

Python Parallelization

Summer 2026

Research Computing Services
IS & T

Download files:

<https://scv.bu.edu/examples/python/tutorials/PythonPar/>



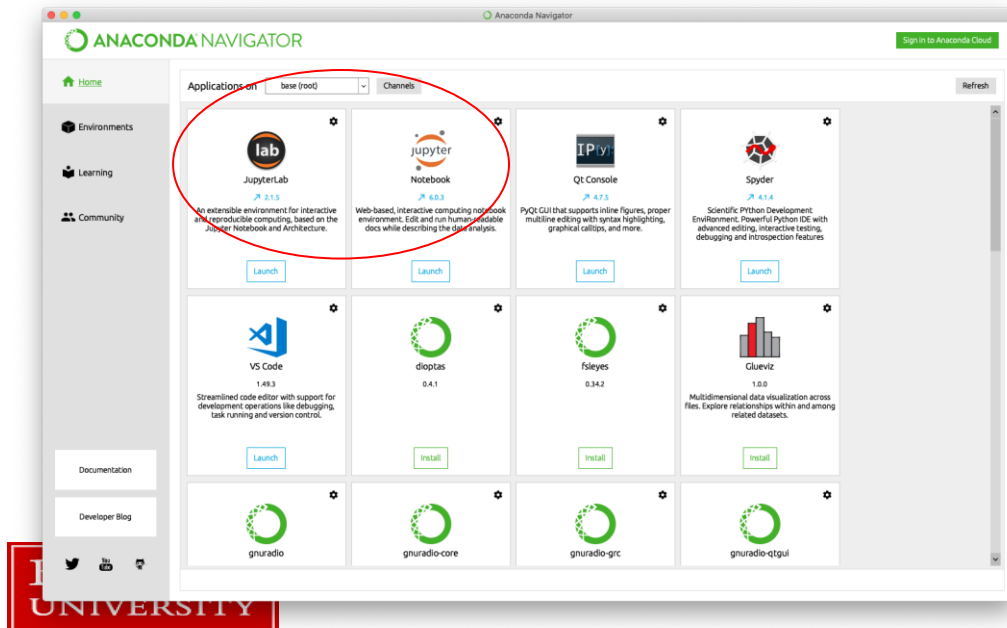
End-of-course Evaluation Form

- Please visit this page at the end of the tutorial 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

Start Jupyter

- Your own computer:
 - Start Anaconda Navigator
 - Find Jupyter Notebook or Jupyter Lab and launch it.



- On SCC OnDemand:
 - Under Interactive Apps choose *Jupyter*.
 - Load **python3/3.13.8** and select **4** cores, then click the Launch button.

Jupyter Notebook
This app will launch a Jupyter Notebook server on a compute node.

Use saved settings
-- select saved settings --

List of modules to load (space separated)
python3/3.13.8 Select Modules

Pre-Launch Command (optional)

Interface
lab

Working Directory
 Select Directory

The directory to start Jupyter in. (Defaults to home directory.)

Extra Jupyter Arguments (optional)

Number of hours
4

Number of cores
4

Download Files

- <https://scv.bu.edu/examples/python/tutorials/PythonPar/>
- Get *py_par_summer_2026.zip*
- (Everyone) - On your computer:
 - Unzip to a convenient folder
 - Windows – right-click on the .zip file → Extract All
- SCC users also do:
 - Open a terminal window in Jupyter
 - Type into the terminal and press Enter:

```
/net/sccl/scratch/get_pypar.sh
```
 - This will unpack to ~/PythonPar

Introduction

- Many programs can perform simultaneous operations, given multiple processors to perform the work.
- The burden of managing this lies on you, the programmer.
- In this tutorial we'll go over a variety of ways to achieve parallelism in Python code.
 - 1) parallelizing your code directly with the *multiprocessing*, *joblib*, and *numba* libraries.
 - 2) finding external libraries that bring parallelization to your 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...
- Parallel Libraries
 - Use a library that internally implements some kind of parallelization.
- 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.

