Introduction to Dask

PARALLEL PROGRAMMING WITH DASK IN PYTHON

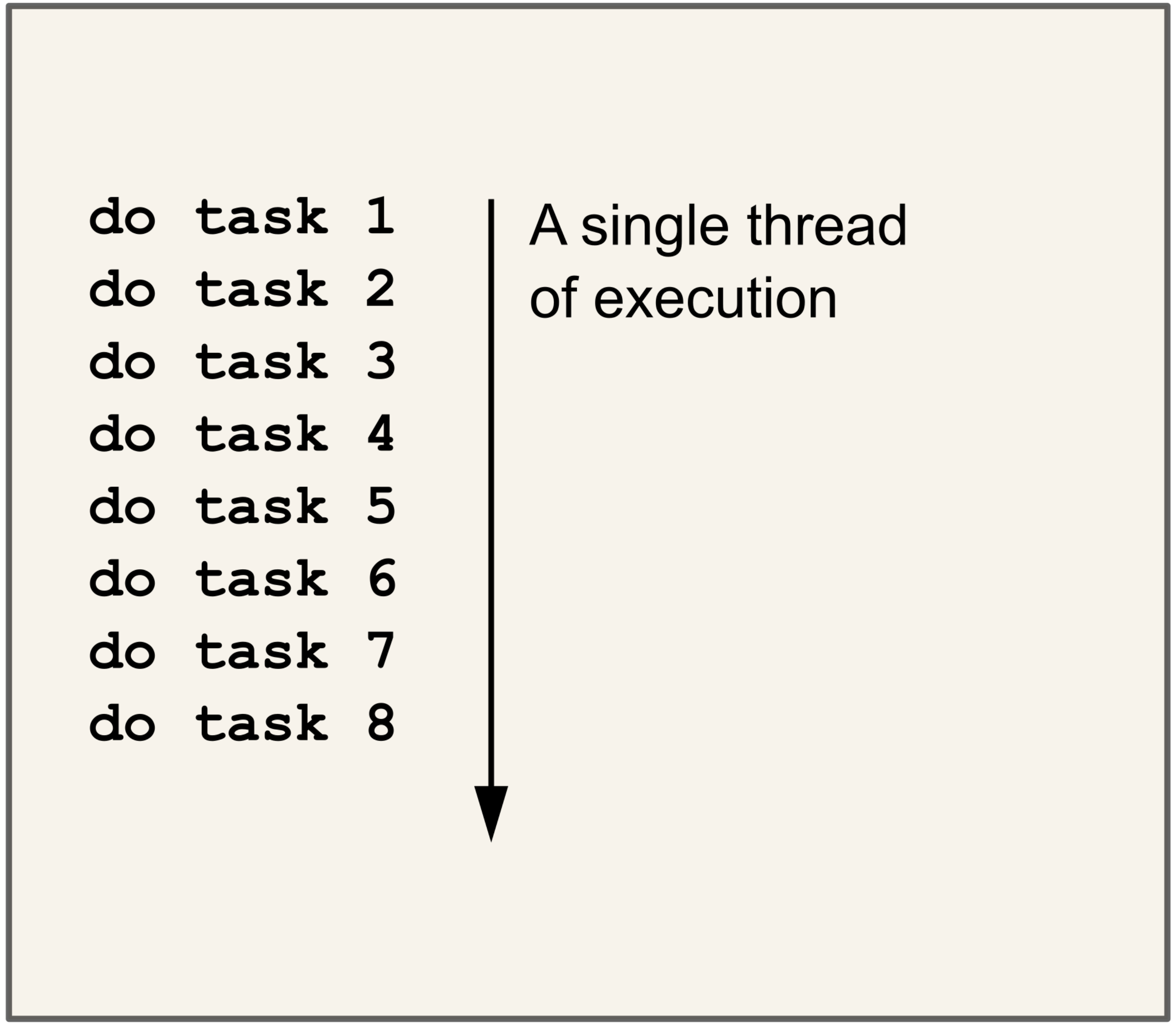
James Fulton
Climate Informatics Researcher

datacamp

Speeding up computations using multiple cores

- Computers have multiple cores
- Code needs to be written to use them
- The Dask package can be used to do this
- Complete our computations faster

