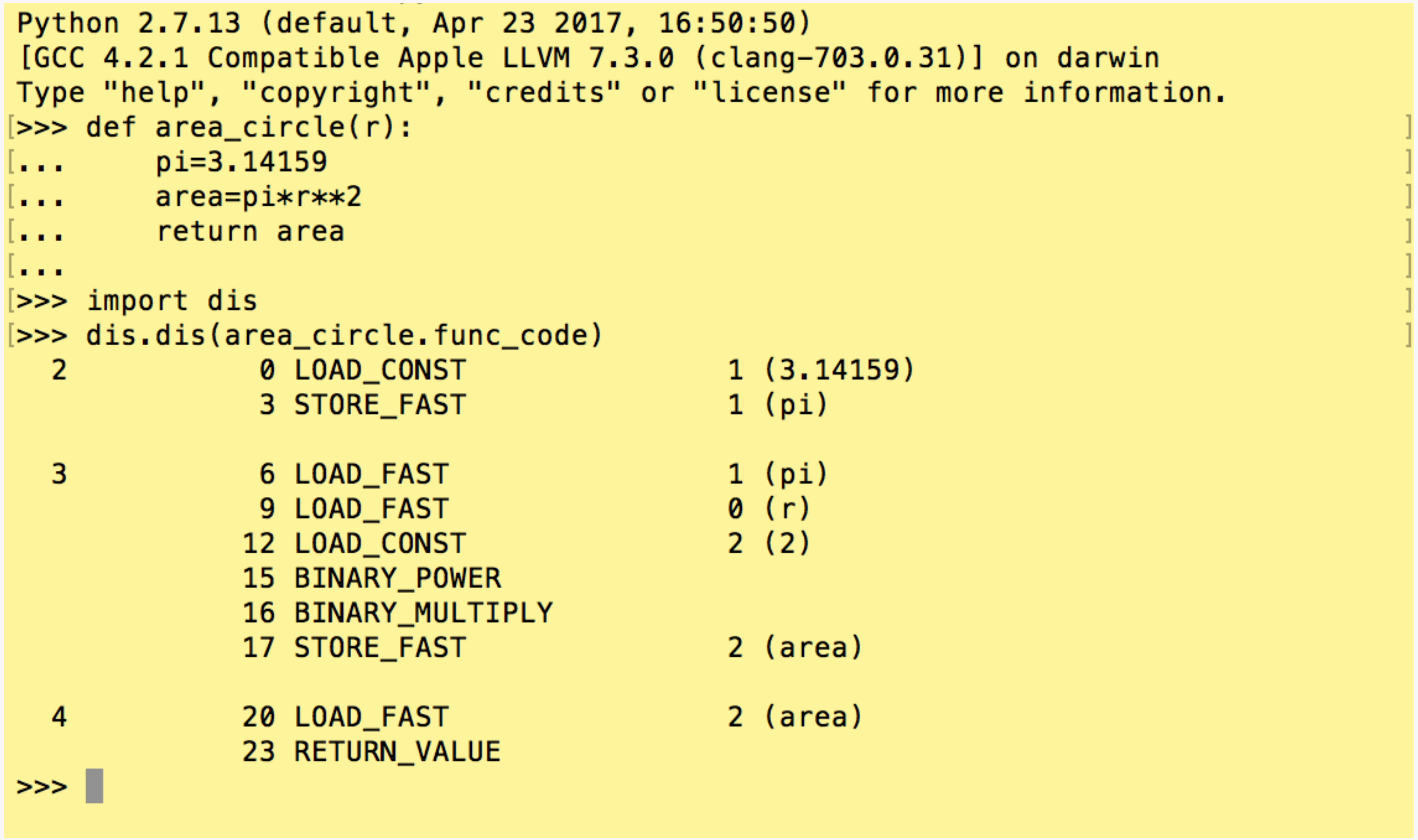
      ```
      module load alcfpython/2.7.14-20180131
      ```

        - A module that loads modules for NumPy, SciPy, MKL, h5py, mpi4py...
        - We build and rebuild alcfpython via Spack to emphasize performance, reproducibility, 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

## Where do We want to spend our time?



# How does CPython work?

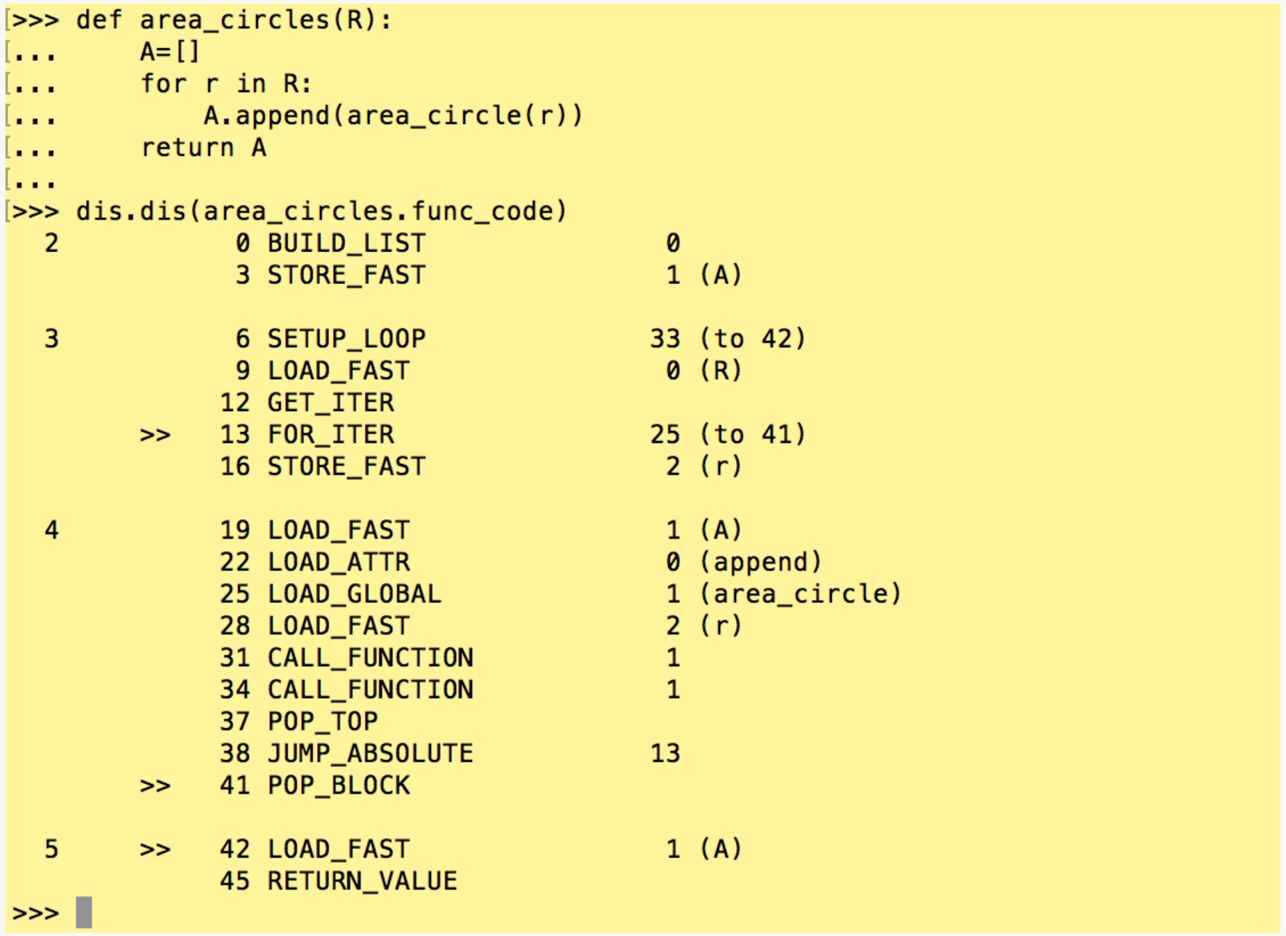


```
Python 2.7.13 (default, Apr 23 2017, 16:50:50)
[GCC 4.2.1 Compatible Apple LLVM 7.3.0 (clang-703.0.31)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> def area_circle(r):
...     pi=3.14159
...     area=pi*r**2
...     return area
...
>>> import dis
>>> dis.dis(area_circle.func_code)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



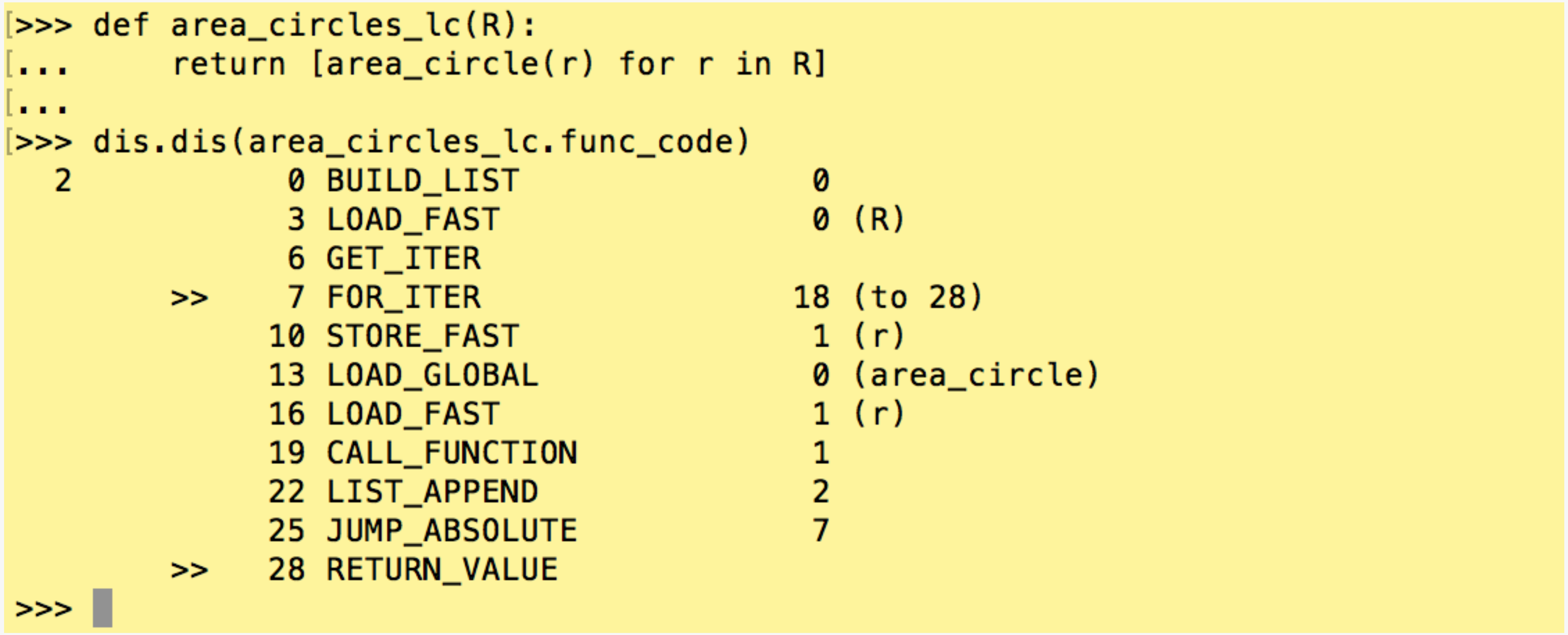
```
[>>> def area_circles(R):
[...     A=[]
[...     for r in R:
[...         A.append(area_circle(r))
[...     return A
[...
[>>> dis.dis(area_circles.func_code)
  2           0 BUILD_LIST               0
              3 STORE_FAST               1 (A)