Performance Considerations

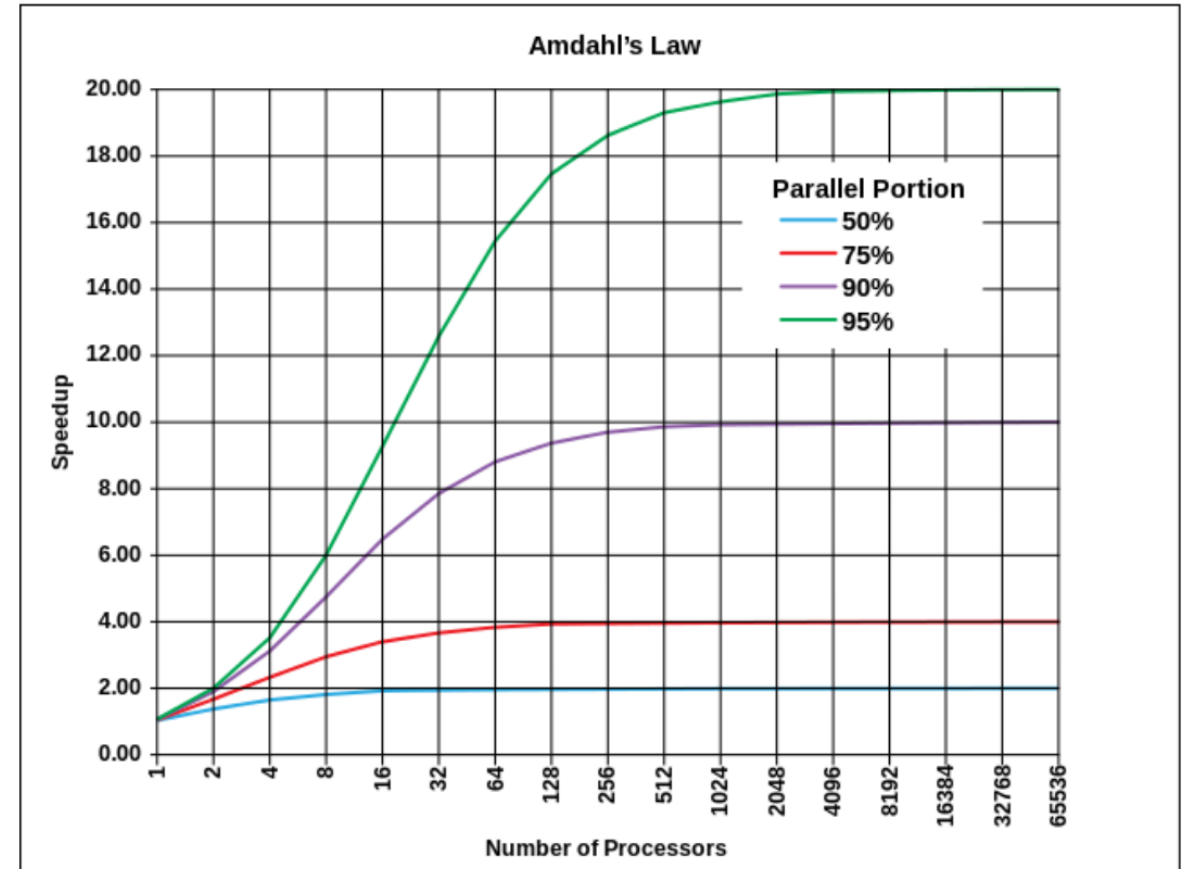
- Not every part of a program can benefit from parallelization.
- Some parts of program are inherently serial.
- Even for a function that can be done in parallel...
 - Is it worth the programming effort?
 - Is it worth the reduction in readability and ability to debug?
 - Does the function use up enough 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](#).

How many cores should Python use?

- The example notebook *get_n_cores.ipynb* provides a function that checks how many cores have been assigned to an SCC job.
- It will also work on your own computers and will return the number of available cores.
 - Based on the common Python library *psutil*
 - Note that it can't differentiate between “performance” and “efficiency” cores.
- Feel free to use this in your own code.

Python Language Parallelism

- Python provides a number of ways to perform parallel (aka concurrent) computations as part of its standard library.
- Read the [official docs](#).

