

# Python with Dask

Research Computing Services  
IS&T

# Dask: “Scale the Python tools that you love”

- [Dask](#) is an open source Python library for parallel computing.
- 3 high level APIs:
  - *DataFrame* – parallel Pandas DataFrames
  - *Array* – parallel Numpy arrays (multidimensional numeric arrays)
  - *Bag* – parallel Python collections (lists, sets, etc)
- Low-level APIs:
  - *Delayed* and *Futures* – parallel Python functions, loops, and more
- Parallelization can scale from using all the cores on your laptop to using whole clusters with hundreds of cores.

# Installation

- See the Dask [install instructions](#)
- Conda environments:
  - `conda install dask -c conda-forge`
- Python virtual envs:
  - `python -m pip install "dask[complete]"`
- Anaconda: Open Navigator, choose *Environments* on the left column.
  - Next to the *base* environment, open a Terminal or Command window.
  - Enter: `conda install dask -c conda-forge`
- SCC: already part of recent *python3* and *academic-ml* modules.

# SCC Tutorial using Jupyter Notebooks

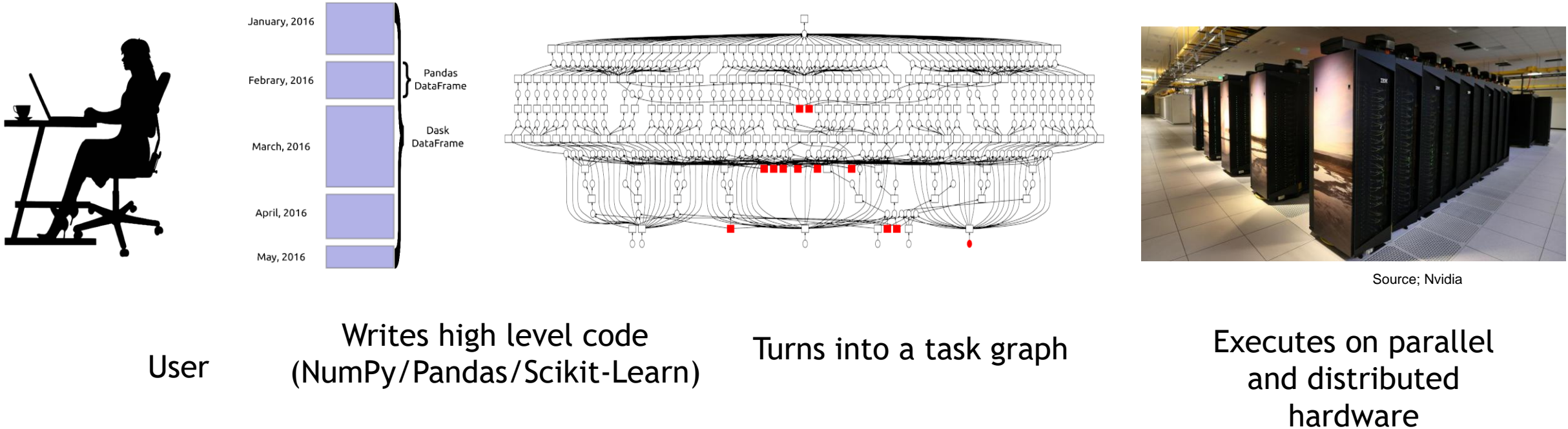
- Open a web browser and go to: `scc-ondemand.bu.edu`
  - Create a Jupyter Notebook session:
    - Use *python3/3.12.4* module.
    - Choose the *lab* Interface.
    - Leave the Working Directory blank.
    - Request **4** cores
    - Set the number of hours to 3.
  - In the Login Nodes menu, pick a login node, and in the command line enter:
    - **`/net/scc1/scratch/setup_dask_tutorial.sh`**
  - Open the Jupyter session in OnDemand and you're ready to go.

# First, some Pandas

- Open the *pandas\_bikes.ipynb* notebook.
- This pulls [ride data](#) from the Bluebikes bike share program.
- It calculates some simple quantities.
- 2011 trip data: ~11MB, ~140k rows.