  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
>>> 
```

# 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.

# Threads and CPython: 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

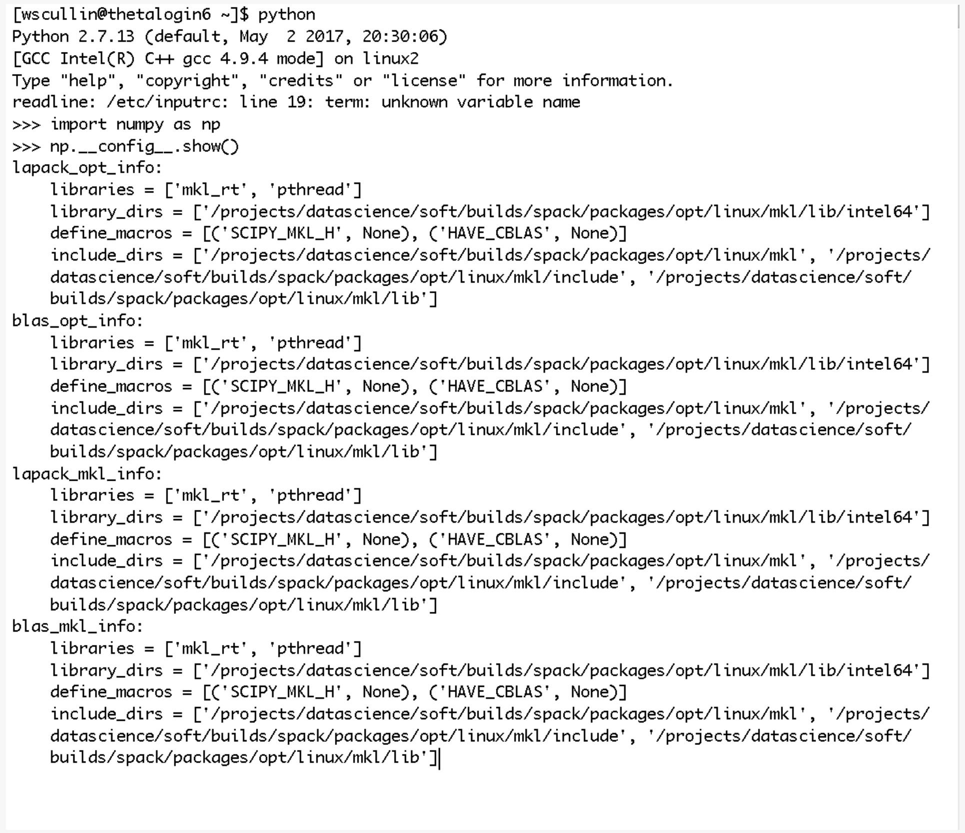
SciPy extends NumPy with common scientific computing tools

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

Problems arise when NumPy isn't well built…

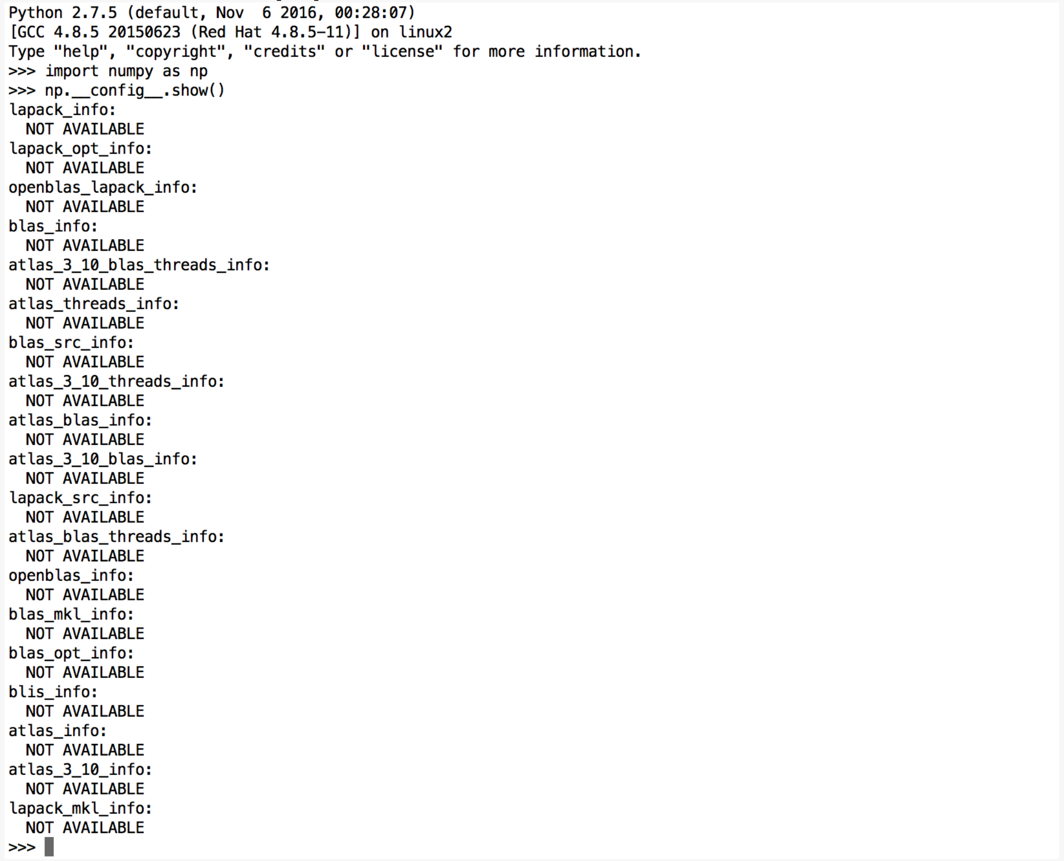
# NumPy and SciPy

## 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']
```