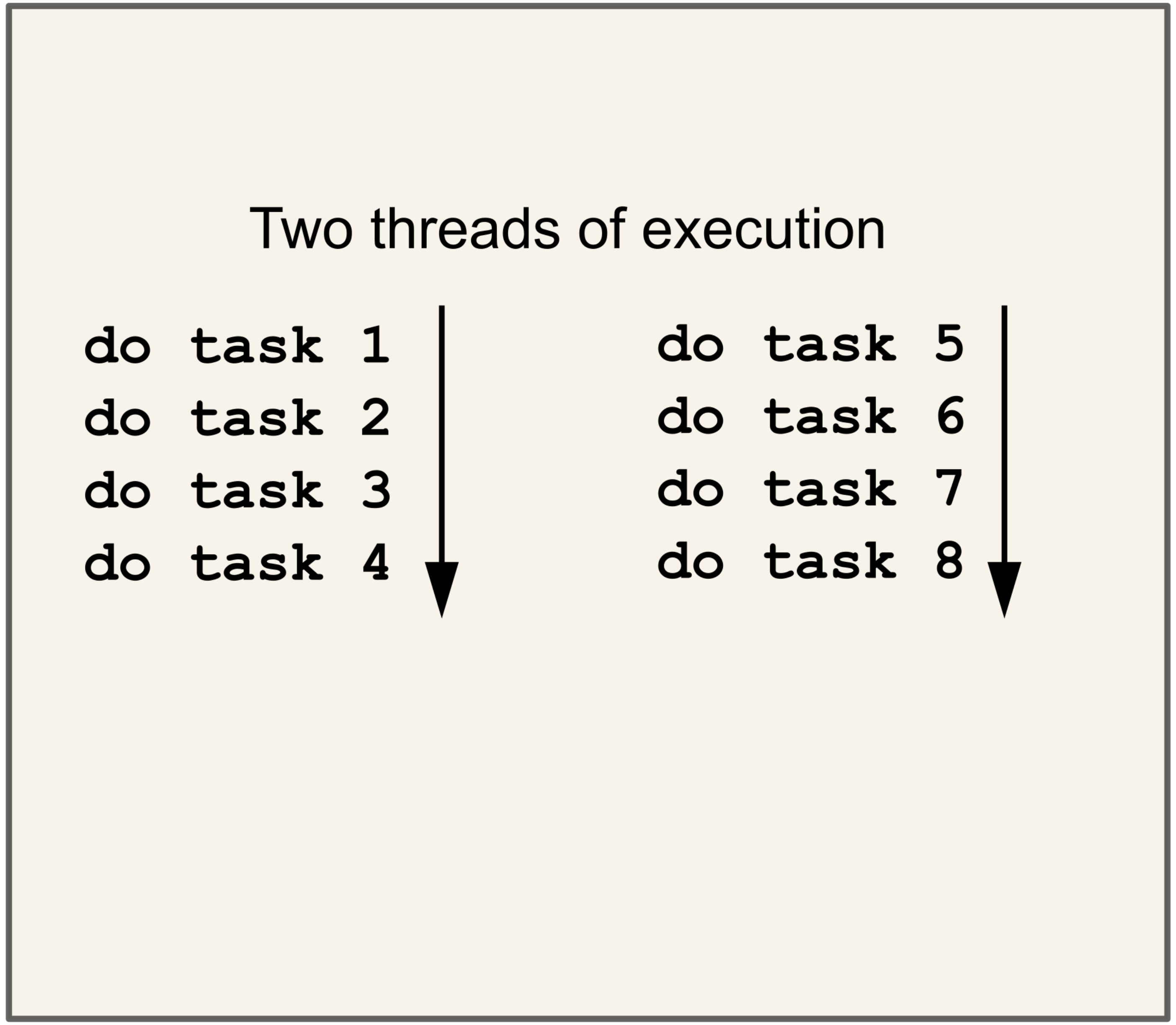
Concurrent programming



```
do task 1
do task 2
do task 3
do task 4
do task 5
do task 6
do task 7
do task 8
```

