Python Parallelization

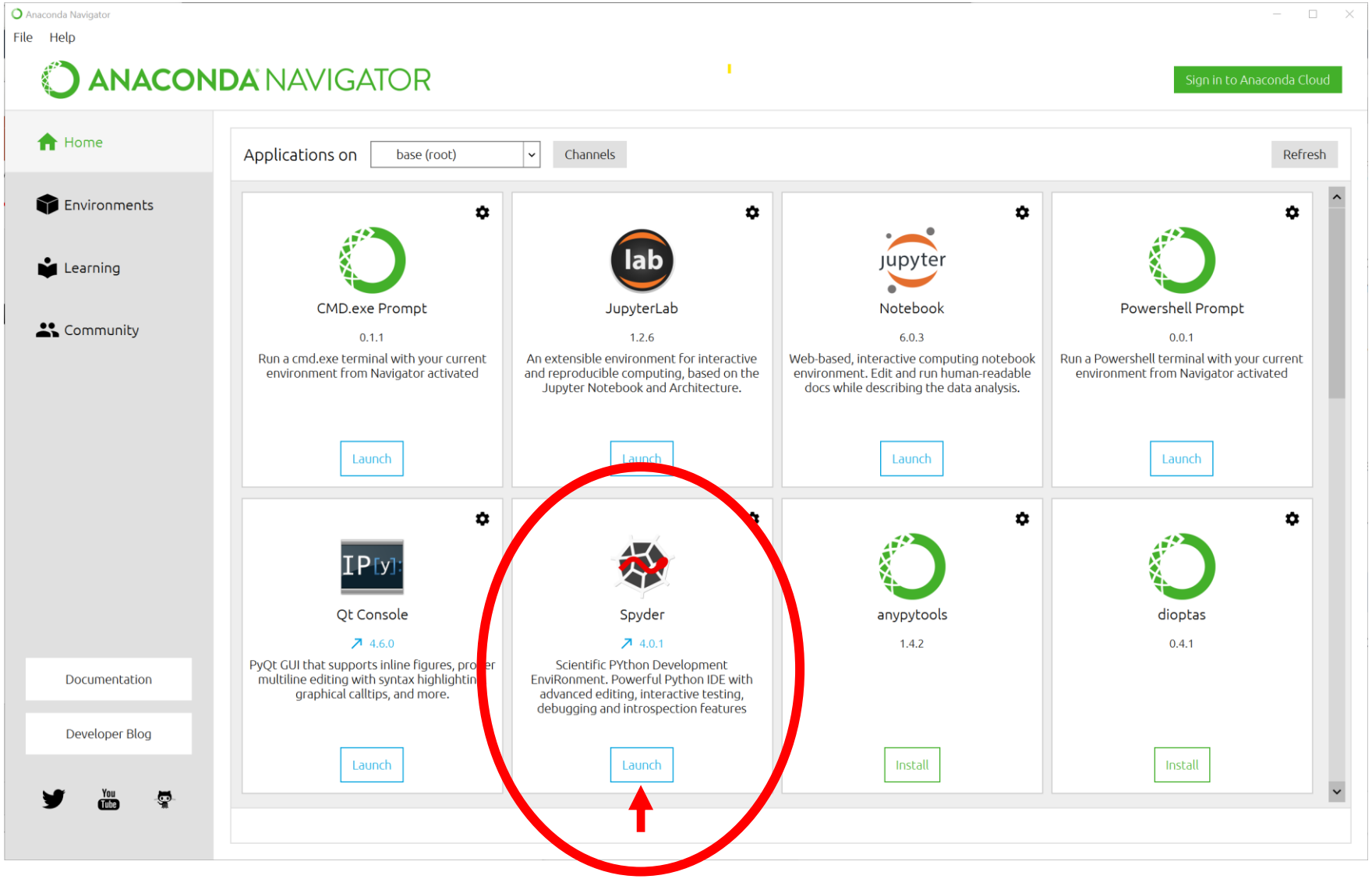
v0.5

Research Computing Services
IS & T

BOSTON UNIVERSITY

Run Spyder

- Start the Anaconda Navigator
- Click on Spyder's Launch button
- Be patient…it takes a while to start.



BOSTON UNIVERSITY

Introduction

- Many programs can perform simultaneous operations, given multiple processors to perform the work.
- Generally speaking, the burden of managing this lies on the programmer.
- In this tutorial we'll go over a variety of ways to achieve parallelism in Python code.

Limits on Program Speed

- **Input/Output** (I/O): The rate at which data can be read from a disk, a network file server, a remote server, a sensor, a user's physical inputs, etc. limits the performance of the program.
- **Memory**: The quantity of memory on the system limits performance.
- **CPU** (or compute): The speed of the processor is the limit on performance.
  - This is most commonly the case for scientific computing.

Types of Parallelization

- On the SCC: queue parallelization.
  - You have N files to process. Submit N jobs.
  - Or, one *job array* that launches N jobs.
  - This often requires little to no changes to your code…
- Multiple Processes
  - Your program launches several copies of itself (or other programs) to solve the computational problem.
- Multiple Threads
  - Your program creates *threads*, which are parts of the **same** program that can execute independently of each other.
- Parallel Libraries
  - Use a library that internally implements some kind of parallelization.

BOSTON UNIVERSITY

Performance Considerations

- Not every part of a program can benefit from parallelization.
- Some parts of program are inherently serial.