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

The Global Interpreter Lock

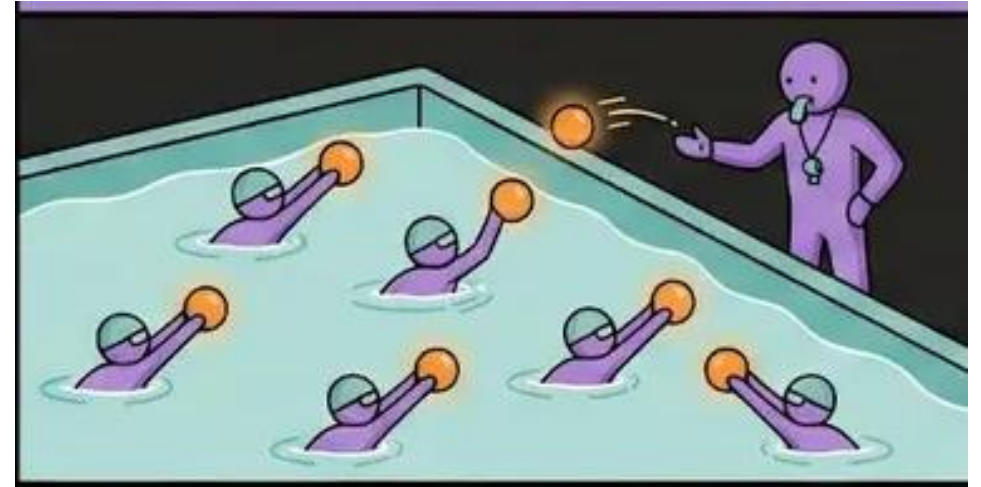
- The GIL limits the amount of multi-threading in the Python interpreter.
 - Originally introduced as part of Python's memory management system.
 - For more details, see [this explanation](#).
- Pure Python code runs in one thread only.
 - This is unlike languages like Java, C#, C++, Fortran, Matlab, or R where threads are easily used by the programmer.
- Multi-threaded code in Python is mostly implemented in external libraries.
 - The recently released Python 3.14 officially supports a multi-threaded interpreter.
 - It will take a long time before this is preferred over 3.14+GIL due to compatibility issues with 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

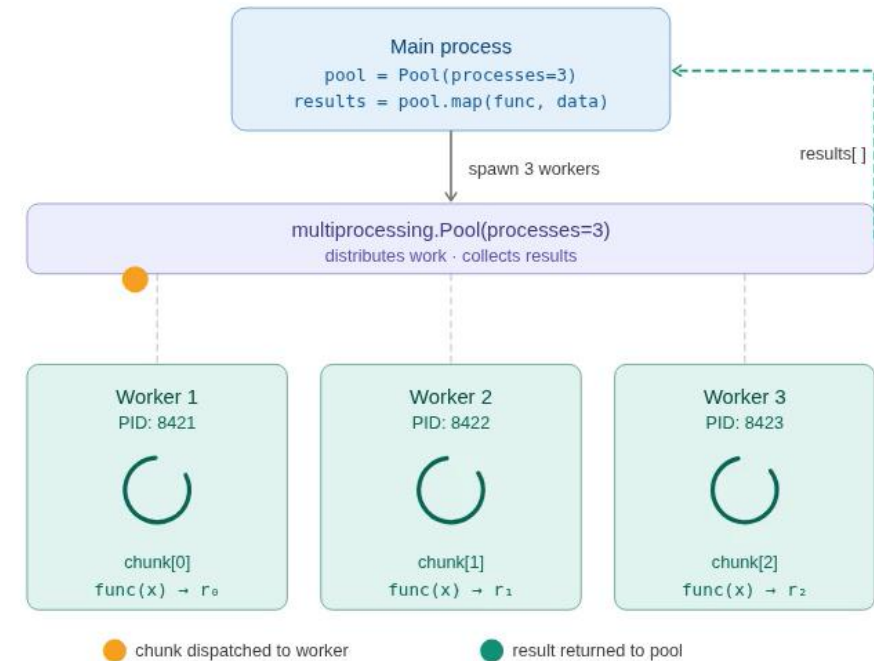
- For CPU-bound problems multiple Python processes can be launched to do computations in parallel.
 - If you just want to parallelize a *for* loop, start here.
- The multiprocessing library handles inter-process communication.



- Most convenient interface: the **Pool**, which provides a set of Python processes that divide work between them.