## 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
```

```
119.68859601020813s                    499.9269280433655s
```

# NumPy and SciPy

Using NumPy appropriately pays off:

```
>>> import timeit
>>> import numpy as np
>>> A = np.linspace(-10,10,100).reshape(10,10)
>>> B = np.linspace(-1.0,1.0,100).reshape(10,10)
>>>
>>> def mat_mult(A,B):
...     """ We're assuming regular 2D NumPy matrixes with dimensions such that
...         A.shape[1] == B.shape[0]"""
...     assert A.shape[1] == B.shape[0], "A[1].shape != B[0].shape"
...     C=np.zeros((A.shape[0],B.shape[1]))
...     for i in range(A.shape[0]):
...         for j in range(A.shape[1]):
...             for k in range(B.shape[1]):
...                 C[i,j] += A[i,k]*B[k,j]
...     return C
...
...
>>> if __name__ == '__main__':
...     setup_str = "from __main__ import  A,B,mat_mult; import numpy as np"
...     cnt = 100000
...     manual_time = timeit.timeit("mat_mult(A,B)", number=cnt, setup=setup_str)
...     numpy_time  = timeit.timeit("np.matmul(A,B)", number=cnt, setup=setup_str)
...     print("Manual Matmul x%d: %24.6fs" %(cnt, manual_time))
...     print("NumPy  Matmul x%d: %24.6fs" %(cnt, numpy_time))
...
Manual Matmul x100000:               409.429088s
NumPy  Matmul x100000:                 1.660264s
```

## 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 tradeoffs.

We will be supporting two options on Theta:

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

You may also wish to consider a commercial distribution:

- Continuum Analytics Anaconda
- Enthought Canopy
- Cray’s Python

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 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

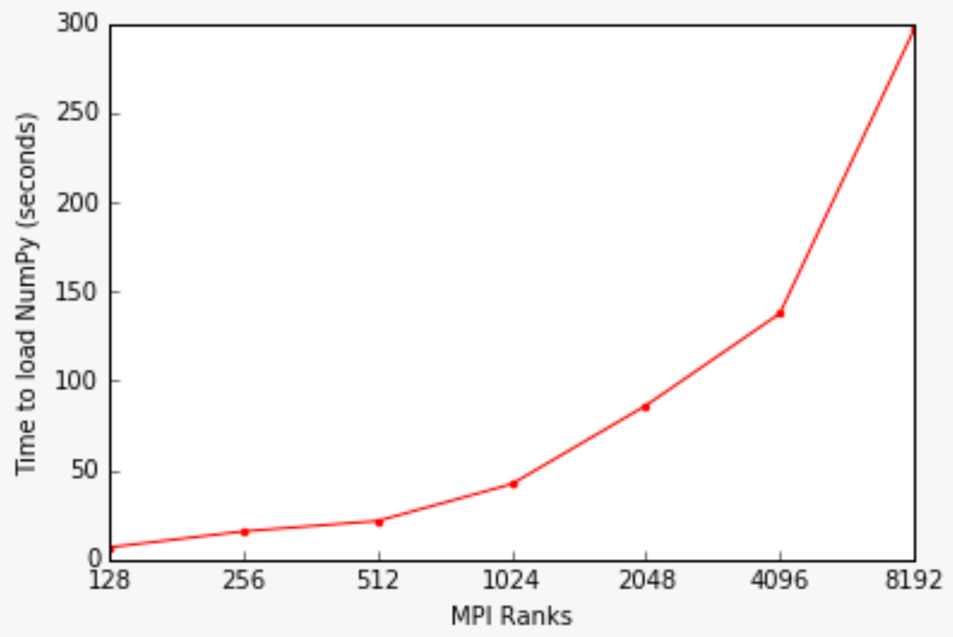
- Debuggers now handle mixed language applications
- Profilers are treating Python as a first-class citizen
- Many parallel solver packages have well-developed Python interfaces

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, e.g.: `Recv()`, `Send()`
- When in doubt: can it be represented as a 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:
  - **lowercase 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)
```

## Profiling Python application

- Right tool for high performance application profiling at all levels
  - Function-level and line-level hotspot analysis, down to disassembly
  - Call stack analysis
  - Low overhead
  - Mixed-language, multi-threaded application analysis
  - Advanced hardware event analysis for native codes (Cython, C++, Fortran) for cache misses, branch misprediction, etc.

| Feature | cProfile | Line_profiler | Intel® VTune™ Amplifier |
|---|---|---|---|
| Profiling technology | Event | Instrumentation | Sampling, hardware events |
| Analysis granularity | Function-level | Line-level | Line-level, call stack, time windows, hardware events |
| Intrusiveness | Medium (1.3-5x) | High (4-10x) | Low (1.05-1.3x) |
| Mixed language programs | Python | Python | Python, Cython, C++, Fortran |

# Using VTune from command line

```
amplxe-cl -collect hotspots \
          -knob sampling-interval=1 \
          -knob analyze-openmp=true \
          -r dir_name_hs
          -- python script.py
```

Analysis type

Analysis parameters, knobs

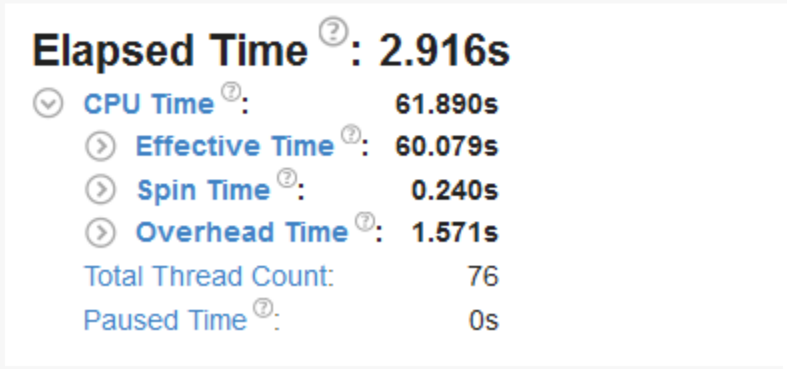
```
amplxe-cl -archive -r dir_name_hs
```

Make trace folder relocatable

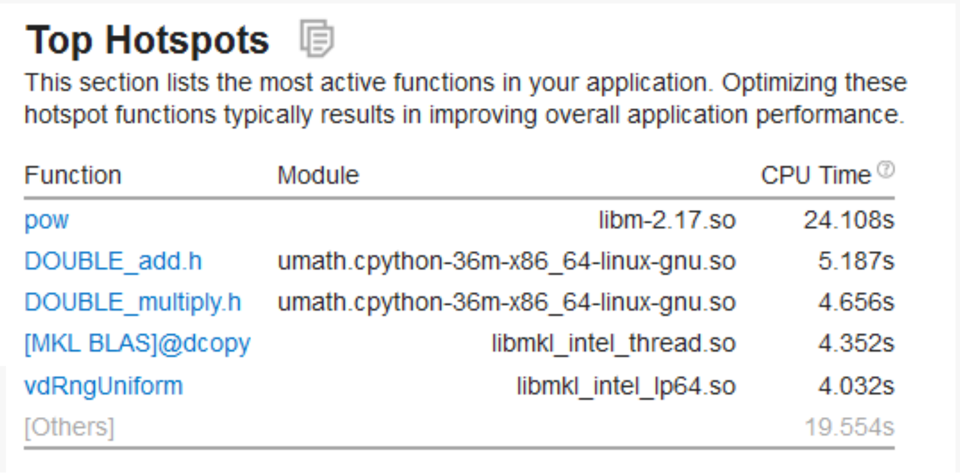
```
amplxe-gui  ./dir_name_hs/ dir_name_hs.amplxe
```

Analyse in GUI on laptop

# VTune trace at a glance [summary tab]



| | |
|---|---|
| **Elapsed Time:** | **2.916s** |
| CPU Time: | 61.890s |
| Effective Time: | 60.079s |
| Spin Time: | 0.240s |
| Overhead Time: | 1.571s |
| Total Thread Count: | 76 |
| Paused Time: | 0s |