A single thread of execution

Multi-threading

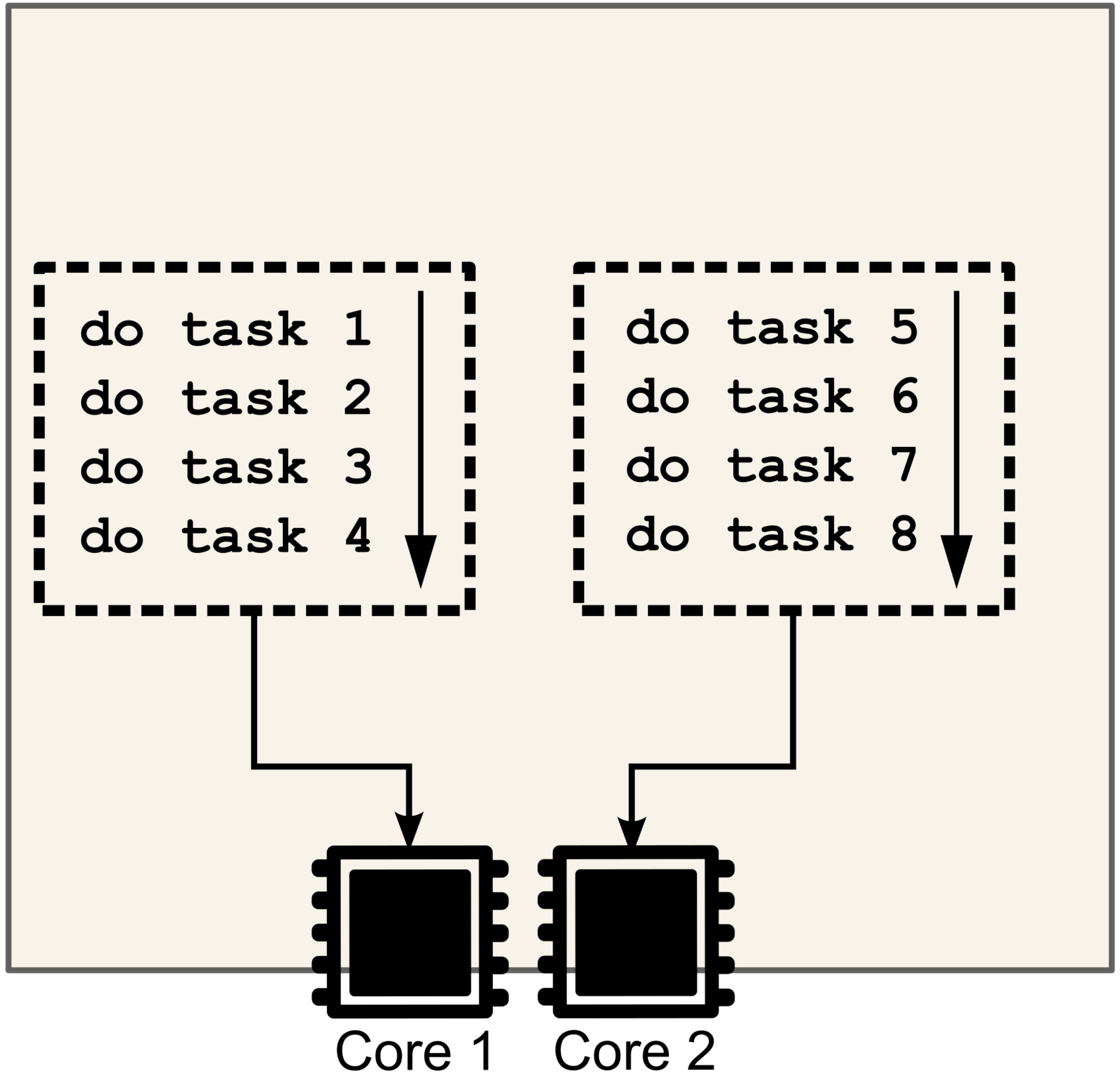


Two threads of execution

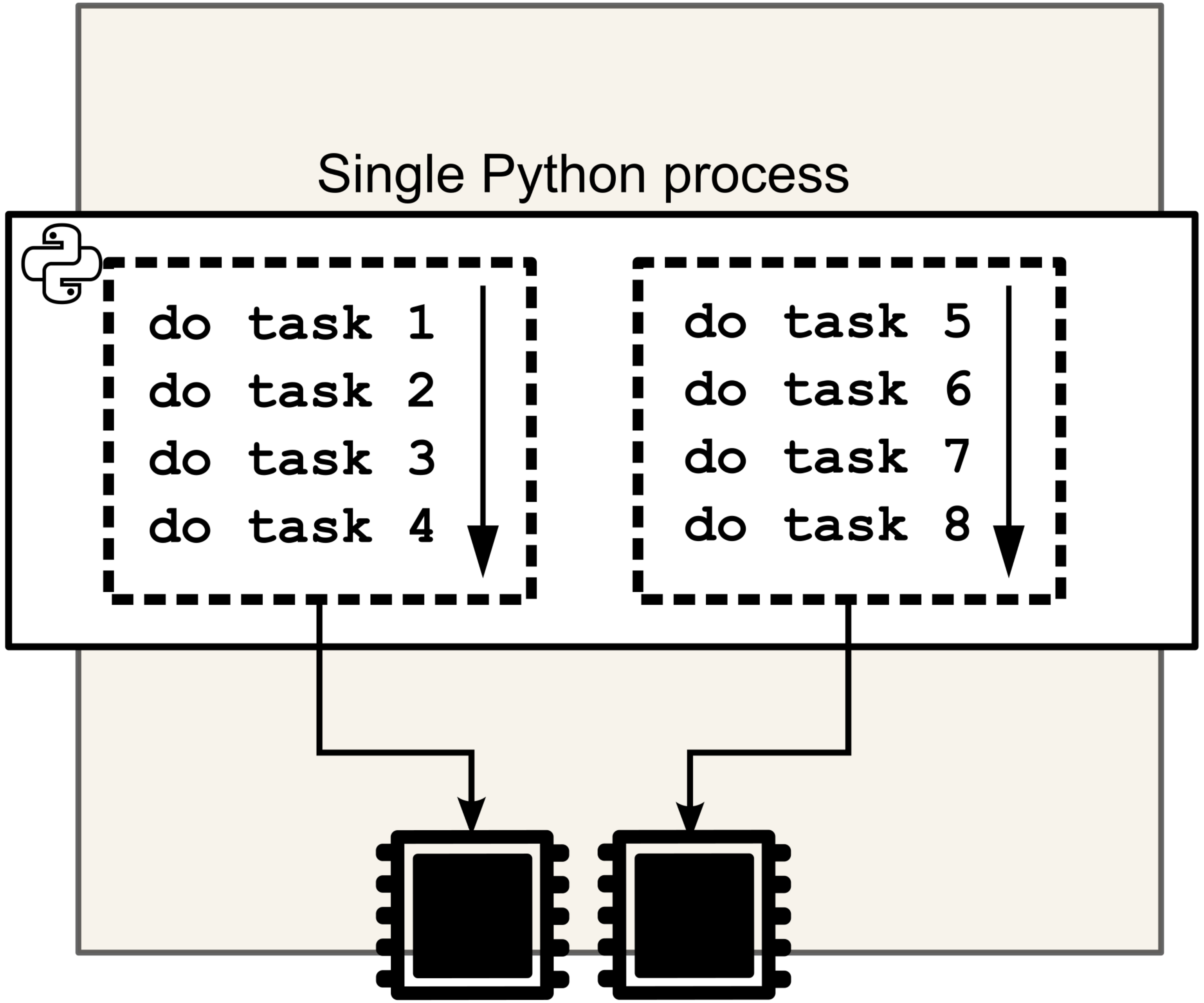
```
do task 1
do task 2
do task 3
do task 4
```

```
do task 5
do task 6
do task 7
do task 8
```