# Dask basics

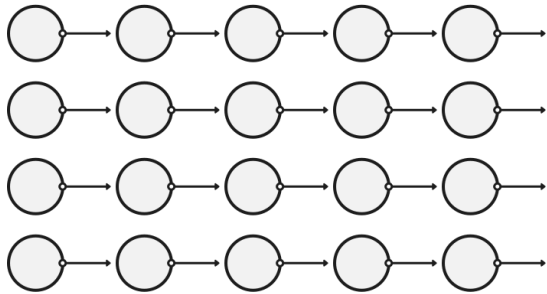


- Using Dask you can develop a solution on your laptop using a subset of your data.
- You can then deploy that exact code onto systems and use potentially thousands of CPUs to process the full data set.

# Dask Task Graph

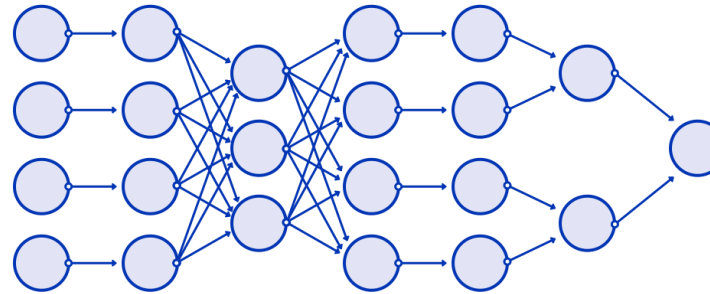
## Embarrassingly Parallel

Hadoop/Spark/Dask/Airflow/Prefect



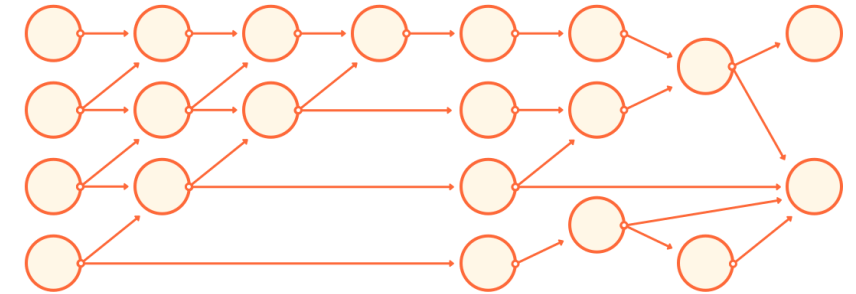
## MapReduce

Hadoop/Spark/Dask



## Full Task Scheduling

Dask/Airflow/Prefect



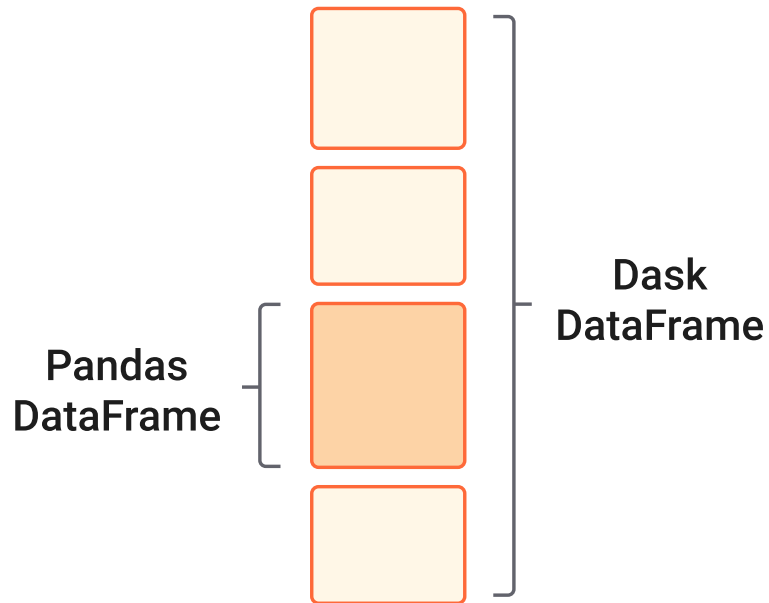
- The [task graph](#) is the series of computations that solve a given problem.
- In Dask you set up a computation then call a *compute()* method on a Dask object to trigger the actual computing.
  - Other methods that trigger computations: *persist()* and *take()*
- Dask will optimize the process of reading data using multiple processors, work within memory limits, store intermediate results, etc. to produce the final product.