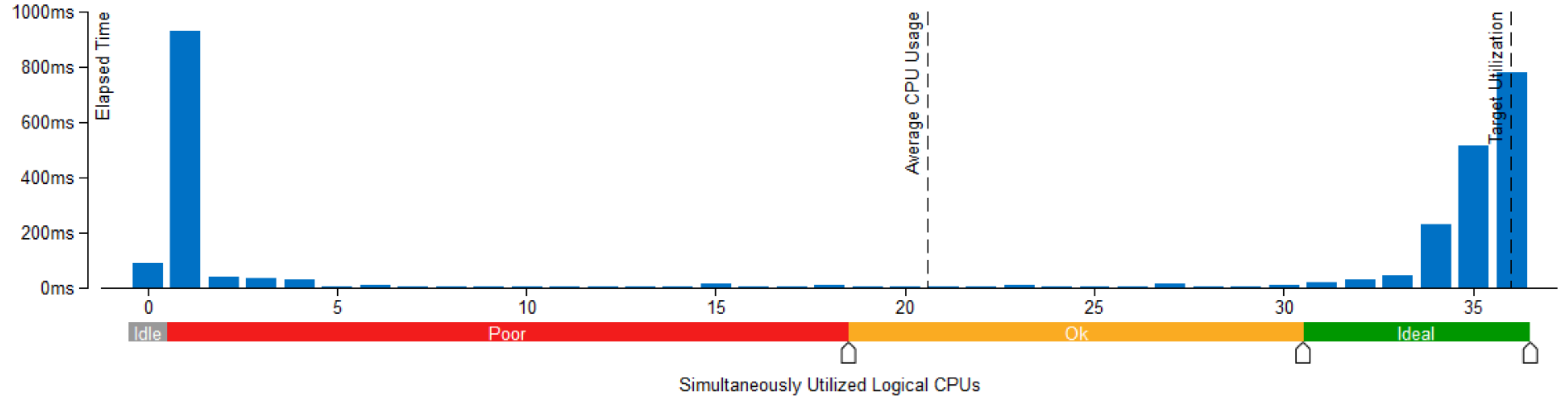
**Top Hotspots**

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

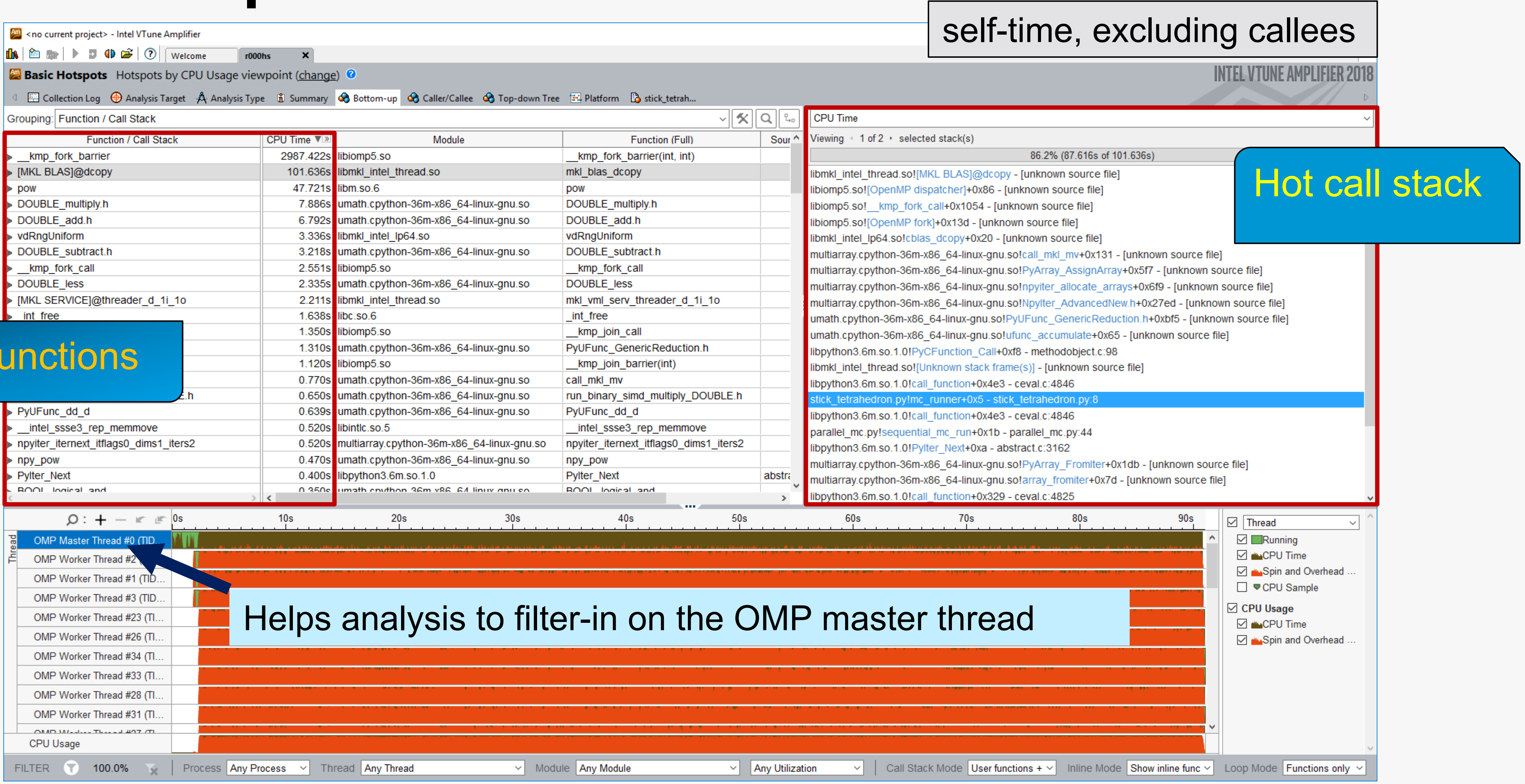
| Function | Module | CPU Time |
|---|---|---|
| pow | libm-2.17.so | 24.108s |
| DOUBLE_add.h | umath.cpython-36m-x86_64-linux-gnu.so | 5.187s |
| DOUBLE_multiply.h | umath.cpython-36m-x86_64-linux-gnu.so | 4.656s |
| [MKL BLAS]@dcopy | libmkl_intel_thread.so | 4.352s |
| vdRngUniform | libmkl_intel_lp64.so | 4.032s |
| [Others] | | 19.554s |

**CPU Usage Histogram**

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.



# Bottom-up view



self-time, excluding callees
Hot functions
Hot call stack
Helps analysis to filter-in on the OMP master thread
<no current project> - Intel VTune Amplifier
Welcome
r000hs
Basic Hotspots Hotspots by CPU Usage viewpoint (change)
INTEL VTUNE AMPLIFIER 2018
Collection Log
Analysis Target
Analysis Type
Summary
Bottom-up
Caller/Callee
Top-down Tree
Platform
stick_tetrah...

Grouping: Function / Call Stack

| Function / Call Stack | CPU Time | Module | Function (Full) | Sour |
|---|---|---|---|---|
| __kmp_fork_barrier | 2987.422s | libiomp5.so | __kmp_fork_barrier(int, int) | |
| [MKL BLAS]@dcopy | 101.636s | libmkl_intel_thread.so | mkl_blas_dcopy | |
| pow | 47.721s | libm.so.6 | pow | |
| DOUBLE_multiply.h | 7.886s | umath.cpython-36m-x86_64-linux-gnu.so | DOUBLE_multiply.h | |
| DOUBLE_add.h | 6.792s | umath.cpython-36m-x86_64-linux-gnu.so | DOUBLE_add.h | |
| vdRngUniform | 3.336s | libmkl_intel_lp64.so | vdRngUniform | |
| DOUBLE_subtract.h | 3.218s | umath.cpython-36m-x86_64-linux-gnu.so | DOUBLE_subtract.h | |
| __kmp_fork_call | 2.551s | libiomp5.so | __kmp_fork_call | |
| DOUBLE_less | 2.335s | umath.cpython-36m-x86_64-linux-gnu.so | DOUBLE_less | |
| [MKL SERVICE]@threader_d_1i_1o | 2.211s | libmkl_intel_thread.so | mkl_vml_serv_threader_d_1i_1o | |
| _int_free | 1.638s | libc.so.6 | _int_free | |
| | 1.350s | libiomp5.so | __kmp_join_call | |
| | 1.310s | umath.cpython-36m-x86_64-linux-gnu.so | PyUFunc_GenericReduction.h | |
| | 1.120s | libiomp5.so | __kmp_join_barrier(int) | |
| | 0.770s | umath.cpython-36m-x86_64-linux-gnu.so | call_mkl_mv | |
| .h | 0.650s | umath.cpython-36m-x86_64-linux-gnu.so | run_binary_simd_multiply_DOUBLE.h | |
| PyUFunc_dd_d | 0.639s | umath.cpython-36m-x86_64-linux-gnu.so | PyUFunc_dd_d | |
| __intel_ssse3_rep_memmove | 0.520s | libintlc.so.5 | __intel_ssse3_rep_memmove | |
| npyiter_iternext_itflags0_dims1_iters2 | 0.520s | multiarray.cpython-36m-x86_64-linux-gnu.so | npyiter_iternext_itflags0_dims1_iters2 | |
| npy_pow | 0.470s | umath.cpython-36m-x86_64-linux-gnu.so | npy_pow | |
| PyIter_Next | 0.400s | libpython3.6m.so.1.0 | PyIter_Next | abstr |
| BOOL_logical_and | 0.350s | umath.cpython-36m-x86_64-linux-gnu.so | BOOL_logical_and | |