Multi-threading



Multi-threading

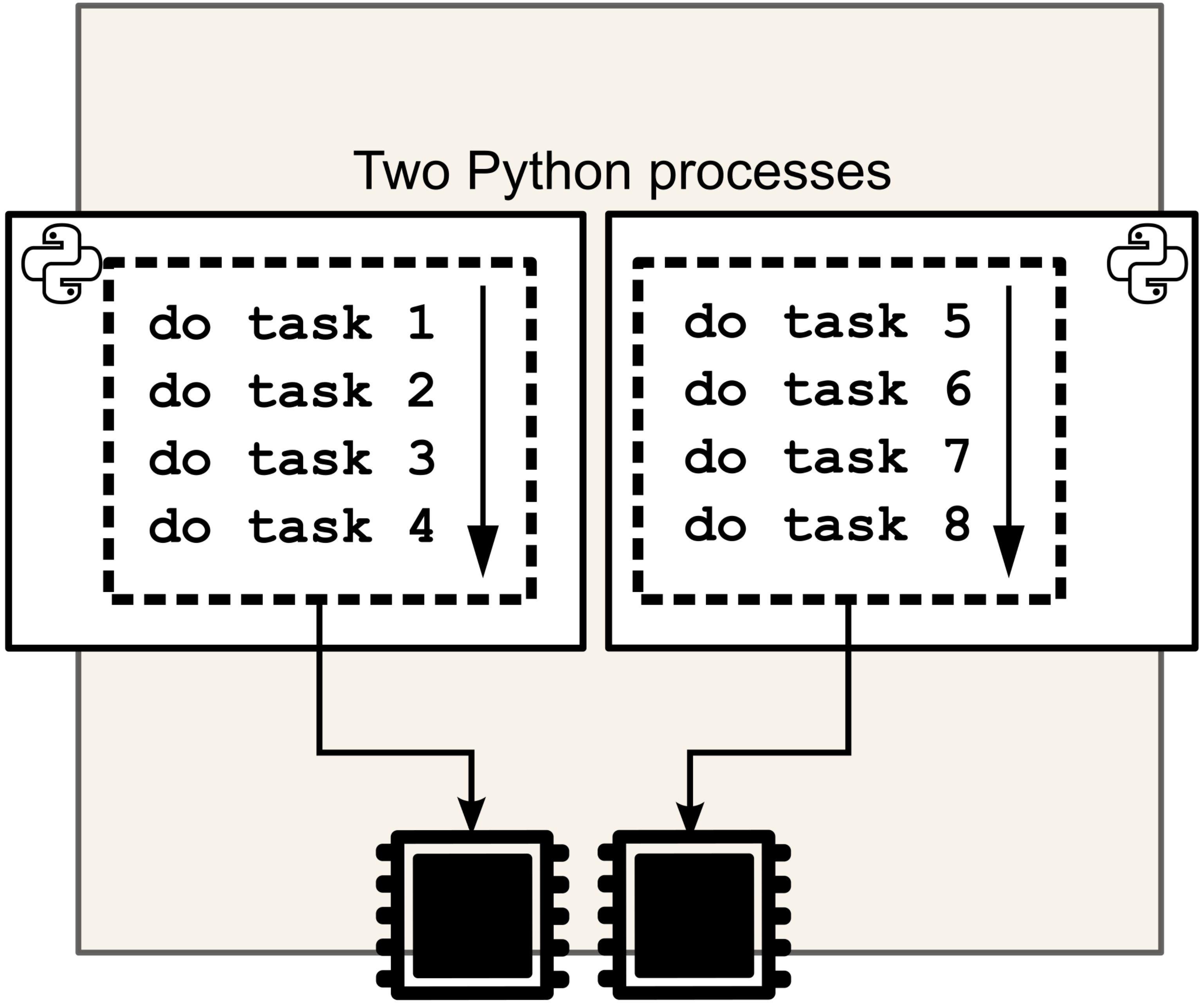


Single Python process

```
do task 1
do task 2
do task 3
do task 4
```

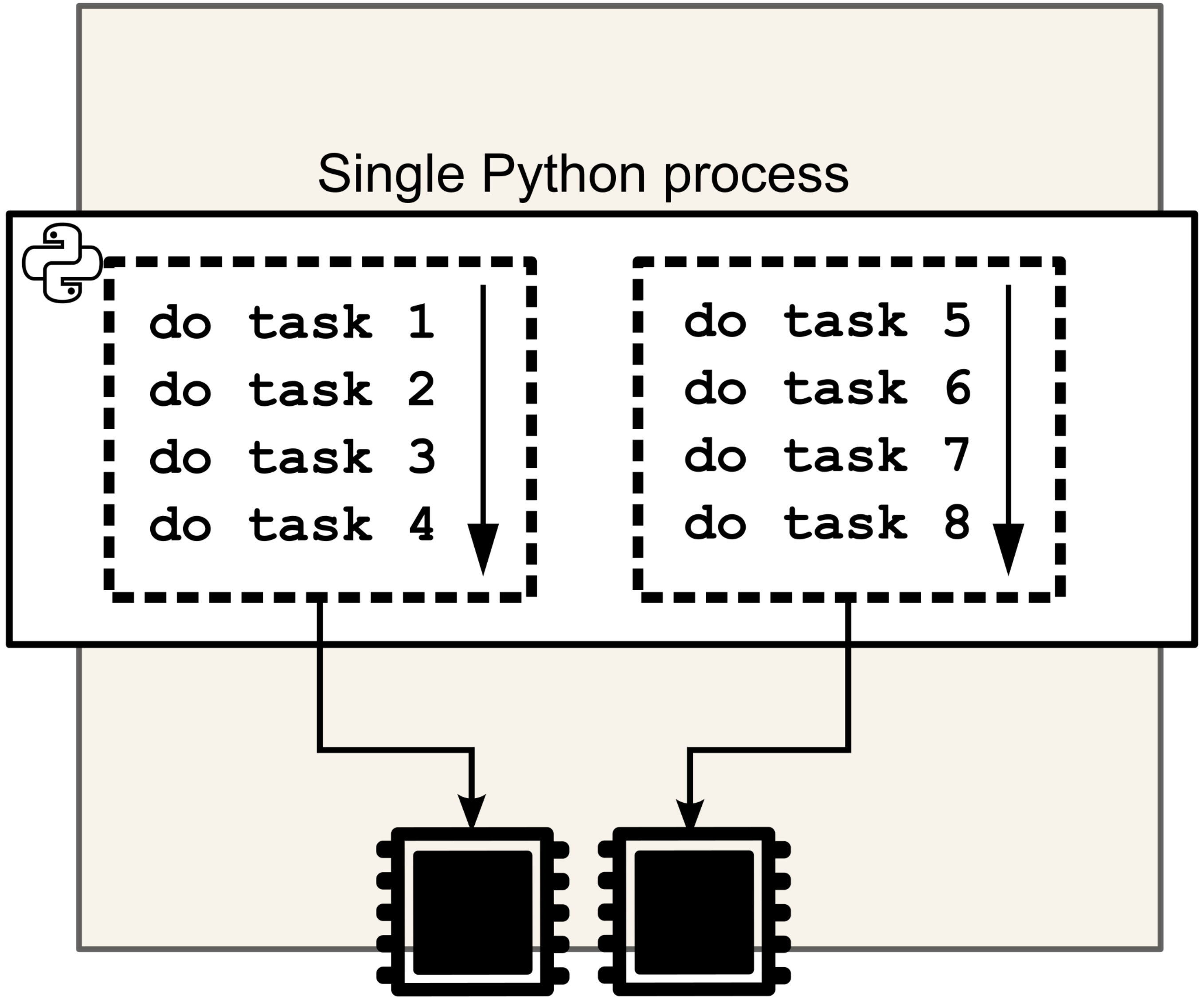
```
do task 5
do task 6
do task 7
do task 8
```