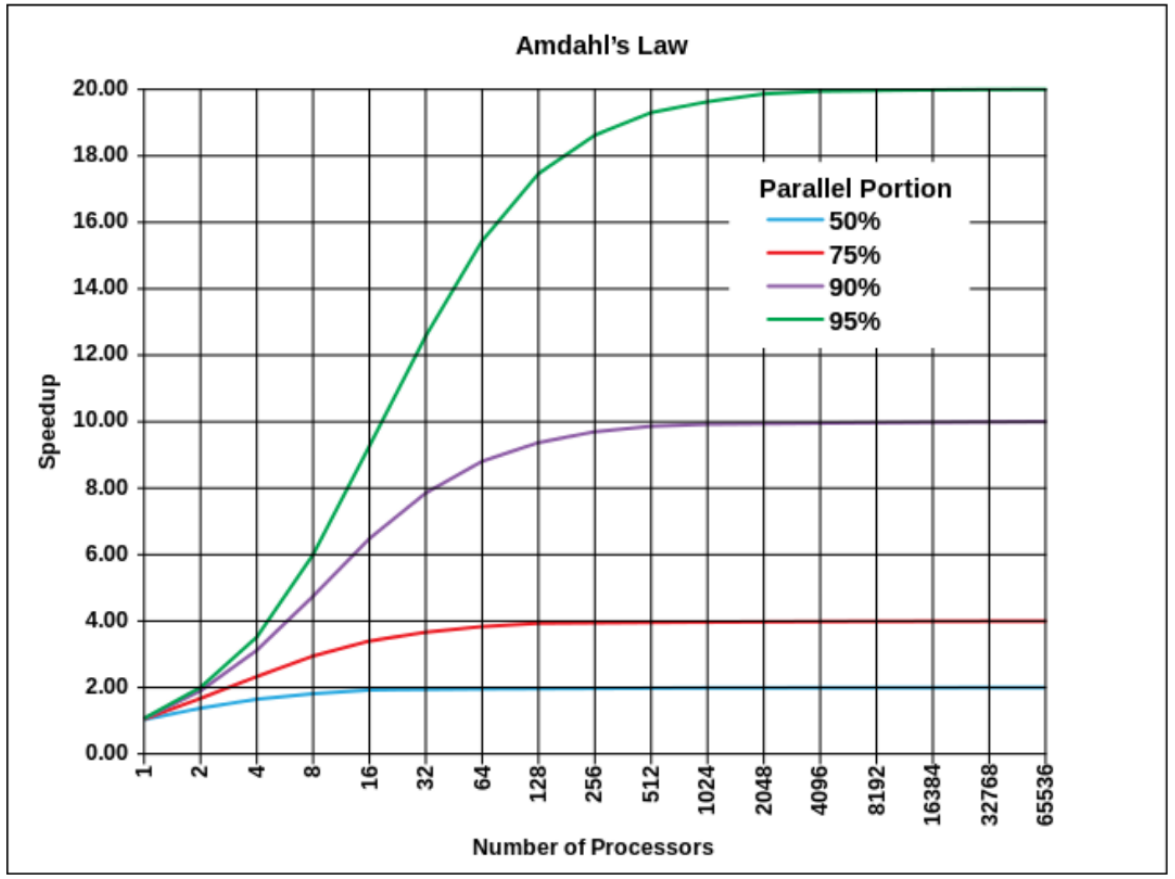
- Even for a function that can be done in parallel…
  - Is it worth the programming effort?
  - Is it worth the reduction in readability and ability to debug?
  - Does the function use up enough program time to make parallel computation worth the overhead?
- Parallelization is a form of optimization. Profile your code.
  - For more on profiling – see our Python Optimization tutorial.

Amdahl's Law

- The speedup ratio *S* is the ratio of time between the serial code (T_1) and the time when using N workers (T_N):

$$S = \frac{T_1}{T_N} = \frac{T_1}{\left(f + \frac{1-f}{N}\right) T_1}$$

N = number of threads or processes
f = fraction of program that is serial



- This is the **theoretical** best speedup achievable with parallelization.

Figure from Wikipedia.

BOSTON UNIVERSITY

A word of caution

- When using the Python *multiprocessing* library, **always** use the *"if __name__"* convention in your main script:

- This will make your script work in interactive Python like Spyder.

```
import multiprocessing
# ...
# python script here with functions
# defined
# ...
def script_function():
    # do python stuff here
    with multiprocessing.Pool(4) as p:
        # code block etc ...

if __name__ == '__main__':
    script_function()
```

- It is **<u>required</u>** on Windows even in Jupyter notebooks.

BOSTON UNIVERSITY

How many cores should Python use?

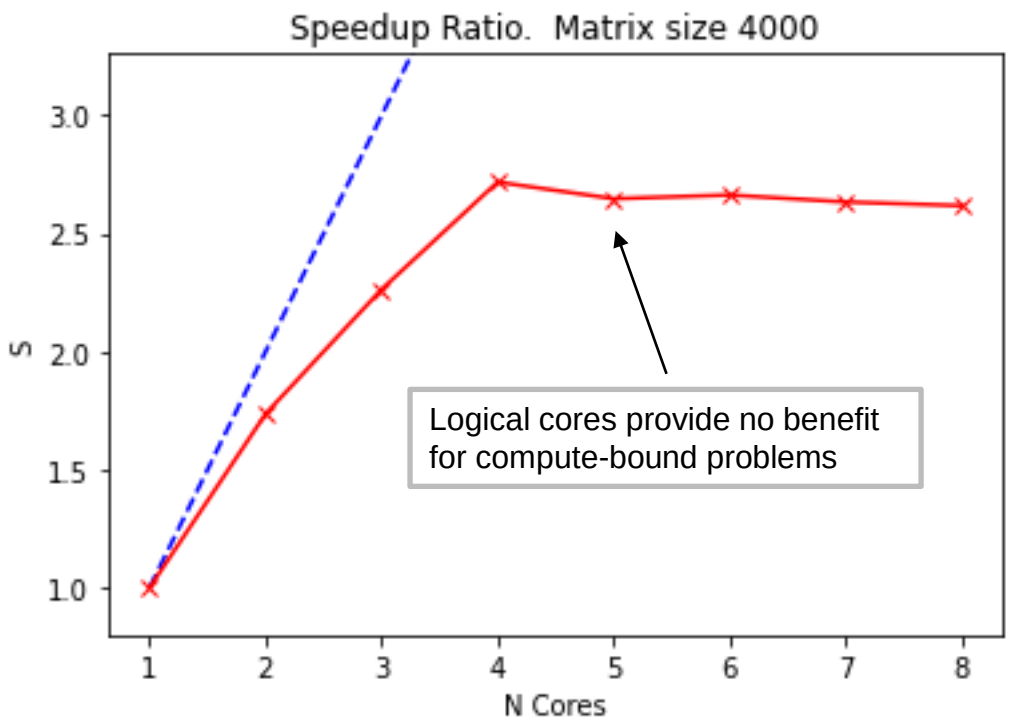
- The example file *get_n_cores.py* provides a function that checks how many cores have been assigned to an SCC job.
  - Based on the common Python library *psutil*
- It will also work on your own computers and will choose the number of installed cores.
- Feel free to use this in your own code.