CPU Time

Viewing 1 of 2 selected stack(s)

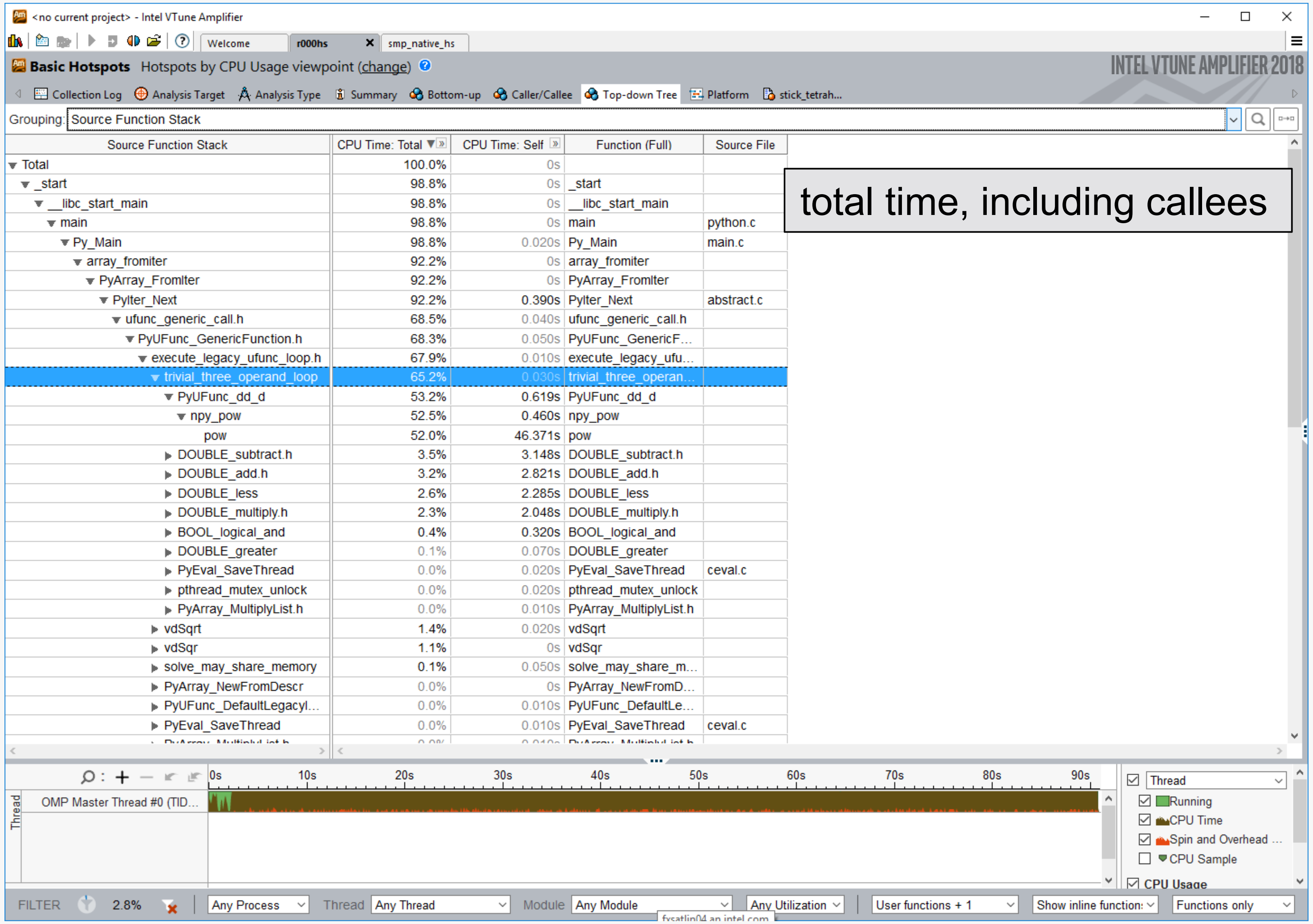
86.2% (87.616s of 101.636s)

```
libmkl_intel_thread.so![MKL BLAS]@dcopy - [unknown source file]
libiomp5.so![OpenMP dispatcher]+0x86 - [unknown source file]
libiomp5.so!__kmp_fork_call+0x1054 - [unknown source file]
libiomp5.so![OpenMP fork]+0x13d - [unknown source file]
libmkl_intel_lp64.so!cblas_dcopy+0x20 - [unknown source file]
multiarray.cpython-36m-x86_64-linux-gnu.so!call_mkl_mv+0x131 - [unknown source file]
multiarray.cpython-36m-x86_64-linux-gnu.so!PyArray_AssignArray+0x5f7 - [unknown source file]
multiarray.cpython-36m-x86_64-linux-gnu.so!npyiter_allocate_arrays+0x6f9 - [unknown source file]
multiarray.cpython-36m-x86_64-linux-gnu.so!NpyIter_AdvancedNew.h+0x27ed - [unknown source file]
umath.cpython-36m-x86_64-linux-gnu.so!PyUFunc_GenericReduction.h+0xbf5 - [unknown source file]
umath.cpython-36m-x86_64-linux-gnu.so!ufunc_accumulate+0x65 - [unknown source file]
libpython3.6m.so.1.0!PyCFunction_Call+0xf8 - methodobject.c:98
libmkl_intel_thread.so![Unknown stack frame(s)] - [unknown source file]
libpython3.6m.so.1.0!call_function+0x4e3 - ceval.c:4846
stick_tetrahedron.py!mc_runner+0x5 - stick_tetrahedron.py:8
libpython3.6m.so.1.0!call_function+0x4e3 - ceval.c:4846
parallel_mc.py!sequential_mc_run+0x1b - parallel_mc.py:44
libpython3.6m.so.1.0!PyIter_Next+0xa - abstract.c:3162
multiarray.cpython-36m-x86_64-linux-gnu.so!PyArray_FromIter+0x1db - [unknown source file]
multiarray.cpython-36m-x86_64-linux-gnu.so!array_fromiter+0x7d - [unknown source file]
libpython3.6m.so.1.0!call_function+0x329 - ceval.c:4825
```

0s
10s
20s
30s
40s
50s
60s
70s
80s
90s
Thread
OMP Master Thread #0 (TID...
OMP Worker Thread #2
OMP Worker Thread #1 (TID...
OMP Worker Thread #3 (TID...
OMP Worker Thread #23 (TI...
OMP Worker Thread #26 (TI...
OMP Worker Thread #34 (TI...
OMP Worker Thread #33 (TI...
OMP Worker Thread #28 (TI...
OMP Worker Thread #31 (TI...
CPU Usage
Thread
☑ 🟩 Running
☑ CPU Time
☑ Spin and Overhead ...
☐ CPU Sample
☑ CPU Usage
☑ CPU Time
☑ Spin and Overhead ...