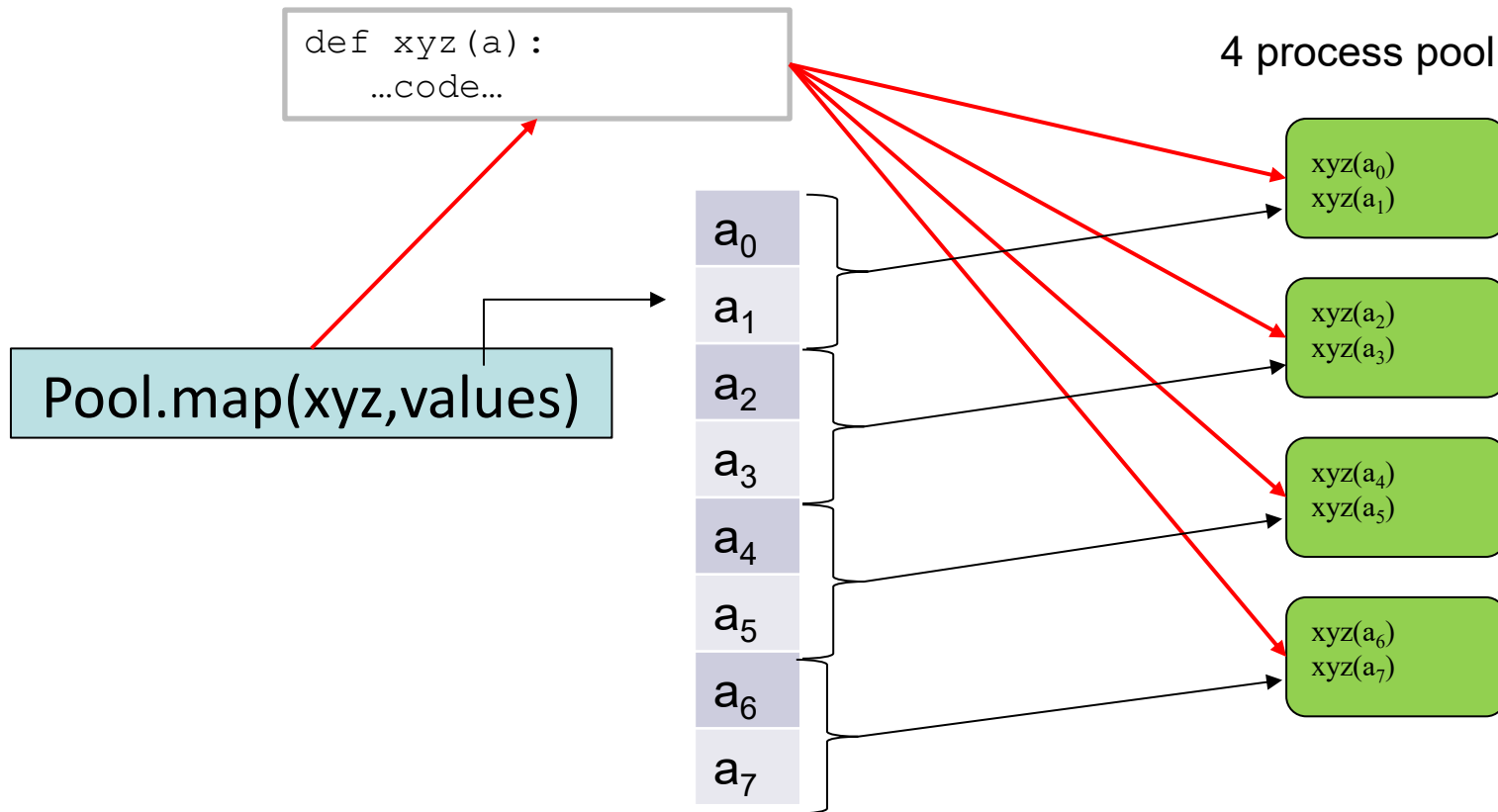
Convert a loop

```
def xyz(x):  
    ''' here we do something that takes  
        some time OR gets called a lot  
        of times '''  
    # this is a regular function,  
    # nothing special  
    result = etc...  
    return result  
  
# somewhere else in your program:  
results = []  
# This loop takes a long time to  
# run:  
for elem in big_list:  
    results.append(xyz(elem))
```



```
import multiprocessing as mp  
  
# Replace the loop with a Pool and a map  
with mp.Pool(processes=nprocs) as pool:  
    results = pool.map(xyz, big_list)  
  
# xyz() now runs in parallel over the  
# elements of big_list.
```