BOSTON UNIVERSITY

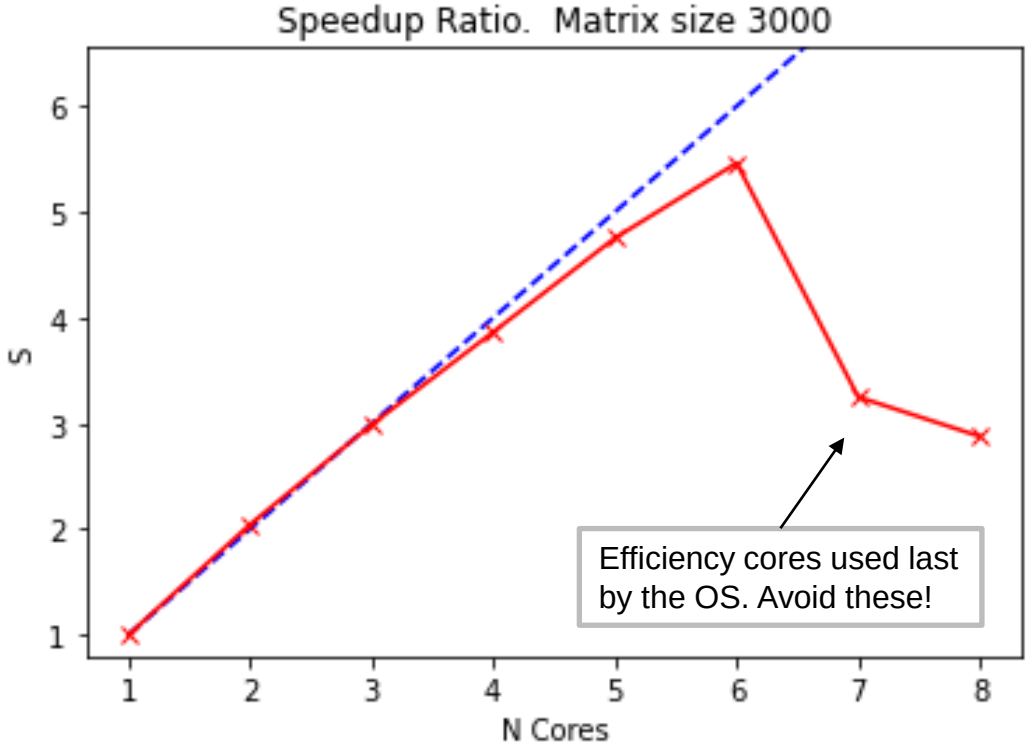
Let's Try!

- In Spyder, open the file *lin_alg.py*
- The computation: a linear algebra matrix-matrix multiplication.
  - Completely CPU-bound, scales well to multiple threads.
- How does your computation scale with the number of threads?
- It plots the speedup ratio. What did you expect? What if you change the size of the matrices?

Logical, Physical, and Efficiency Cores



- Intel Core i7-1165G7
  - 4 real cores, 4 logical cores



- Macbook Pro (from 2021)
  - Apple M1 Pro CPU
    - 6 performance cores, 2 efficiency
      - About This Mac → More Info → System Report
    - get_n_cores() → reports 8 cores

BOSTON UNIVERSITY

The Python *psutil* library can't yet auto-detect efficiency cores. It will report them as physical cores.

Python Language Parallelism

- Python provides a number of ways to perform parallel (aka concurrent) computations.
- Read the official docs.

Library	Common Usage
threading and *asyncio*	I/O-bound programs. Example: web server, network service
multiprocessing	CPU-bound parallel execution.
concurrent.futures	Modern-style wrapper on top of threading & multiprocessing. Useful for GUIs or porting code to Python that uses this approach.
subprocess	Launching external processes.

Python Language Parallelism

- There are many external libraries available.

Library	Common Usage
numba	Function compiler, automatic multithreading
dask	Scalable auto-parallelizing library for data science, including scalable Pandas dataframes (and **much** more). Can use multiple compute nodes.
polars	An auto-multithreaded alternative for Pandas.
joblib	A popular library for straightforward parallelization.
ray	Library that's popular in machine learning applications.
mpire	A newer library, syntax is deliberately very similar to *multiprocessing* with higher performance.

The Global Interpreter Lock

- The GIL limits the amount of multi-threading in the Python interpreter.
  - Originally introduced as part of Python's memory management system.
  - For more details, see this explanation.
- Pure Python code runs in one thread only.
  - This is unlike languages like Java, C#, C++, Fortran, Matlab, or R where threads are easily used by the programmer.
- Multi-threaded code in Python is mostly implemented in external libraries.