# Dask Data Structures

- DataFrame – based on Pandas DataFrames.
- Array – based on Numpy arrays
- Bag – based on Python collections, like a Python list.



# DataFrame



Executes  
the task  
graph

## Dask DataFrame API

Reads ALL CSVs

```
>>> import dask.dataframe as dd
>>> df = dd.read_csv('2014-*.csv')
>>> df.head()
   x  y
0  1  a
1  2  b
2  3  c
3  4  a
4  5  b
5  6  c
```

Creates  
a task  
graph

```
>>> df2 = df[df.y == 'a'].x + 1
>>> df2.compute()
0    2
3    5
Name: x, dtype: int64
```

## pandas DataFrame API

Reads ONE CSV

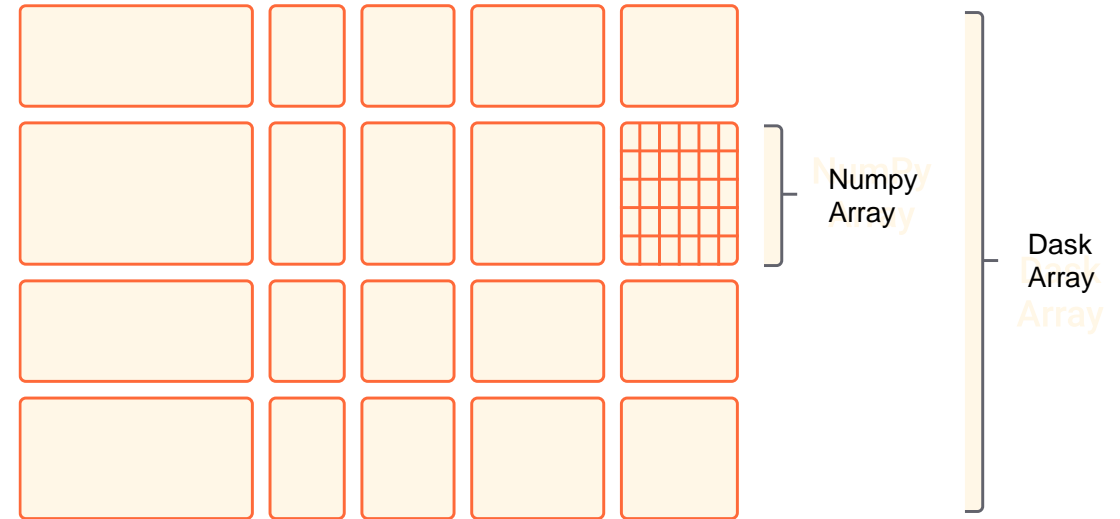
```
>>> import pandas as pd
>>> df = pd.read_csv('2014-1.csv')
>>> df.head()
   x  y
0  1  a
1  2  b
2  3  c
3  4  a
4  5  b
5  6  c
```

```
>>> df2 = df[df.y == 'a'].x + 1
>>> df2
0    2
3    5
Name: x, dtype: int64
```

- [DataFrame docs](#)

- The dask.DataFrame is almost completely compatible with a Pandas DataFrame
- **Dask DataFrames can be spread across multiple cores/computers for parallel computations.**
- You can easily transition your code from plain Pandas to Dask.