How the Pool.map() Works

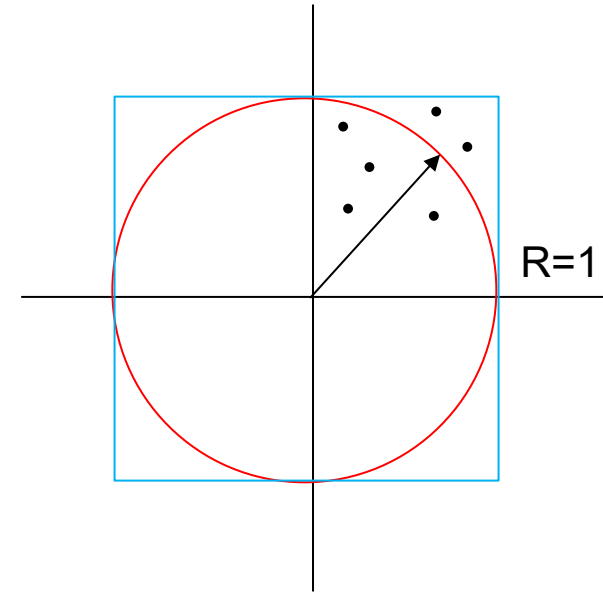


values is an iterable
(list, generator, set, numpy vector, etc.)

- 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 data is evenly distributed across the workers as much as possible.
- The main process unpickles the results and puts them into a list.


Example

- Open *pool_basics.ipynb*
- This calculates the value of π




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:



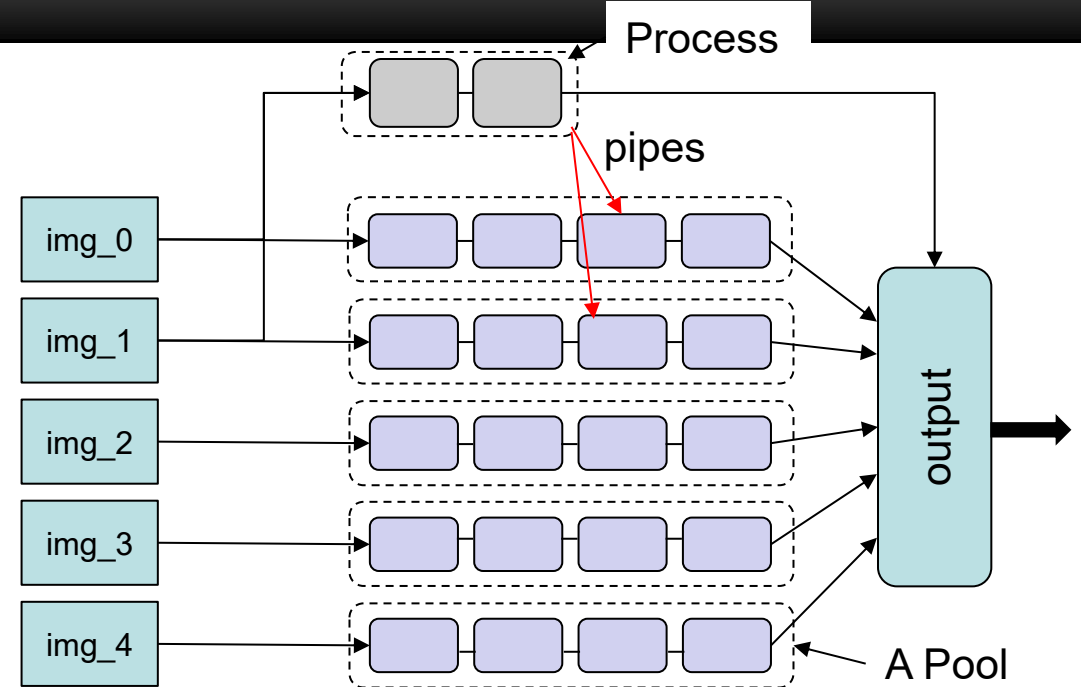
```
import itertools  
a='arg1'  
b=range(3)  
  
list(zip(b,itertools.repeat(a)))  
# --> [(0, 'arg1'),  
#      (1, 'arg1'),  
#      (2, 'arg1')]
```

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.