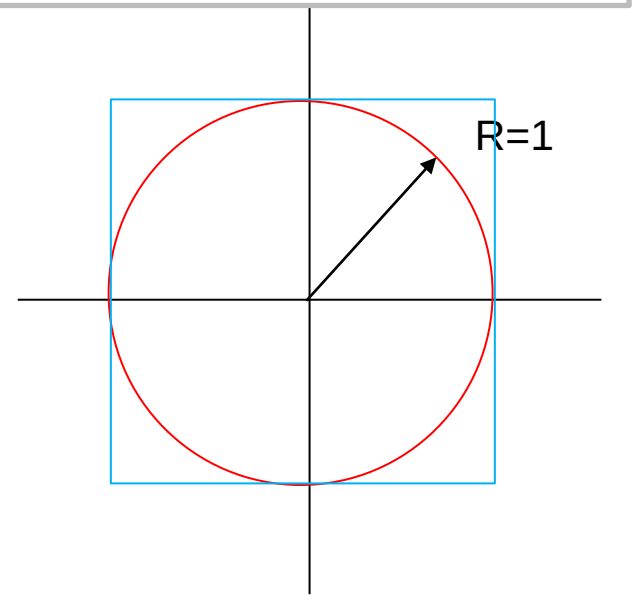
Python Threading

- The Python *threading* library allows for multiple threads to be created.
- Only 1 can actually execute at a time: **do not use this** for CPU-bound problems.
- This works well for I/O-bound problems.
- Each thread runs as soon as it has received data
  - Most of the threads are waiting for data from the disk, the network, the user, etc.
  - Application examples: Python web servers, file servers, network service, calling a web server API…

Python Multiprocessing

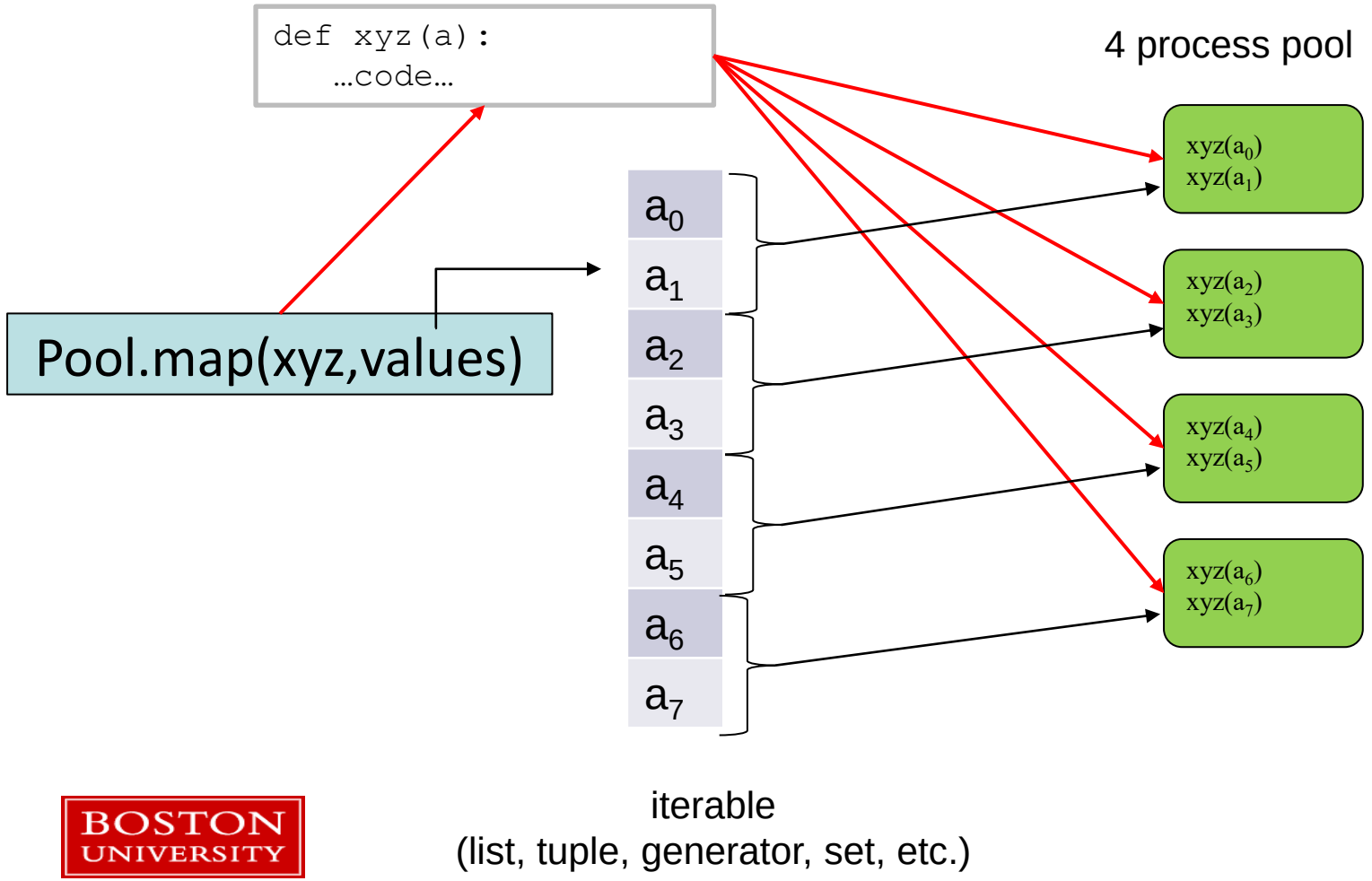
- For CPU-bound problems multiple Python processes can be launched to do computations in parallel.
  - If you just want to parallelize a *for* loop, start here.
- The multiprocessing library handles inter-process communication automatically.
- Most convenient interface: the **Pool**, which provides a set of Python processes that divide work between them.