FILTER 100.0% Process: Any Process; Thread: Any Thread; Module: Any Module; Any Utilization; Call Stack Mode: User functions +; Inline Mode: Show inline func; Loop Mode: Functions only

# Top-down view



| Source Function Stack | CPU Time: Total | CPU Time: Self | Function (Full) | Source File |
|---|---|---|---|---|
| Total | 100.0% | 0s | | |
| _start | 98.8% | 0s | _start | |
| __libc_start_main | 98.8% | 0s | __libc_start_main | |
| main | 98.8% | 0s | main | python.c |
| Py_Main | 98.8% | 0.020s | Py_Main | main.c |
| array_fromiter | 92.2% | 0s | array_fromiter | |
| PyArray_FromIter | 92.2% | 0s | PyArray_FromIter | |
| PyIter_Next | 92.2% | 0.390s | PyIter_Next | abstract.c |
| ufunc_generic_call.h | 68.5% | 0.040s | ufunc_generic_call.h | |
| PyUFunc_GenericFunction.h | 68.3% | 0.050s | PyUFunc_GenericF… | |
| execute_legacy_ufunc_loop.h | 67.9% | 0.010s | execute_legacy_ufu… | |
| trivial_three_operand_loop | 65.2% | 0.030s | trivial_three_operan… | |
| PyUFunc_dd_d | 53.2% | 0.619s | PyUFunc_dd_d | |
| npy_pow | 52.5% | 0.460s | npy_pow | |
| pow | 52.0% | 46.371s | pow | |
| DOUBLE_subtract.h | 3.5% | 3.148s | DOUBLE_subtract.h | |
| DOUBLE_add.h | 3.2% | 2.821s | DOUBLE_add.h | |
| DOUBLE_less | 2.6% | 2.285s | DOUBLE_less | |
| DOUBLE_multiply.h | 2.3% | 2.048s | DOUBLE_multiply.h | |
| BOOL_logical_and | 0.4% | 0.320s | BOOL_logical_and | |
| DOUBLE_greater | 0.1% | 0.070s | DOUBLE_greater | |
| PyEval_SaveThread | 0.0% | 0.020s | PyEval_SaveThread | ceval.c |
| pthread_mutex_unlock | 0.0% | 0.020s | pthread_mutex_unlock | |
| PyArray_MultiplyList.h | 0.0% | 0.010s | PyArray_MultiplyList.h | |
| vdSqrt | 1.4% | 0.020s | vdSqrt | |
| vdSqr | 1.1% | 0s | vdSqr | |
| solve_may_share_memory | 0.1% | 0.050s | solve_may_share_m… | |
| PyArray_NewFromDescr | 0.0% | 0s | PyArray_NewFromD… | |
| PyUFunc_DefaultLegacyI… | 0.0% | 0.010s | PyUFunc_DefaultLe… | |
| PyEval_SaveThread | 0.0% | 0.010s | PyEval_SaveThread | ceval.c |

total time, including callees

## Python mode vs. native mode

amplxe-cl … -mrte-mode=native -- python script.py ← treat python application as a binary

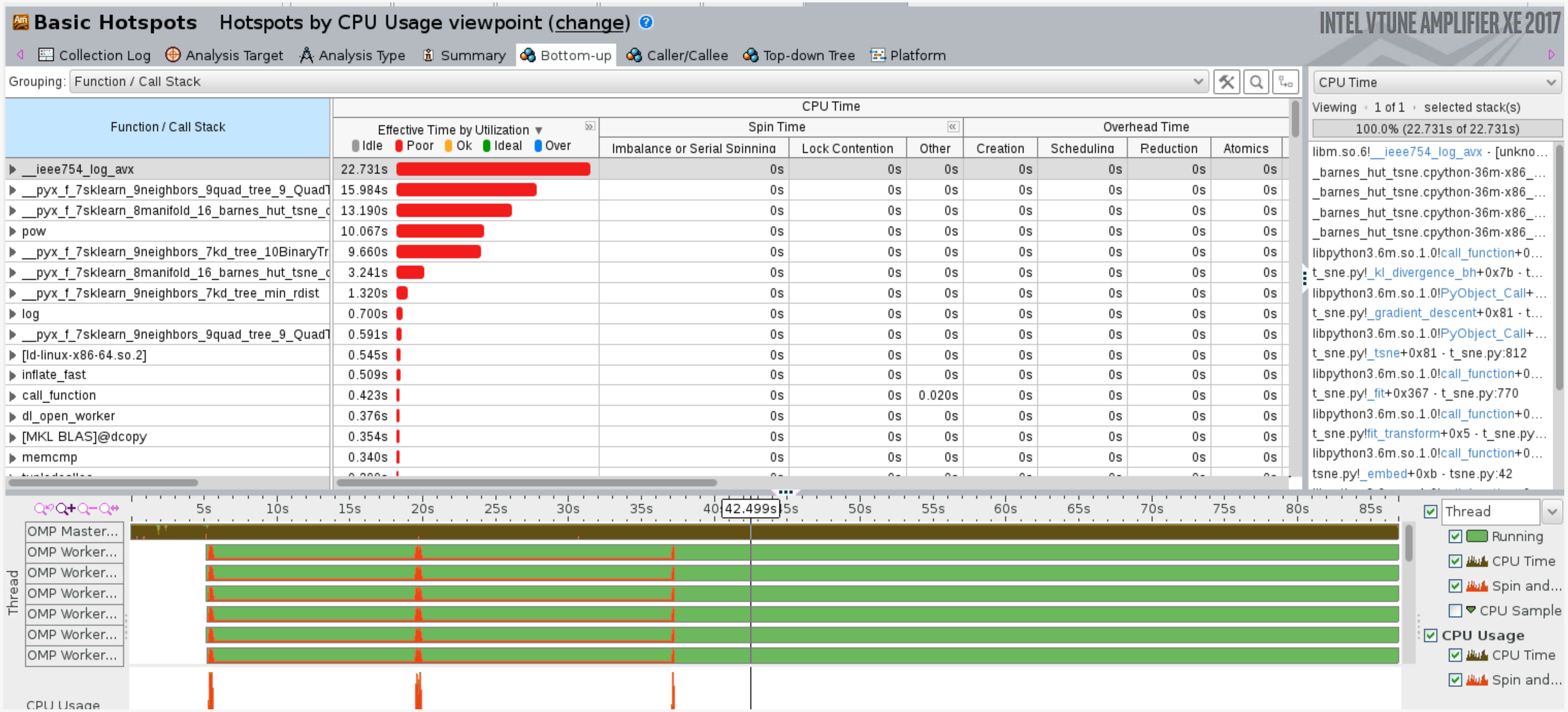
VTune will no see python functions, only C functions behind them.

Useful if the default -mrte-mode=python is running into issues.

amplxe-cl ….  -mrte-mode=native -run-pass-thru=--no-altstack – python script.py

On Linux, if "stack size too small" error seen.

# Cython: seen as generated C



Grouping: Function / Call Stack

| Function / Call Stack | CPU Time | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Effective Time by Utilization ▼ (Idle, Poor, Ok, Ideal, Over) | Spin Time | | | Overhead Time | | | |
| | | Imbalance or Serial Spinning | Lock Contention | Other | Creation | Scheduling | Reduction | Atomics |
| ▶ __ieee754_log_avx | 22.731s | 0s | 0s | 0s | 0s | 0s | 0s | 0s |
| ▶ __pyx_f_7sklearn_9neighbors_9quad_tree_9_QuadT | 15.984s | 0s | 0s | 0s | 0s | 0s | 0s | 0s |
| ▶ __pyx_f_7sklearn_8manifold_16_barnes_hut_tsne_c | 13.190s | 0s | 0s | 0s | 0s | 0s | 0s | 0s |
| ▶ pow | 10.067s | 0s | 0s | 0s | 0s | 0s | 0s | 0s |
| ▶ __pyx_f_7sklearn_9neighbors_7kd_tree_10BinaryTr | 9.660s | 0s | 0s | 0s | 0s | 0s | 0s | 0s |
| ▶ __pyx_f_7sklearn_8manifold_16_barnes_hut_tsne_c | 3.241s | 0s | 0s | 0s | 0s | 0s | 0s | 0s |
| ▶ __pyx_f_7sklearn_9neighbors_7kd_tree_min_rdist | 1.320s | 0s | 0s | 0s | 0s | 0s | 0s | 0s |
| ▶ log | 0.700s | 0s | 0s | 0s | 0s | 0s | 0s | 0s |
| ▶ __pyx_f_7sklearn_9neighbors_9quad_tree_9_QuadT | 0.591s | 0s | 0s | 0s | 0s | 0s | 0s | 0s |
| ▶ [ld-linux-x86-64.so.2] | 0.545s | 0s | 0s | 0s | 0s | 0s | 0s | 0s |
| ▶ inflate_fast | 0.509s | 0s | 0s | 0s | 0s | 0s | 0s | 0s |
| ▶ call_function | 0.423s | 0s | 0s | 0.020s | 0s | 0s | 0s | 0s |
| ▶ dl_open_worker | 0.376s | 0s | 0s | 0s | 0s | 0s | 0s | 0s |
| ▶ [MKL BLAS]@dcopy | 0.354s | 0s | 0s | 0s | 0s | 0s | 0s | 0s |
| ▶ memcmp | 0.340s | 0s | 0s | 0s | 0s | 0s | 0s | 0s |