More complex algorithms

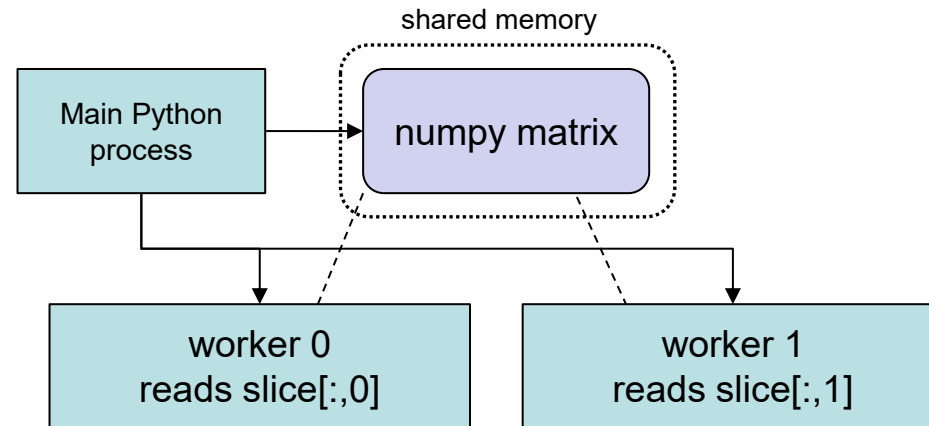
- multiprocessing.Pool applies easily to for-loop parallelization. What about more complex patterns?
- Other tools in [multiprocessing](#):
 - Start & stop processes that execute functions.
 - pipes, queues → send data between processes
 - Shared memory → processes access data without it being copied to them
 - locks, semaphores → protect serial-only resources from parallel access
 - Writing to a common file, updating shared memory, etc.
 - See sample files in code/extra:
`procs_and_pipes.py`,
`recursive_multiprocessing.py`,
`producer_consumer.py`



A pipeline where multiple images go through a series of filters. 2 images get copied to a separate set of filters. The *output* stage aggregates these results.

- For extensive tutorials on the many ways to use the multiprocessing library see:
<https://superfastpython.com/tutorial-archive.html#python-multiprocessing>

joblib



- [joblib](#) is a library for simple, efficient parallel computing in Python.
 - Features:
 - Easy parallelization of *for* loops and other common parallel scenarios
 - **Intended for large data** – easily handles sharing of large data structures between parallel processes
 - [Memoization](#) – recognizes and avoids the re-computation of previous function calls
 - Fast writing/reading Python structures from a file.
 - Avoids awkwardness on Windows for parallel computations.

Re-visit parallel π

- Open notebook *joblib_pi.ipynb*
- Let's see how this compares with the multiprocessing solution.

Your turn to parallelize a problem...

- Open the file *my_pool.ipynb*
 - The problem: count the characters in 1M English words
 - You'll use *joblib* to parallelize the solution.



- [numba](#): auto-compiler for Python code.
- Supports [auto-parallelization](#). Their *prange* function creates a parallelized loop.
- This lets you create multithreaded Python code.
 - No inter-process communication.
 - Fast startup/shutdown of threads
- Thread control environment 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.
- Numba can compile a Python function so it executes on an Nvidia GPU.

Basic numba usage

- Decorate your function:

- `@numba.jit`
 - “object mode”
 - Compiles code and allows for non-compilable Python functions or data types to be used. Will call back out to the Python interpreter when needed.
- `@numba.njit`
 - “nopython mode”
 - Numba will only work with types that it can compile.
 - This is **much faster**. Try to do this.

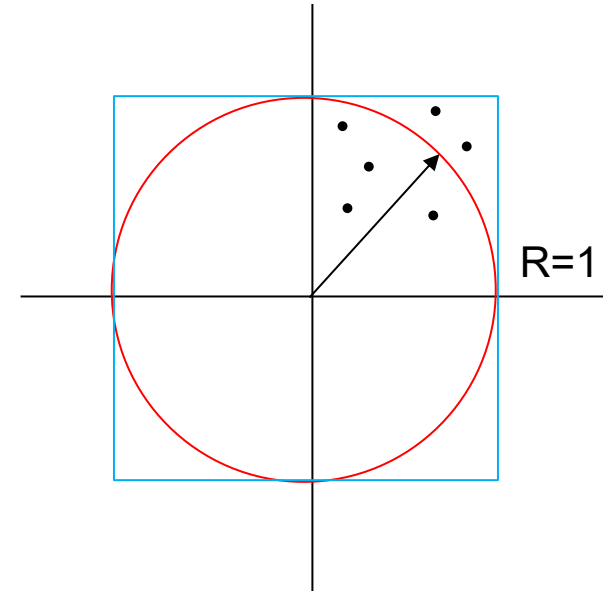
```
@numba.njit(parallel=True, fastmath=True, cache=True)
def numba_jit_loop(mat):
    ''' A parallel double for loop over
        a 2D numpy ndarray '''
    rows, cols = mat.shape
    for i in numba.prange(rows):
        for j in range(cols):
            mat[i,j] = 2.0 * mat[i,j] - 1.0
    return mat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

- Options:
 - `fastmath=True`: allows the compiler to use special CPU instructions.
 - `cache=True`: after compiling, stash a copy for the next time the program runs.
 - `parallel=True`: use multiple threads
- `numba.prange()` → parallel version of the `range()` function.
- numba can also [auto-parallelize](#).

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!

Once again, let's calculate π with Python and numba



Open *numba_pi.py*

When is this useful?

- If your Python code heavily uses numpy data structures then it **may** benefit from automatic threading or compilation from *numba*.
 - Or, if your code has a lot of “for” loops over lists or other iterables
- *numba* has been implementing a growing number of Python data types, [see their docs](#) for the latest.
 - Now supported: homogeneous lists and sets, typed dictionaries
- **Read the Numba docs.**
 - Numba is under continuous rapid development - new features appear all the time.
- Experiment! more threads is not always better.
 - The overhead of launching threads and distributing work can easily exceed the parallel execution speedup for small problems.