Parallel processing

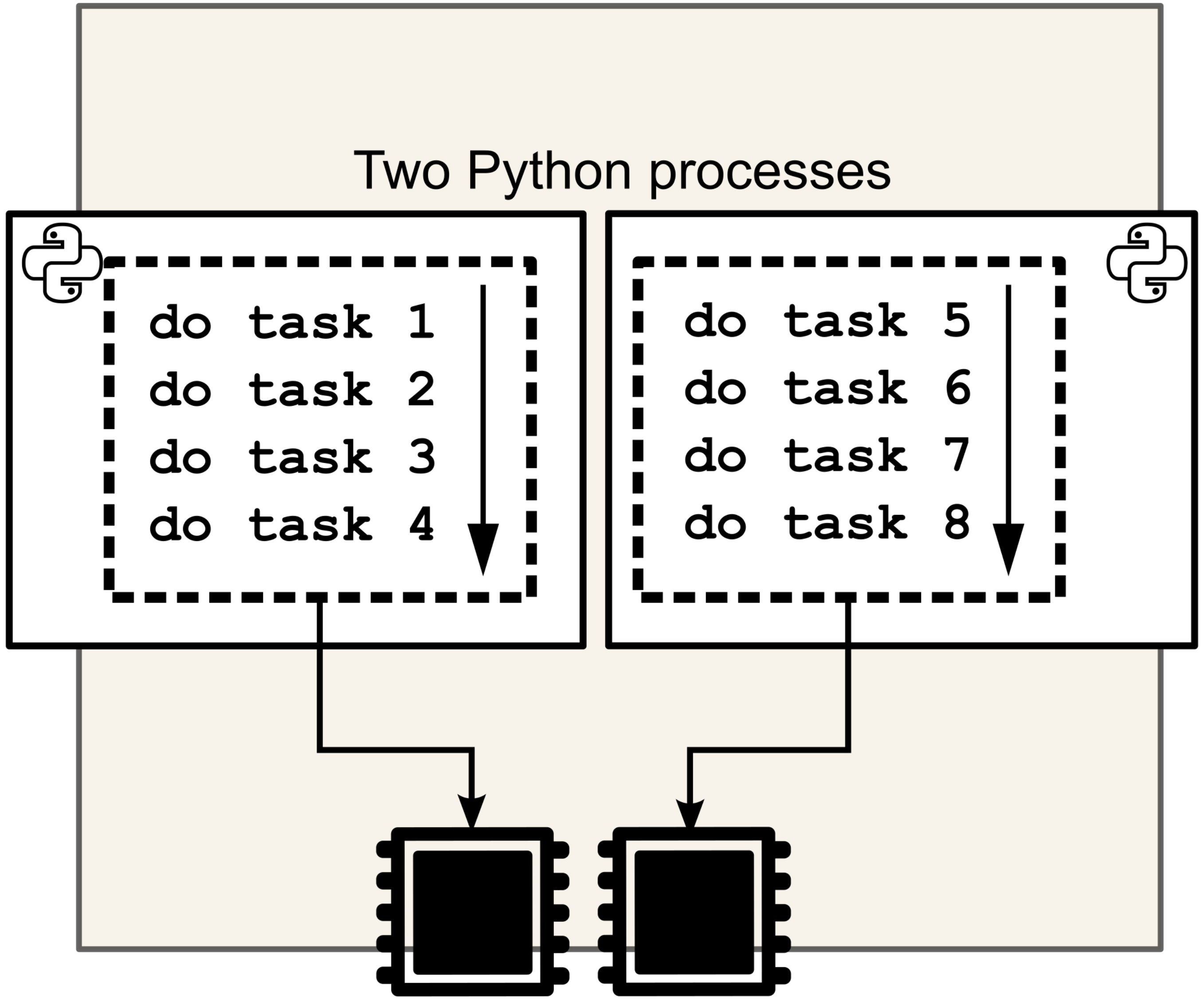


Parallel programming

Multi-threading



Parallel processing



Lazy evaluation

- Computations are not run until the moment the result is needed
- The steps required to compute the result are stored for later
- Dask splits the tasks between threads or processes