- Example: *pool_basics.py*



BOSTON UNIVERSITY

How the Pool.map() Works



- A function is pickled and sent to each pool worker.
- The collection of data is split up, pickled, and sent to each worker.
- Each worker unpickles the function & data, runs the function on each element of the collection, pickles the result, and sends it back.
- The main process unpickles the results and puts them into a list.

BOSTON UNIVERSITY

multiprocessing.pool.Pool.map() options

- The Pool is the simplest way to add parallelism to Python code.
- Arguments: `map(function, iterable, chunksize)`

- **function**: the function to be applied to each element of the iterable
- **iterable**: a list, set, generator, dictionary, i.e. something that can be looped over
- **chunksize**: "This method chops the iterable into a number of chunks which it submits to the process pool as separate tasks. The (approximate) size of these chunks can be specified by setting *chunksize* to a positive integer."

Your turn to parallelize a problem…

- Open the file *my_pool.py*
  - The problem: count the characters in 1M English words
  - You'll implement a Pool to parallelize the solution.

Multiple iterables – Pool.starmap()

- To pass multiple arguments use starmap()

```
def xyz(a,b):
        return a+b

vals = [(1,2), (3,4)]

with mp.Pool(processes=2) as pool:
        sums = pool.starmap(xyz,vals)

# 2 function calls happen in parallel:
#    xyz(1,2)
#    xyz(3,4)
```

- If you have 1 object and a list, try this to create a list for starmap:

```
import itertools
a='arg1'
b=range(3)

list(zip(b,itertools.repeat(a)))
# --> [(0, 'arg1'),
#      (1, 'arg1'),
#      (2, 'arg1')]
```

BOSTON UNIVERSITY

Pool.imap() and Pool.imap_unordered()

- *map()* has a disadvantage in that the iterable must be fully **in memory** before it can be distributed.

- *imap()* is lazier. It will assign chunks of work to each worker and pull them as needed from the iterable.
  - Generators can be used to save RAM in the main process.

- *imap_unordered()* is similar but it does not guarantee the output order matches the input order.
  - Good for when computations take a varying amount of time.

imap()

```
def xyz(a,b):
        return a+b

# A generator function
def gen_vals(N):
    for i in range(N):
        # yield evens and odds
        yield 2 * i, 2 * i + 1

with mp.Pool(processes=2) as pool:
        sums = pool.imap(xyz,gen_vals(1000),chunksize = 4)
```

- For pool worker 1, 4 calls to gen_vals() are completed → [(0,1),(2,3),(4,5),(6,7)]
- This list is sent to worker 0.
  - Worker 0 calls xyz(0,1),then xyz(2,3) etc and returns the results in a list to the main Python process.
- Four more calls are done and that list goes to worker 1.
- When worker 0 is completed another 4 calls to gen_vals() are done to create the next chunk, etc.
- The generator *gen_vals()* never creates all 1000 sets of numbers in memory.

BOSTON UNIVERSITY

multiprocessing is quite extensive…

- More functionality exists for the Pool method.
  - Shared memory between workers (avoids copies in interprocess communication)
  - Asynchronous methods – *map_async*, *starmap_async*
    - These let the main process keep running after dispatching work.

- Process control:
  - Launch Python processes, do a calculation, wait for one or more processes to finish.
  - Interprocess communication using Queue and Pipe classes.
  - Synchronization using the Barrier, Lock, and Semaphore classes.

  - This can be used to implement **much** more elaborate parallelization strategies than the Pool at the expense of more programmer labor.

Using map, starmap, imap, imap_unordered

- If:
  - You have function calls being applied to some iterable (e.g. list of data objects, set of files, sets of simulation parameters, etc.)
  - The function call is *computationally expensive* – it takes a while to run.
  - Each function call is independent of the others.
    - Ex. Each input file in a list is read and processed separately.
- Then:
  - The multiprocessing.Pool is worth investigating for your code.
- Else:
  - Try the multiprocessing.Process code. This can be used to build more sophisticated parallelization strategies. Or investigate some other libraries…

Parallelization with External Libraries

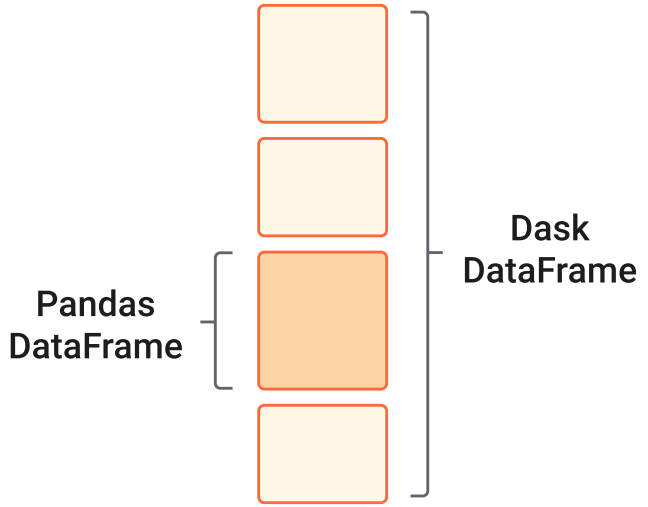
- Python *multiprocessing*: built into Python, works well on a broad array of problems, performs pretty well.

- When to look elsewhere:
  - Your dataset is greater than the amount of RAM you have available
    - You are dealing with large Pandas dataframes, numpy arrays, CSV files, database fetches, etc.
  - You have numpy-centered numeric calculations
    - Ex. A custom image processing algorithm
  - You want to scale past a single compute node
  - *mp* is causing problems due to RAM usage or poor scaling due to its multi-process nature

Parallel Pandas?