CPU Time

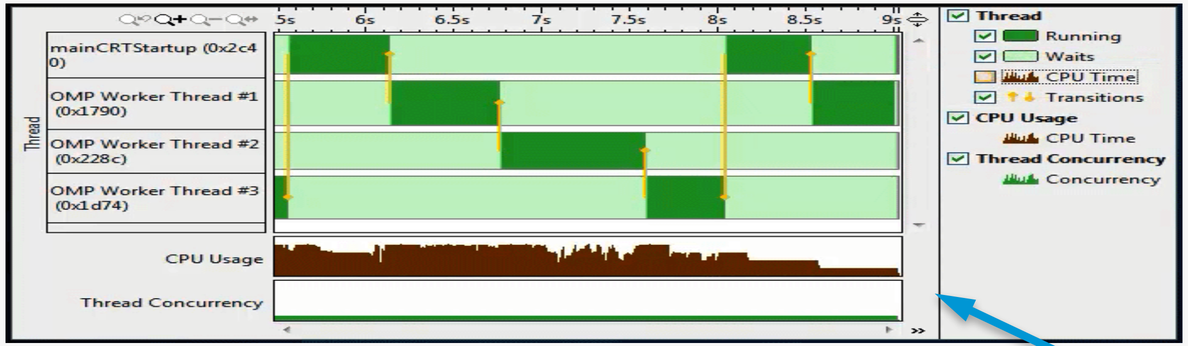
Viewing 1 of 1 selected stack(s)

100.0% (22.731s of 22.731s)

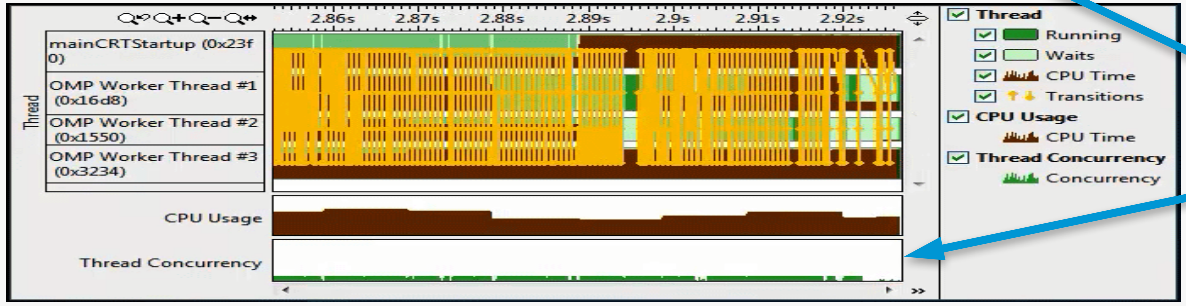
```
libm.so.6!__ieee754_log_avx - [unkno…
_barnes_hut_tsne.cpython-36m-x86_…
_barnes_hut_tsne.cpython-36m-x86_…
_barnes_hut_tsne.cpython-36m-x86_…
_barnes_hut_tsne.cpython-36m-x86_…
libpython3.6m.so.1.0!call_function+0…
t_sne.py!_kl_divergence_bh+0x7b - t…
libpython3.6m.so.1.0!PyObject_Call+…
t_sne.py!_gradient_descent+0x81 - t…
libpython3.6m.so.1.0!PyObject_Call+…
t_sne.py!_tsne+0x81 - t_sne.py:812
libpython3.6m.so.1.0!call_function+0…
t_sne.py!_fit+0x367 - t_sne.py:770
libpython3.6m.so.1.0!call_function+0…
t_sne.py!fit_transform+0x5 - t_sne.py…
libpython3.6m.so.1.0!call_function+0…
tsne.py!_embed+0xb - tsne.py:42
```

# Common problematic patterns

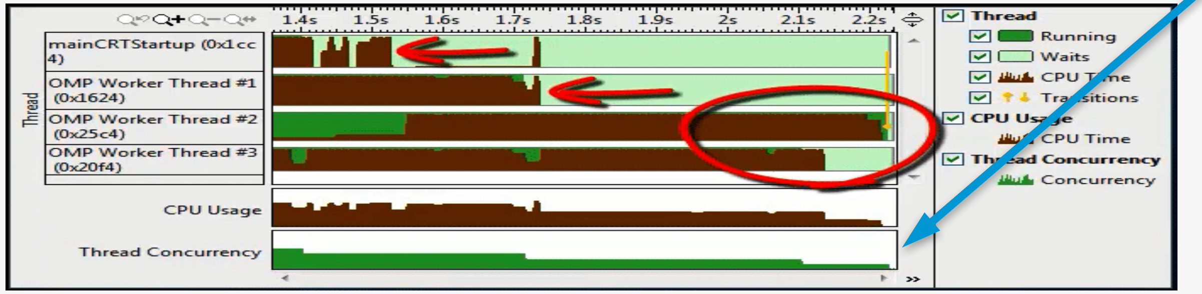
Coarse Grain Locks



High Lock Contention



Load Imbalance



**Low Concurrency**

# Intel® Trace Collector & Analyzer

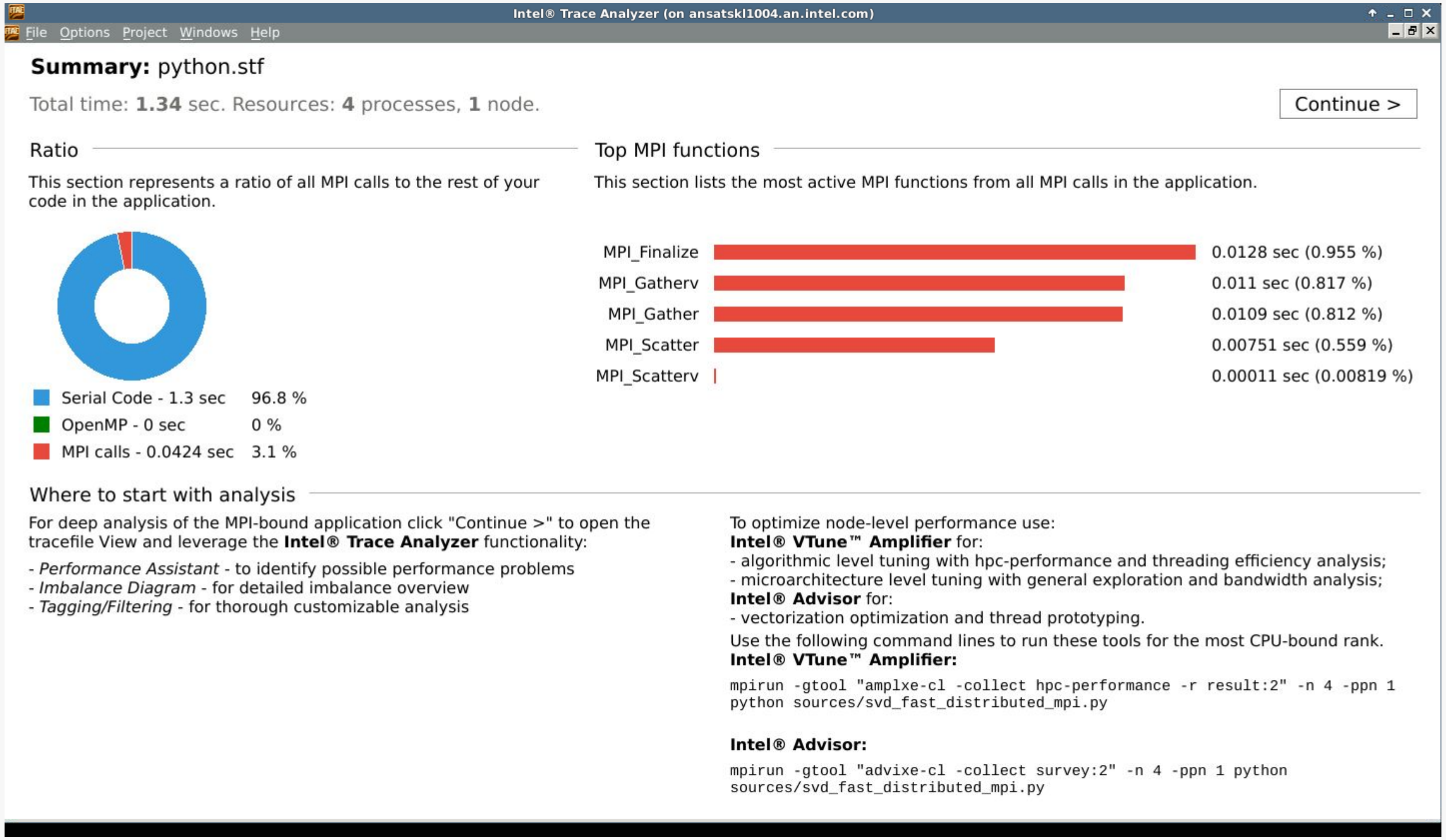
collect traces

```
LD_PRELOAD="/opt/intel/itac/2018.1.017/intel64/slib/libVT.so $CONDA_PREFIX/lib/libmpi.so" \
          mpiexec -n 4  python script.py
```

```
… mpiexec -gtool "amplxe-cl -collect hpc-performance -r result:1-4" …
```

Use VTune to analyze node-level performance.
Can also use Intel® Advisor:    advixe-cl

## Summary view


Intel® Trace Analyzer (on ansatskl1004.an.intel.com)
File Options Project Windows Help
Continue >

**Summary:** python.stf

Total time: **1.34** sec. Resources: **4** processes, **1** node.