Dask delayed

```python
from dask import delayed

def my_square_function(x):
    return x**2

# Create delayed version of above function
delayed_square_function = delayed(my_square_function)
```

Dask delayed

```python
from dask import delayed

def my_square_function(x):
    return x**2

# Create delayed version of above function
delayed_square_function = delayed(my_square_function)

# Use the delayed function with input 4
delayed_result = delayed_square_function(4)

# Print the delayed answer
print(delayed_result)
```

```
Delayed('my_square_function-7f71b132-70a9-457a-aa52-604e8c34f8a7')
```

Dask delayed

```python
from dask import delayed

def my_square_function(x):
    return x**2

# Delay and use function
delayed_result = delayed(my_square_function)(4)

print(delayed_result)
```

```
Delayed('my_square_function-7f71b132-70a9-457a-aa52-604e8c34f8a7')
```

Computing the answer

```python
from dask import delayed

def my_square_function(x):
    return x**2

delayed_result = delayed(my_square_function)(4)

real_result = delayed_result.compute() # <- This line is where the calculation happens

# Print the answer
print(real_result)
```

```
16
```

Using operations on delayed objects

```
delayed_result1 = delayed(my_square_function)(4)

# Math operations return delayed object
delayed_result2 = (4 + delayed_result1) * 5

print(delayed_result2.compute())
```

```
100
```

Lazy evaluation

```
x_list = [30, 85, 14, 12, 27, 62, 89, 15, 78,  0]

sum_of_squares = 0

for x in x_list:
    # Square and add numbers
    sum_of_squares += delayed(my_square_function)(x)
```

Lazy evaluation

```python
x_list = [30, 85, 14, 12, 27, 62, 89, 15, 78,  0]

sum_of_squares = 0

for x in x_list:
    # Square and add numbers
    sum_of_squares += delayed(my_square_function)(x)

result = sum_of_squares.compute()

# Print the answer
print(result)
```

```
27268
```

Sharing computation

```
delayed_intermediate = delayed(my_square_function)(3)

# These two results both use delayed_intermediate
delayed_result1 = delayed_intermediate - 5
delayed_result2 = delayed_intermediate + 4

# delayed_3_squared will be computed twice
print('delayed_result1:', delayed_result1.compute())
print('delayed_result2:', delayed_result2.compute())
```

```
delayed_result1: 4
delayed_result2: 13
```

Sharing computation

```
import dask

# delayed_intermediate will be computed once
comp_result1, comp_result2 = dask.compute(delayed_result1, delayed_result2)

print('comp_result1:', comp_result1)
print('comp_result2:', comp_result2)
```

```
delayed_result1: 4
delayed_result2: 13
```

Let's practice!