```
from numba import njit
import numpy as np

@njit
def foo():
    d = dict()
    k = {1.0: np.arange(1), 2.0: np.arange(2)}
    # The following tells the compiler what
    # the key type and the value type are for `d`.
    d[3] = np.arange(3)
    d[5] = np.arange(5)
    return d, k

d, k = foo()
print(d)      # {3: [0 1 2], 5: [0 1 2 3 4]}
print(k)      # {1: [0], 2: [0 1]}
```

Dictionaries must have a uniform type for keys
and a uniform type for values:

```
d = {type_1:type_2, type_1:type_2,...}
```

Parallelization with External Libraries

- When to look beyond *joblib*, *multiprocessing*, and *numba*?

Your dataset is greater than the amount of RAM you have available

- You are dealing with large Pandas dataframes, numpy arrays, CSV files, database queries, etc.

You have numpy-centered numeric calculations

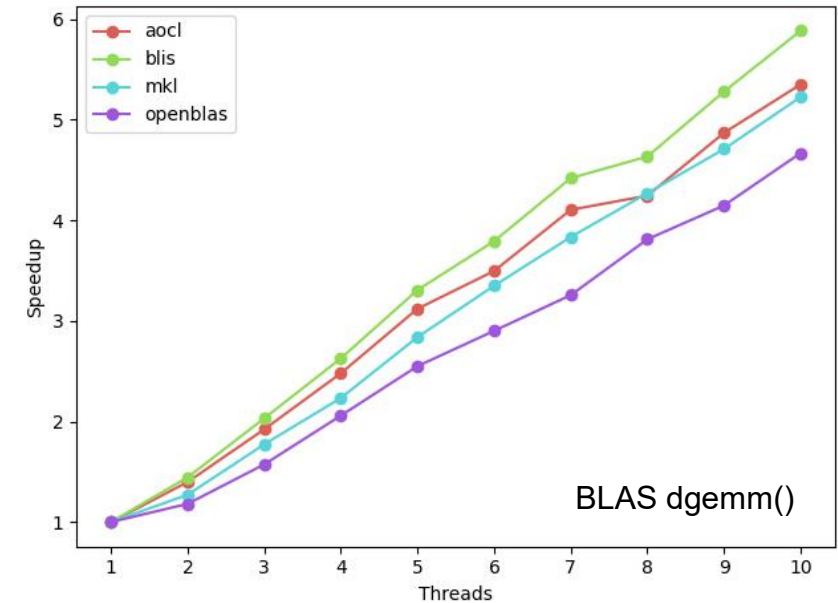
- Ex. A custom image processing algorithm based on numpy functions, not basic numpy statements.

You want to scale past a single compute node

You want to implement more complex parallel algorithms

A Multithreaded Foundation

- The foundation of numeric computing in Python is the numpy library.
- Many numpy operations are implemented as calls to the BLAS and LAPACK libraries.
- BLAS: “Basic Linear Algebra Subroutines”
- LAPACK: “Linear Algebra Package”
- BLAS/LAPACK optimization is an area of active research.
 - Very high performance libraries are available, and usually they can **automatically multithread**.



- BLAS:
 - Vectors, dot products, norms.
 - Vector-matrix multiplication, triangular solvers
 - Matrix-matrix multiplication
- LAPACK:
 - Least squares
 - Eigenvalues
 - Singular value decomposition
 - Matrix factorization

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']
```