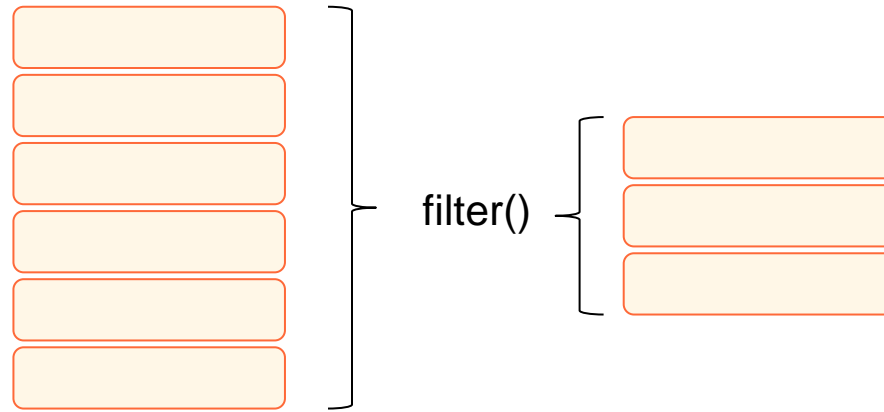
# Array



## ■ Array

- Breaks up a Numpy ndarray into smaller arrays.
- Allows for processing ndarrays that need more memory than is available on the computer.
- Defaults to multithreaded computations that use shared memory to avoid data communication costs.

# Bag



- Bag

- Implements list-style processing with functions like `map()`, `filter()`, `groupby()` on collections of generic Python objects.
- Example:
  - read a set of JSON data files and convert them into Python data structures.
  - Filter out bad data.
  - Reformat data into a DataFrame compatible format
  - Convert to a Dask DataFrame, continue processing.

# Dask-ify the Pandas Notebook

- Now open the notebook *dask\_bikes.ipynb*
- This is a straightforward conversion of the calls to Pandas & Numpy to use equivalent ones with Dask **DataFrames** and **Arrays**.

# Dask Bag Example

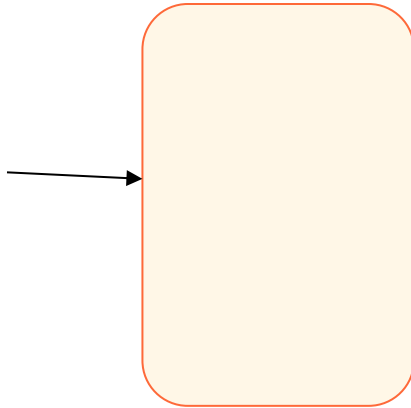
- Use a Bag when the data source does not easily match the Array or DataFrame datatypes.
- It acts like a parallel Python list or tuple
  - However: you cannot index it or use it with 'for' loops.
- Process a bag by applying functions that transform elements like the Python `map()` and `filter()` functions.
- Bag types can be converted to Arrays or DataFrames or saved as files.
- Open *dask\_bag.ipynb* and let's give it a try!

# So far...

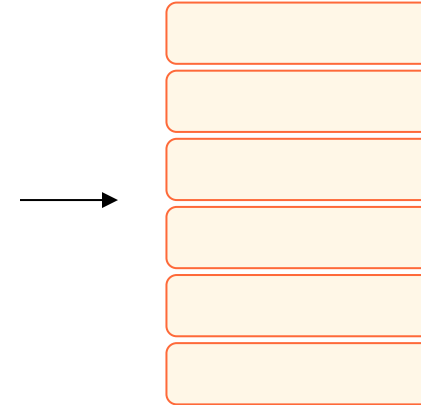
- We've tried the 3 basic data types: Array, DataFrame, Bag.
- The Dask DataFrame was slower than Pandas when analyzing a single file.
  - There is overhead associated with building, optimizing, and executing the task graph.
  - Dask DataFrames add overhead to its use of Pandas so for a single file Pandas is faster.
- Both the Bag and DataFrame made it trivially easy to scale up from single file processing to many file processing.
  - A similar scaling could be shown going from reading a small file to a giant file.

# Partitioning

- Conceptually, a Dask DataFrame or Array is one big thing we manipulate.



- To stay within memory limits and facilitate computing parts of the task graph, it is broken down into pieces called *partitions*



- Partition sizes are auto-computed by default but can be changed using the *repartition()* function.
- If filtering and other transformations have reduced the amount of data you may need to manually repartition.