- It's possible to do some parallel calculations with Pandas and *multiprocessing* but it's not straightforward.
- The strategy would be to send columns of dataframes to different processes and merge the results.
- This does not work with common Pandas operations like *agg*, *groupby*, *query,* etc.

Dask https://dask.org



- Parallelizing pandas operations can be complex.
  - What if your data is too large to even read into a pandas DataFrame?
- Dask provides an equivalent DataFrame class that natively supports parallel computations.
  - Most pandas code can be handled via Dask just by importing the dask library instead of pandas.
  - Specific tutorial: https://tutorial.dask.org/01_dataframe.html
  - Installed with SCC python3 modules.
- Parallel computations can be run on a single computer or on a cluster using MPI communication.
- Large data sets can be loaded piecemeal to work **within the memory limits** of the computer.

Open *par_pandas_dask.py*

BOSTON UNIVERSITY

pip install (not needed on the SCC)

```
pip install dask distributed dask-jobqueue
```

conda install

```
conda install dask dask-core
```

Dask

- Dask supports parallelism beyond Pandas.

- Dask Array: parallel numpy arrays
  - Includes efficient shared-memory access to these arrays
- Dask Bag: parallelize generic functions like *map* or *groupby* on large collections
  - Example: reformat every line of a CSV file so it can be converted to a DataFrame
- Dask Delayed: parallelize things that don't work with the other approaches.
  - This can be used in place of *multiprocessing* and can be applied to wider variety of programs than a *multiprocessing.Pool*

Polars

- “Polars is a lightning fast DataFrame library/in-memory query engine.”
  - 2-20x faster than Pandas, for many operations
  - Efficiently uses memory and multiple cores
  - This is a relatively recent library, developed at RPI in 2020.

- If you are working with DataFrame style programs and Pandas:
  - Polars benchmarks as significantly faster than Pandas or Dask (which uses Pandas)
  - A conversion from Pandas to Polars is essentially a re-write of your program due to significant differences in syntax

- Parallelize/scale up existing Pandas codebase → try dask
- New or smaller project→ try Polars to see if you like it

Parallelization via Underlying Libraries

- Enabling parallelism in compiled code (C, C++, etc.) libraries that are being used by your Python code is very convenient.
- For many Python codes, this can be sufficient to achieve good parallel speedups without re-writing your code around multiprocessing.
- This is particularly true for codes that make heavy use of Pandas, numpy, and scipy data structures and routines.

Common Parallel Libraries

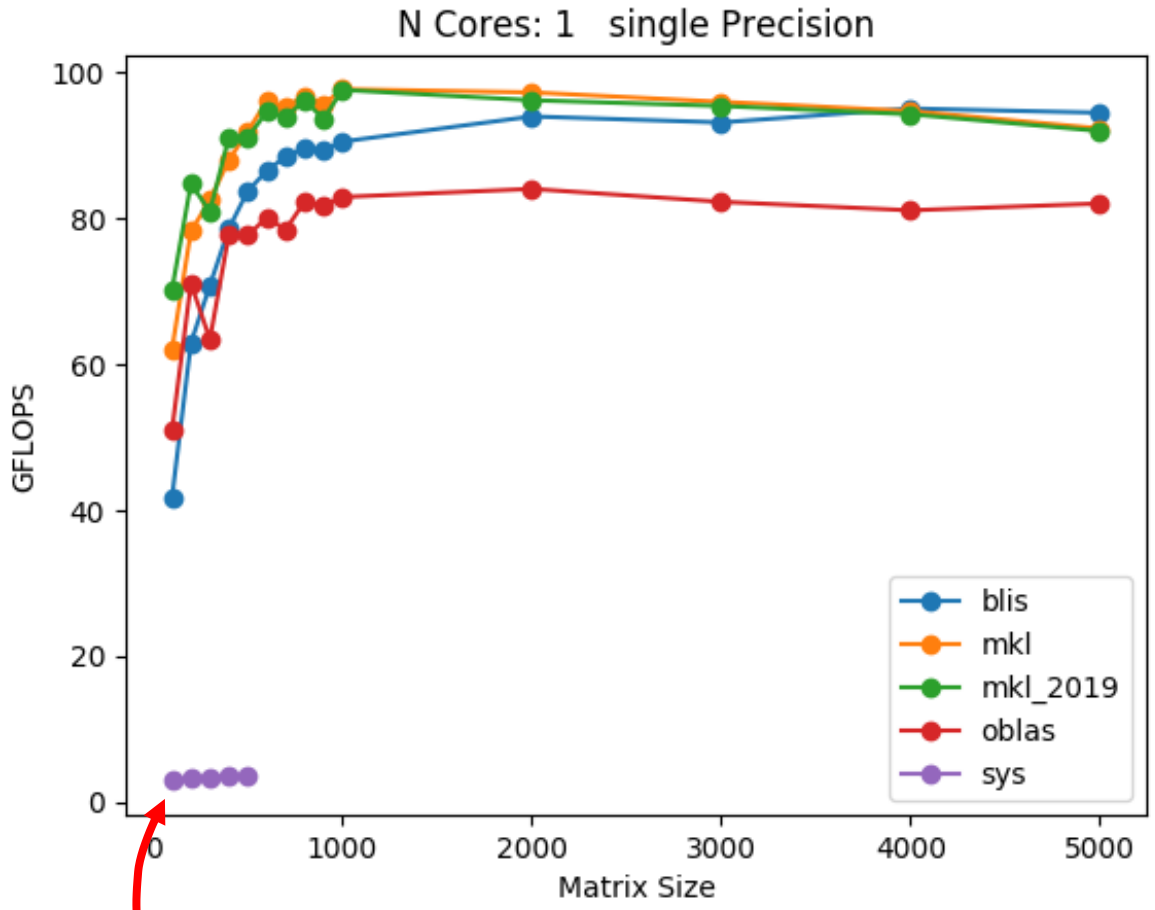
Python Library	Underlying Lib.	Threading Lib.
numpy (scipy, pandas, etc.)	BLAS or MKL	OpenMP or MKL
cv2	OpenCV (C++)	OpenMP or pthreads
Tensorflow, Keras, PyTorch	CUDA or OpenCL	OpenMP or GPU threads
numba	numba C++ libs	Intel TBB
numexpr	numexpr libs	OpenMP