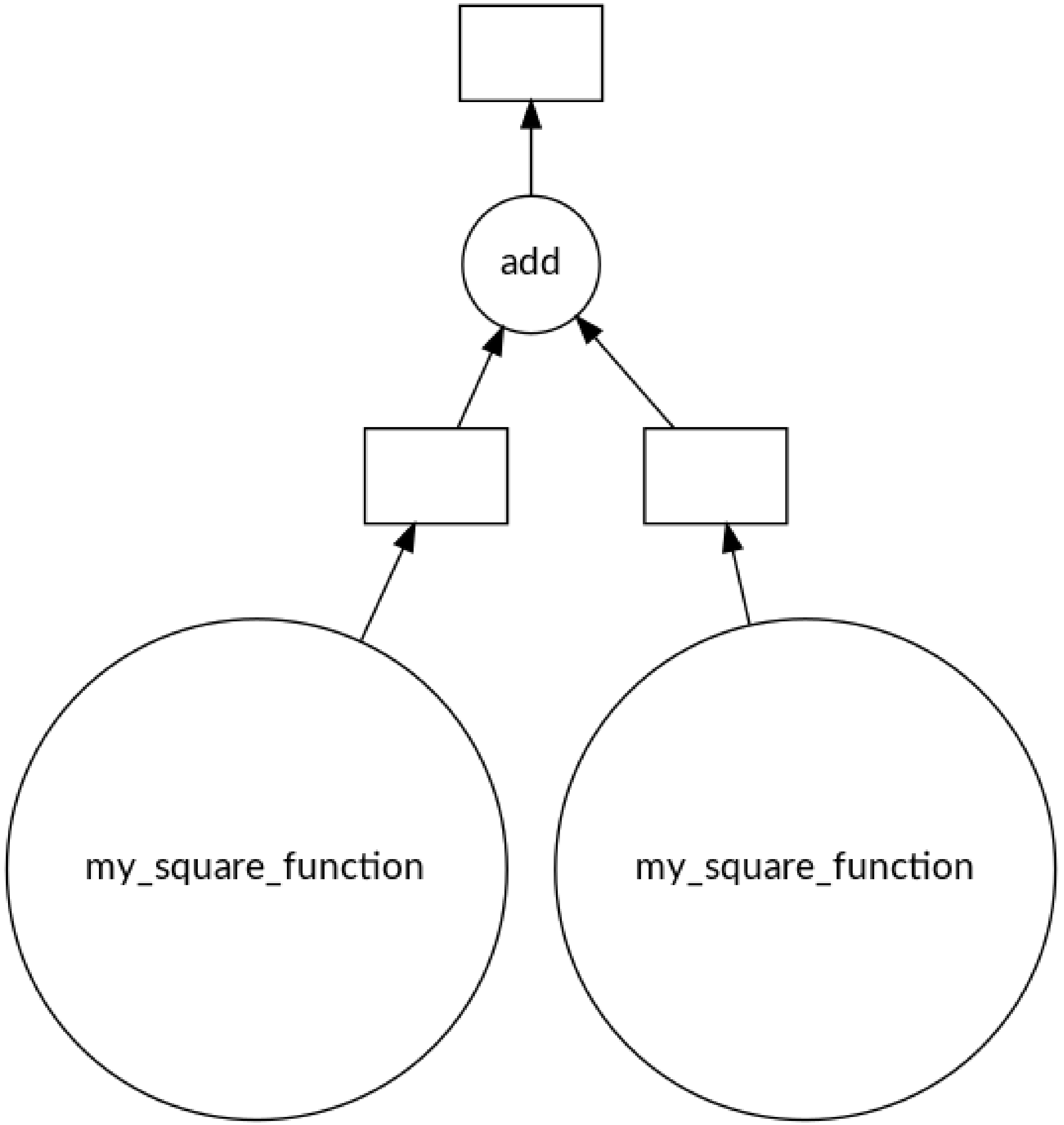
Task graphs and scheduling methods

PARALLEL PROGRAMMING WITH DASK IN PYTHON

James Fulton
Climate Informatics Researcher

Visualizing a task graph

```python
# Create 2 delayed objects
delayed_num1 = delayed(my_square_function)(3)
delayed_num2 = delayed(my_square_function)(4)

# Add them
result = delayed_num1 + delayed_num2

# Plot the task graph
result.visualize()
```

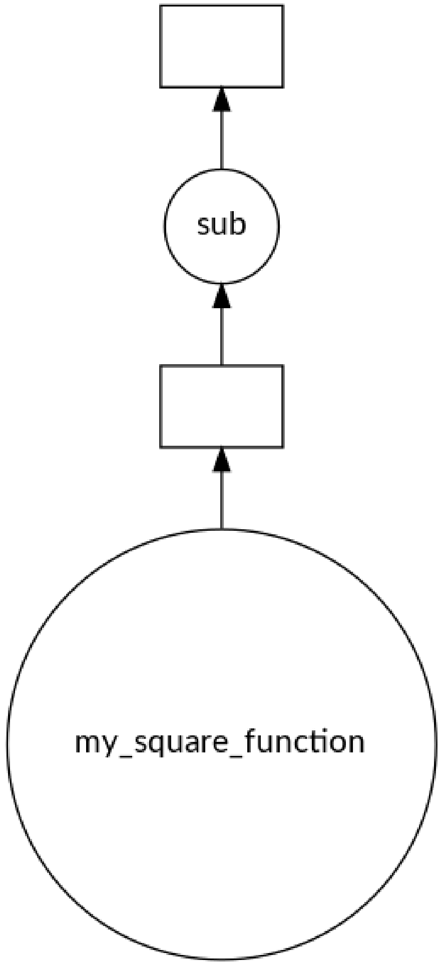


Overlapping task graph

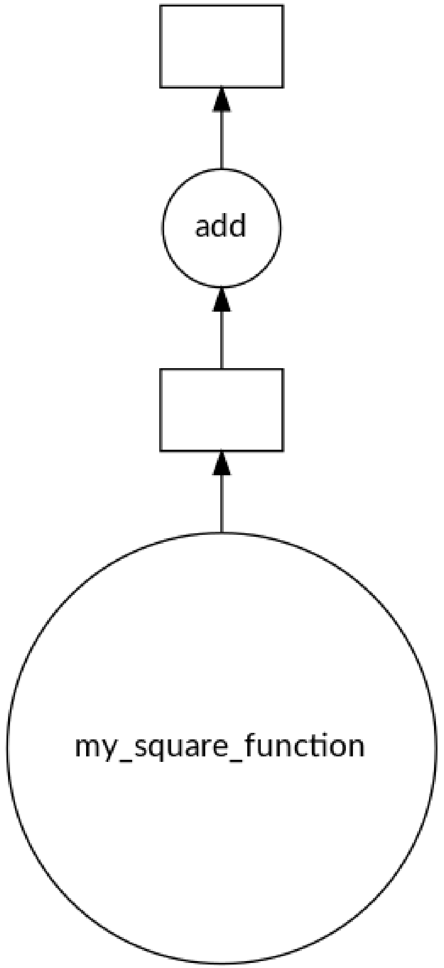
```
delayed_intermediate = delayed(my_square_function)(3)


# These two results both use delayed_intermediate_result
delayed_result1 = delayed_intermediate - 5
delayed_result2 = delayed_intermediate + 4
```