- A DataFrame is partitioned into mini-Pandas DataFrames.
- A Bag is partitioned into tuples of Bag elements.
  - Reading files, probably one-per-file
  - Reading lines-per-file, into tuples of lines.

- An Array is partitioned (“chunked”) by dimension,
  - E.g. a 3D array of size 1000x200x6
    - 6 1000x200 chunks
    - 40 50x200x3 chunks
    - etc

# DataFrame Partitions and I/O

- Imagine you have a large Dask DataFrame in memory
  - Say, representing 600M rows and 20 columns. It is split into 1000 partitions.
- How can you make a single CSV file from this?
- Let's check the [DataFrame.to\\_csv\(\) docs](#)
- Here's the [list of DataFrame to storage functions](#).

```
# The entire Dask Data Frame
# must go into memory on the
# main processor, be careful!
pandas_df = dask_df.compute()
pandas_df.to_csv('filename.csv')
```

```
dask_df.to_csv('filename.csv')
# each partition gets written
# separately in parallel:
#     filename_01.csv
#     filename_02.csv
#     ...etc...
# Then you have to join them
# together with your own code.
```

```
# Partitions are appended in turn:
dask_df.to_csv('filename.csv',
               single_file=True)
# filename.csv is written
```



# compute() and persist()

- Define a dataframe and do some filtering
- compute() is called twice
- This triggers **two** evaluations of the task graphs, starting from the file reads.

```
import dask.dataframe as dd

# read some CSV files containing
# sales data for some stuff.
df = dd.read_csv('/path/to/*.csv')
# do some cleaning
df = df.dropna()
# ignore low sales counts
df = df[df['units_sold'] > 1000]
# Get some info
med_count = df['units_sold'].median().compute()
print(f'Median sales: {med_count}')

# Group by salesperson
sales_info = df.groupby('salesperson')['units_sold'].\
    sum().compute()
# sales_info --> Pandas dataframe
print(sales_info)
```

- This approach preserves memory at the expense of computational time.

# compute() and persist()

- `.persist()` – evaluate the task graph to this point, then keep everything in memory.
  - Input files are read once.
- The two calls to `compute()` then execute off the same dataframe.
- Dask has other ways to [manipulate](#) the task graph.

```
import dask.dataframe as dd

# read some CSV files containing
# sales data for some stuff.
df = dd.read_csv('/path/to/*.csv')
# do some cleaning
df = df.dropna()
# ignore low sales counts
df = df[df['units_sold'] > 1000]

# Compute the task graph to this
# point and keep it in memory.
df = df.persist()

# Get some info
med_count = df['units_sold'].median().compute()
print(f'Median sales: {med_count}')

# Group by salesperson
sales_info = df.groupby('salesperson')['units_sold'].sum().compute()
# sales_info --> Pandas dataframe
print(sales_info)
```

# Scaling up Dask Task Graphs

- Dask can [automatically parallelize](#) many parts of its task graph.
  - You can choose to parallelize with threads, processes, or a mixture of both.
  - Threads:
    - good choice when work mostly involves Arrays or DataFrames
    - These mostly call into numpy and Pandas functions and don't use many Python data types.
  - Processes:
    - best when using plain Python types (lists, strings, dicts, custom Python classes), Dask Bags, or Dask delayed.
- launches multiple copies of Python that exchange data as needed.