- Using Python for scientific computing naturally leads to the use of several libraries that support parallel computation using multiple threads. Those are built on top of a small set of threading libraries. Lots of other Python libraries use these “behind the scenes”.

- BLAS: Basic Linear Algebra Subprograms
- MKL: Intel Math Kernel Library
- TBB: Intel Thread Building Blocks

BLAS

- The **B**asic **L**inear **A**lgebra **S**ubprograms library provides a variety of functions for linear algebra type calculations.
- This underlies a staggering number of algorithms and computations including much of numpy and scipy.
- High performance threaded BLAS libraries continue to be an active area of computer science research.



- SCC benchmark.
- Note poor performance of default Linux system BLAS library!

BOSTON UNIVERSITY

Numpy BLAS library

Anaconda, Windows

```
In [4]: np.show_config()
blas_mkl_info:
    libraries = ['blas', 'cblas', 'lapack', 'blas', 'cblas', 'lapack']
    library_dirs = ['D:\\bld\\numpy_1595523081734\\_h_env\\Library\\lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['D:\\bld\\numpy_1595523081734\\_h_env\\Library\\include']
blas_opt_info:
    libraries = ['blas', 'cblas', 'lapack', 'blas', 'cblas', 'lapack', 'blas', 'cblas', 'lapack']
    library_dirs = ['D:\\bld\\numpy_1595523081734\\_h_env\\Library\\lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['D:\\bld\\numpy_1595523081734\\_h_env\\Library\\include']
lapack_mkl_info:
    libraries = ['blas', 'cblas', 'lapack', 'blas', 'cblas', 'lapack']
    library_dirs = ['D:\\bld\\numpy_1595523081734\\_h_env\\Library\\lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['D:\\bld\\numpy_1595523081734\\_h_env\\Library\\include']
lapack_opt_info:
    libraries = ['blas', 'cblas', 'lapack', 'blas', 'cblas', 'lapack', 'blas', 'cblas', 'lapack']
    library_dirs = ['D:\\bld\\numpy_1595523081734\\_h_env\\Library\\lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['D:\\bld\\numpy_1595523081734\\_h_env\\Library\\include']

In [5]:
```

- You can see the exact libraries that Numpy is using with the command. The output will depend on the Python installation:

```
numpy.show_config()
```

BOSTON UNIVERSITY

python3/3.10.12 module on SCC

```
>>> np.show_config()
blas_armpl_info:
  NOT AVAILABLE
blas_mkl_info:
  NOT AVAILABLE
blis_info:
    libraries = ['blis', 'blis']
    library_dirs = ['/share/pkg.8/blis/0.9.0/install/lib']
    define_macros = [('HAVE_CBLAS', None)]
    include_dirs = ['/share/pkg.8/blis/0.9.0/install/include/blis']
    language = c
    runtime_library_dirs = ['/share/pkg.8/blis/0.9.0/install/lib']
blas_opt_info:
    libraries = ['blis', 'blis']
    library_dirs = ['/share/pkg.8/blis/0.9.0/install/lib']
    define_macros = [('HAVE_CBLAS', None)]
    include_dirs = ['/share/pkg.8/blis/0.9.0/install/include/blis']
    language = c
    runtime_library_dirs = ['/share/pkg.8/blis/0.9.0/install/lib']
lapack_armpl_info:
  NOT AVAILABLE
lapack_mkl_info:
  NOT AVAILABLE
openblas_lapack_info:
  NOT AVAILABLE
openblas_clapack_info:
  NOT AVAILABLE
flame_info:
  NOT AVAILABLE
accelerate_info:
  NOT AVAILABLE
lapack_info:
    libraries = ['lapack', 'lapack']
    library_dirs = ['/share/pkg.8/blis/0.9.0/install/lib']
    language = f77
    runtime_library_dirs = ['/share/pkg.8/blis/0.9.0/install/lib']
    extra_link_args = ['-L/share/pkg.8/blis/0.9.0/install/lib', '-llapack']
lapack_opt_info:
    libraries = ['lapack', 'lapack', 'blis', 'blis']
    library_dirs = ['/share/pkg.8/blis/0.9.0/install/lib']
    language = c
    runtime_library_dirs = ['/share/pkg.8/blis/0.9.0/install/lib']
    extra_link_args = ['-L/share/pkg.8/blis/0.9.0/install/lib', '-llapack']
    define_macros = [('HAVE_CBLAS', None), ('NO_ATLAS_INFO', 1)]
    include_dirs = ['/share/pkg.8/blis/0.9.0/install/include/blis']
Supported SIMD extensions in this NumPy install:
    baseline = SSE,SSE2,SSE3,SSSE3,SSE41,POPCNT,SSE42,AVX
    found = F16C,FMA3,AVX2,AVX512F,AVX512CD,AVX512_SKX,AVX512_CLX
    not found = AVX512_CNL,AVX512_ICL
```

Enabling Threaded Libraries on the SCC

- Many libraries on the SCC that use multiple cores are built on the OpenMP or MKL threading libraries.
- The SCC disables this threading by default when you load Python or miniconda modules by setting environment variables.
  - Why? Because most jobs are single-threaded, and automatic threading leads to jobs using more cores than they should…and then the jobs are killed by the process reaper.
- In a compute job or at the command line you can enable these threads and they will automatically be used.