**Ratio**

This section represents a ratio of all MPI calls to the rest of your code in the application.

| | |
|---|---|
| 🟦 Serial Code - 1.3 sec | 96.8 % |
| 🟩 OpenMP - 0 sec | 0 % |
| 🟥 MPI calls - 0.0424 sec | 3.1 % |

**Top MPI functions**

This section lists the most active MPI functions from all MPI calls in the application.

| | |
|---|---|
| MPI_Finalize | 0.0128 sec (0.955 %) |
| MPI_Gatherv | 0.011 sec (0.817 %) |
| MPI_Gather | 0.0109 sec (0.812 %) |
| MPI_Scatter | 0.00751 sec (0.559 %) |
| MPI_Scatterv | 0.00011 sec (0.00819 %) |

**Where to start with analysis**

For deep analysis of the MPI-bound application click "Continue >" to open the tracefile View and leverage the **Intel® Trace Analyzer** functionality:

- *Performance Assistant* - to identify possible performance problems
- *Imbalance Diagram* - for detailed imbalance overview
- *Tagging/Filtering* - for thorough customizable analysis

To optimize node-level performance use:
**Intel® VTune™ Amplifier** for:
- algorithmic level tuning with hpc-performance and threading efficiency analysis;
- microarchitecture level tuning with general exploration and bandwidth analysis;

**Intel® Advisor** for:
- vectorization optimization and thread prototyping.

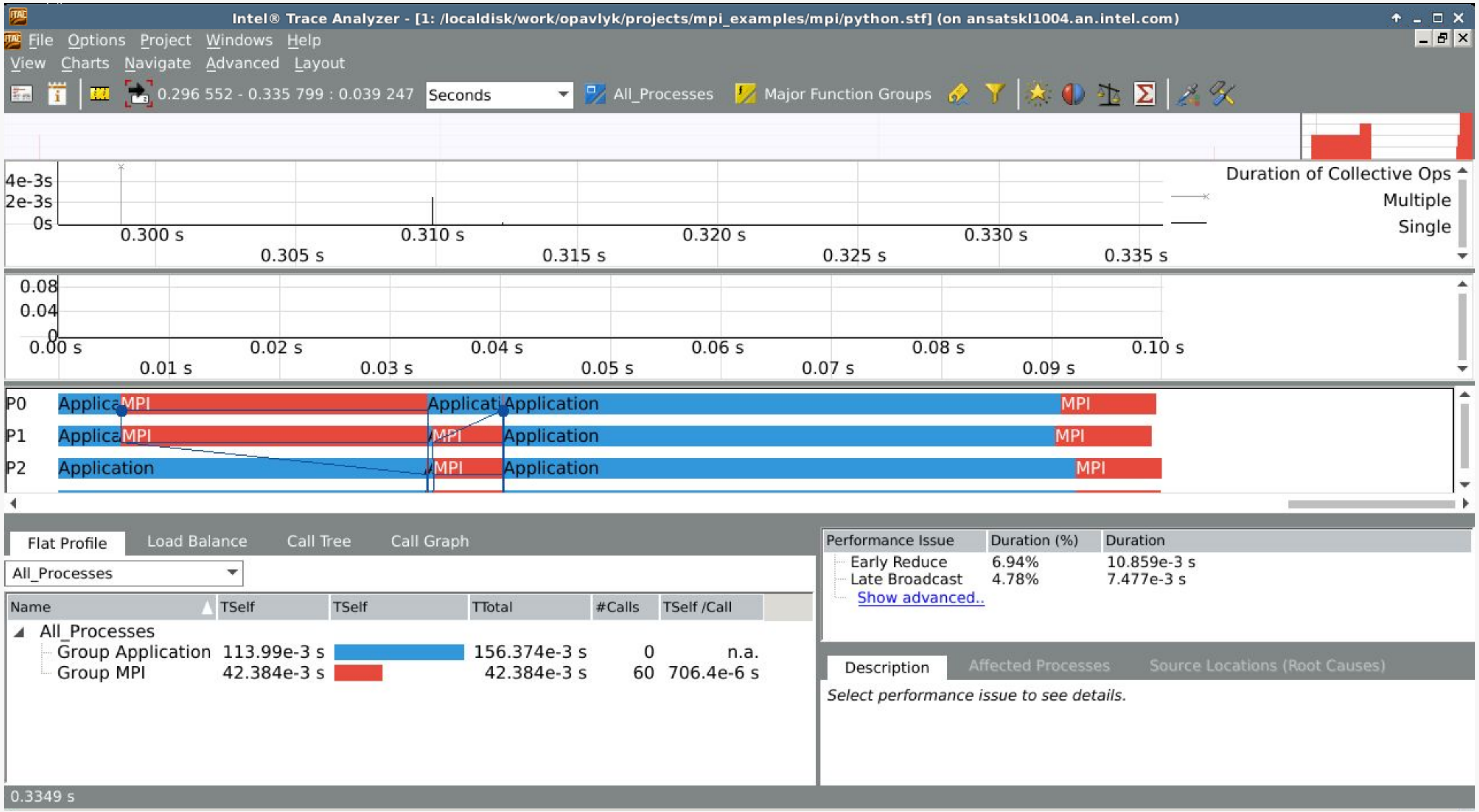
Use the following command lines to run these tools for the most CPU-bound rank.
**Intel® VTune™ Amplifier:**

```
mpirun -gtool "amplxe-cl -collect hpc-performance -r result:2" -n 4 -ppn 1
python sources/svd_fast_distributed_mpi.py
```

**Intel® Advisor:**

```
mpirun -gtool "advixe-cl -collect survey:2" -n 4 -ppn 1 python
sources/svd_fast_distributed_mpi.py
```

# Time-line view in traceanalyzer



Intel® Trace Analyzer - [1: /localdisk/work/opavlyk/projects/mpi_examples/mpi/python.stf] (on ansatskl1004.an.intel.com)

| Name | TSelf | TSelf | TTotal | #Calls | TSelf /Call |
|---|---|---|---|---|---|
| All_Processes | | | | | |
| Group Application | 113.99e-3 s | 🟦 | 156.374e-3 s | 0 | n.a. |
| Group MPI | 42.384e-3 s | 🟥 | 42.384e-3 s | 60 | 706.4e-6 s |

| Performance Issue | Duration (%) | Duration |
|---|---|---|
| Early Reduce | 6.94% | 10.859e-3 s |
| Late Broadcast | 4.78% | 7.477e-3 s |
| Show advanced.. | | |

*Select performance issue to see details.*

0.3349 s

# Enumerated admonishments

- Benchmark and profile as you develop
- Control your environment
- Ask if you can do an operation with NumPy or SciPy
- Watch your data types – use NumPy datatypes
- Never mix forking and threading – ie: Python multiprocessing
- Avoid threading in Python – use threads in compiled modules
- Check the build configurations of your important Python modules
- Beware of thread affinity:

  ```
  aprun -n … -N … -e KMP_AFFINITY=none -d … -j …
  ```
- Watch startup times carefully
- Search before you write code – someone else has likely already implemented the solution you seek
- On Cray systems, you'll need the `-b` flag to `aprun` with any sort of environment manager

## Questions?

### See also:

#### ECP Python Tutorial:

https://github.com/wscullin/ecp_python_tutorial

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

#### Intel Python Distribution:

http://software.intel.com/en-us/distribution-for-python