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

`numpy.show_config()`

python3/3.13.8 module on SCC

```
>>> np.show_config()
Build Dependencies:
blas:
  detection method: pkgconfig
  found: true
  include directory: /share/pkg.8/flexiblas/3.4.5/install/include/flexiblas
  lib directory: /share/pkg.8/flexiblas/3.4.5/install/lib64
  name: flexiblas
  openblas configuration: unknown
  pc file directory: /share/pkg.8/flexiblas/3.4.5/install/lib64/pkgconfig
  version: 3.4.5
lapack:
  detection method: pkgconfig
  found: true
  include directory: /share/pkg.8/flexiblas/3.4.5/install/include/flexiblas
  lib directory: /share/pkg.8/flexiblas/3.4.5/install/lib64
  name: flexiblas
  openblas configuration: unknown
  pc file directory: /share/pkg.8/flexiblas/3.4.5/install/lib64/pkgconfig
  version: 3.4.5
```

- python3/3.13.8 uses *flexiblas*, which lets you choose which BLAS library you want to use.
- For more info on the SCC run:

`module help python3/3.13.8`

`module help flexiblas/3.4.5`

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.

Threading Environment Variables on the SCC

Variable	Threading Library
OMP_NUM_THREADS	OpenMP, MKL
OPENBLAS_NUM_THREADS	OpenBLAS, if you installed numpy into conda or virtual environments
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 way to be used in place of the variable.
 - OpenCV example: `cv2.setNumThreads(ncores)`

Enable OpenMP Threading in a Job

- 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 python3/3.13.8

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

# Run your Python script, as
# this uses a lot of numpy code
# and might benefit from threads:
python myscript.py

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

mpi4py



- If you are working on problems that require:
 - Exacting control over complex parallel algorithms running across multiple processes
 - (maybe) multiple compute nodes to run
 - The sky's the limit – MPI is the standard HPC multi-node parallelization library, it will scale to thousands of compute nodes.
- ...then the mpi4py library might be what you need.
- We have a [tutorial](#) on programming with the MPI (Message Passing Interface) library on June 11.
 - It won't cover mpi4py directly, but anything you learn there will greatly help you in Python!

Pandas in Parallel

- Modin
 - Implements ~90% of the Pandas DataFrame API.
 - Autoscales dataframe calculations onto available cores.
 - Part of the Ray library, can use multiple nodes.
- cudf
 - Run Pandas on Nvidia GPUs. From the rapids.ai collection.
- Dask DataFrames
 - Can autoscale and use available cores.
 - Can use multiple compute nodes.
 - Built on top of Pandas.
- “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 in 2020.
 - This is **not** compatible with your existing Pandas code. Using it requires a re-write of your code.
 - See their [migration guide](#).

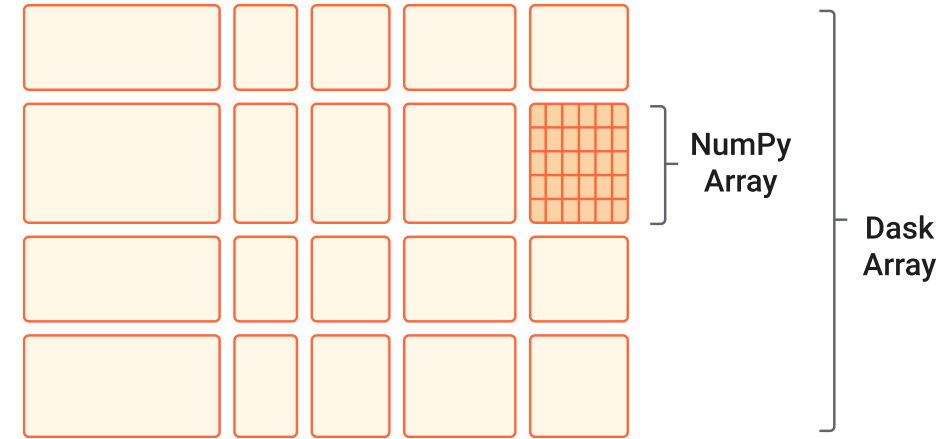


RAPIDS



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: read a file where each line is a JSON string. Convert to a format that can be converted to a Dask DataFrame.
- [Dask Delayed](#): parallelize things that don't work with the other approaches.
 - For example, the image processing graph from a few slides ago.
- RCS is offering a Dask tutorial on June 10.



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