BOSTON UNIVERSITY

Threading Environment Variables on the SCC

Variable	Threading Library
OMP_NUM_THREADS	OpenMP, MKL, numexpr
MKL_NUM_THREADS	MKL
NUMBA_NUM_THREADS	numba
NUMEXPR_NUM_THREADS	numexpr

- Setting these variables to a value >1 will enable automatic threading for code that uses the matching threading library.
- These should be set **before** running Python.
- Some libraries have their own internal mechanism can be used in place of the variable.
  - OpenCV example: `cv2.setNumThreads(integer_val)`

On Macs with the Accelerate library use VECLIB_MAXIMUM_THREADS instead of OMP_NUM_THREADS

Enable OpenMP Threading in a Job

- Request a multi-core job:
  - qrsh -pe omp 4
- SCC jobs automatically set the variable NSLOTS to the number of requested cores.
- Environment variables can be set in various ways on different operating systems. Here is a guide for Windows, Linux, and Mac OSX.

Example qsub script:

```
#!/bin/bash -l
# Ask for 4 cores.
#$ -pe omp 4

module load python3/3.10.5

# This sets the number of
# allowed threads to 4.
export OMP_NUM_THREADS=$NSLOTS

# Run your Python script:
python myscript.py

#....did it run faster?
```

BOSTON UNIVERSITY

numba

- numba: auto-compiler for Python code.
  - Can compile code for GPU execution.
- Supports auto-parallelization. Their *prange* function creates a parallelized loop.
- This lets you do low-level threading via Python.
- Thread control variable:

  NUMBA_NUM_THREADS

- Numba can also compile Python code so it is callable from C or C++.
- Read the User Manual and the Reference Manual
- Check out the assortment of environment variables that can be set to influence Numba behavior.

BOSTON UNIVERSITY

numba usage

- Use the decorators *@numba.jit* or *@numba.njit*
- There are 2 modes:
  - object: Python types are used. numba must call out to Python to retrieve values.
  - nopython – no Python types are used, numba accesses values directly.
    - This is faster. Try to do this.

```
@numba.njit(parallel=True, fastmath=True)
def numba_jit_loop(mat):
    ''' A parallel double for loop over
        a 2D numpy ndarray '''
    rows,cols = mat.shape
    for i in numba.prange(rows):
        for j in numba.prange(cols):
            mat[i,j] = 2.0 * mat[i,j] - 1.0
```

- @numba.jit(nopython=True)
- @numba.njit
  - These force nopython mode.

- fastmath=True: allows the compiler to use special CPU instructions.

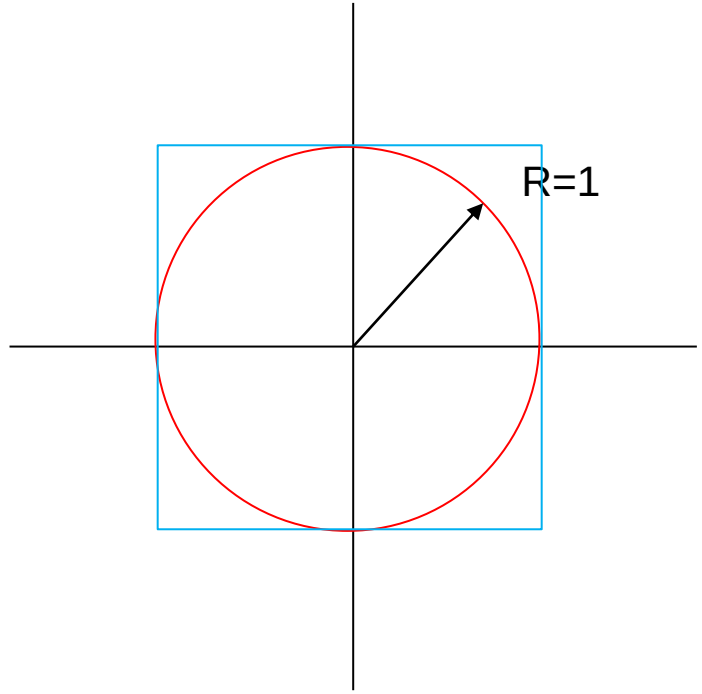
BOSTON UNIVERSITY

open *numba_par.py*

numba usage

- In general, use numpy ndarrays and functions with numba for the best performance.
  - Avoid calls to Python functions and sub-libraries
- numba'd functions should only call other numba'd functions
- This is a large library – test, profile, read the docs!

Let's calculate π with Python and numba



Open *numba_pi.py*

BOSTON UNIVERSITY

numba

- Profiling is necessary with numba. Make sure numba provides a speedup before trying it in parallel.
- Open *numba_par.py* for some examples of applying numba.
- Then we'll look at *numba_convert.py* to see how an existing function might be converted to run faster under numba.

When is this useful?

- If your Python code heavily uses numpy data structures then it **may** benefit from automatic threading or compilation from *numba.*

- **Read the Numba docs.**
  - Numba is under continuous rapid development - new features appear all the time.

- Eperiment! more threads is not always better.
  - The overhead of launching threads and distributing work can easily exceed the parallel execution speedup.

End-of-course Evaluation Form

- Please visit this page and fill in the evaluation form for this course.
- Your feedback is highly valuable to the RCS team for the improvement and development of tutorials.
- If you visit this link later please make sure to select the correct tutorial – name, time, and location.

http://scv.bu.edu/survey/tutorial_evaluation.html

BOSTON UNIVERSITY