# Parallel Computations

- The notebook *dask\_bikes\_parallel.ipynb* allows Dask to do some auto-parallelization.
  - There's just 1 cell added to the previous notebook.
- This is done by enabling the [Dask Scheduler](#) to use parallel threads, where it is possible to do so.
  - This uses the provided `get_n_cores()` function that works nicely on the SCC.
  - Depending on what you're doing, you may get good performance just using this approach.

```
: import dask
dask.config.set(scheduler='threads', num_workers=get_n_cores())
```

- There is an option to use the 'processes' option to launch multiple processes, but this is not the recommended approach (see distributed slides coming up)

# Dask.distributed

- The Dask.distributed library:
  - Choose threads or processes
  - Limit memory usage
  - Lower-level access to parallel computations
- This should be considered a companion to the main Dask library.
  - Allows for tighter control of parallel computations, greater design flexibility.
- Now revisit our *dask\_bikes\_parallel\_dist.ipynb* notebook.
  - This has been re-worked to properly use Dask.distributed to compute via processes.
  - Note there's a plain Python copy too: *dask\_bikes\_parallel\_dist.py*

# Highly Parallel Dask Computations

- Once you've got a Dask-based computation to run on the SCC, you can use lots of cores, try requesting 4, 8, 16, 28, or 32.
- These are best submitted as non-interactive batch jobs to avoid long waits in OnDemand.
- Use the `if __name__ == '__main__':` section to make sure that Dask.distributed launches correctly.
- Experiment with processes vs. threads for performance:
  - 1 thread, 16 processes
  - 2 threads, 8 processes
  - Etc.

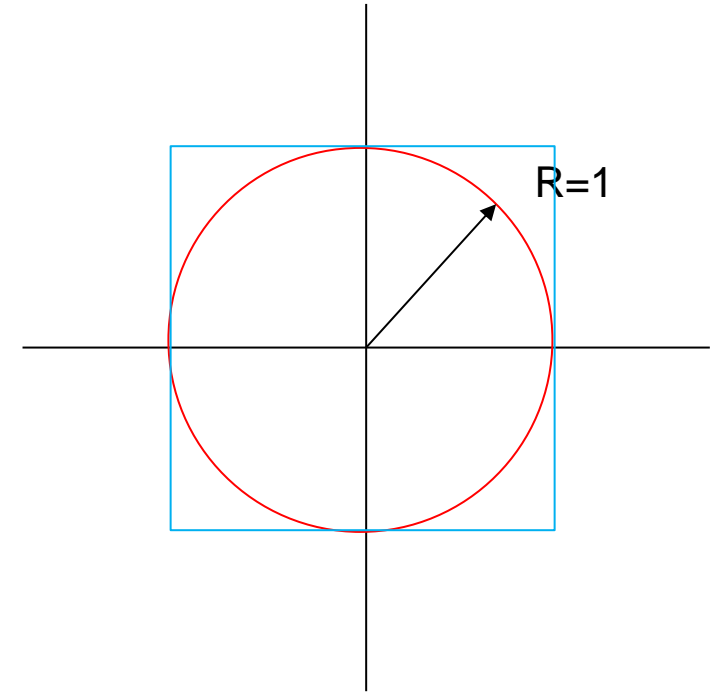
# Multi-node Dask

- This uses the [Dask-MPI](#) library.
- If you have a Dask-based calculation that requires multiple nodes on the SCC, contact RCS and we'll help you get this set up.

# Dask.delayed

- Parallelization tool for any Python function.
- This is the tool when you want to parallelize Python function calls
- Pick N random points in the upper quadrant, count how many are in the circle ( $N_c$ ), and then calculate:

Let's calculate  $\pi$  with Dask.delayed  
*delayed\_calc\_pi.ipynb*



$$\pi \approx \frac{4 N_c}{N}$$



# Dask Performance

- Rather than re-creating the info here, let's go take a look at the docs for:
- [Best Practices](#)
- [Debugging and Performance](#)

# Incorporating Dask into Your Code

- Starting points depending on what your code does:
  - Pandas DataFrames → Dask DataFrame
  - Large Numpy arrays → Dask Array (or [Xarray](#) + Dask)
  - Functions that process and manipulate various kinds of data that don't neatly fit into DataFrames (of any kind) → Dask Bag
  - Python multiprocessing or joblib libraries for Python-level parallelism → Dask Delayed
  - And of course these can be mixed & matched any way that seems appropriate.
- Parallelism:
  - Mostly Dask DataFrames or Arrays → try the simple threaded scheduler
  - Anything else → `dask.distributed` processes (single or multithreaded)