Overlapping task graph

```
delayed_result1.visualize()
```

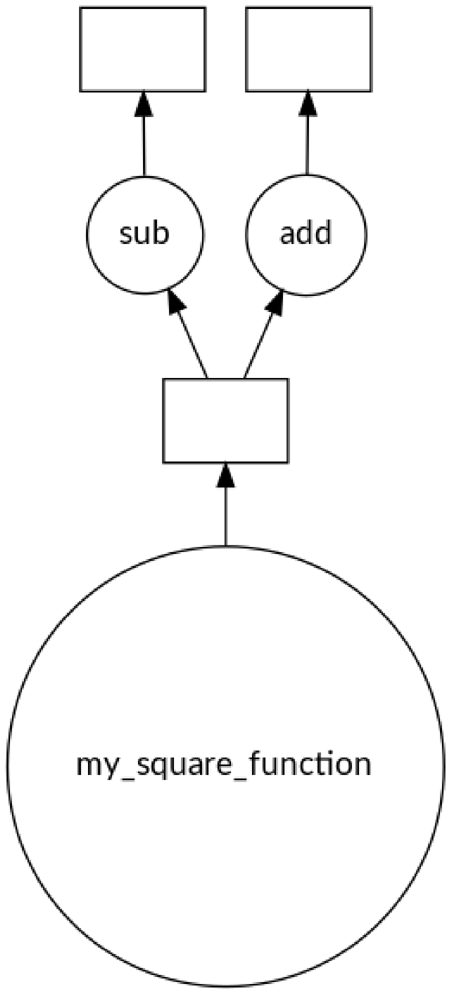


```
delayed_result2.visualize()
```



Overlapping task graph

```
# Plot the task graph
dask.visualize(delayed_result1, delayed_result2)
```



Multi-threading vs. parallel processing

Moving data

Parallel processing

- Processes have their own RAM space

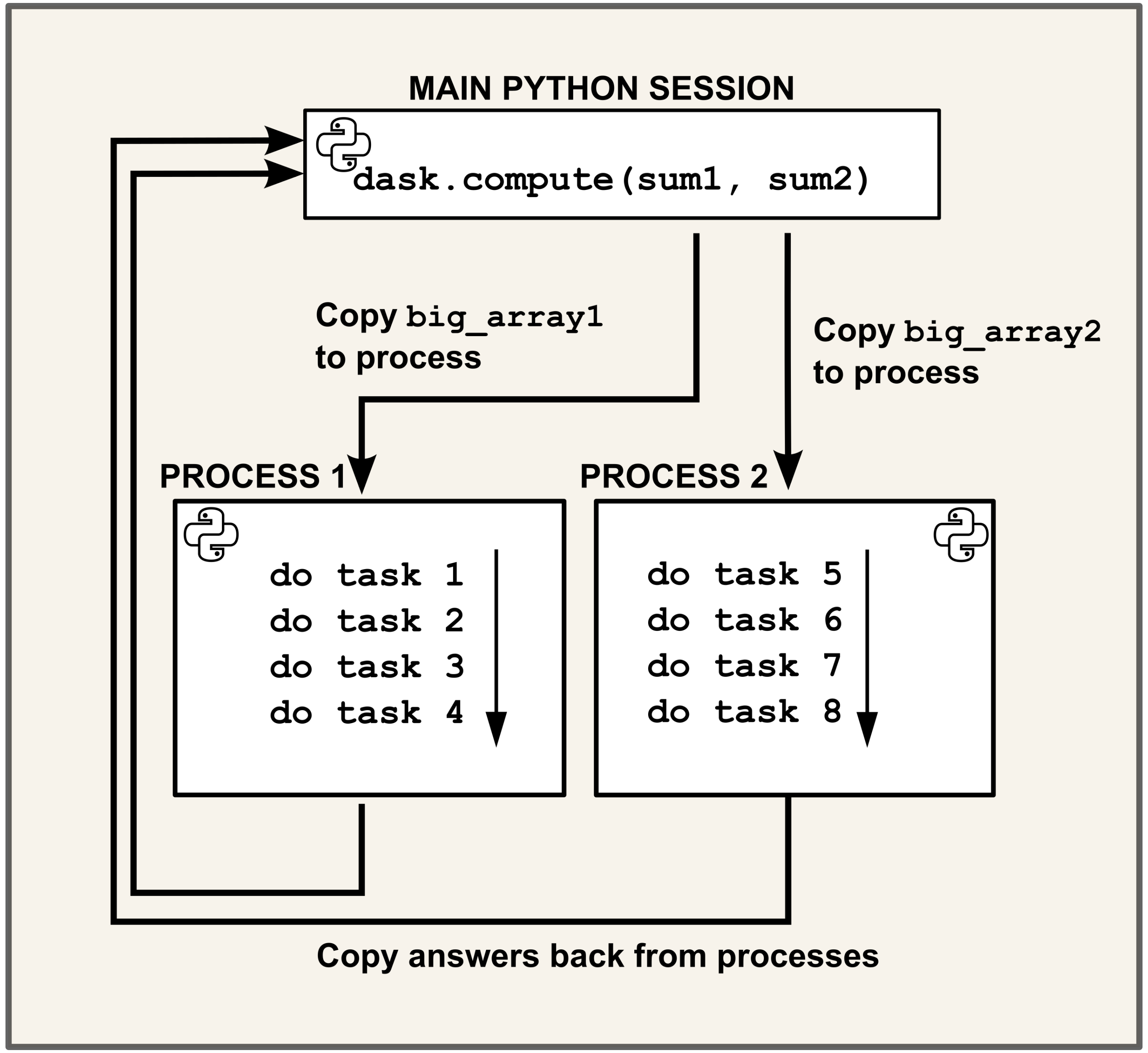
Multi-threading

- Threads use the same RAM space

Multi-threading vs. parallel processing

```
# Run a sum on two big arrays
sum1 = delayed(np.sum)(big_array1)
sum2 = delayed(np.sum)(big_array2)

# Compute using processes
dask.compute(sum1, sum2)
```

- Slow using parallel processing



MAIN PYTHON SESSION

```
dask.compute(sum1, sum2)
```

Copy `big_array1` to process

Copy `big_array2` to process

PROCESS 1

```
do task 1
do task 2
do task 3
do task 4
```

PROCESS 2

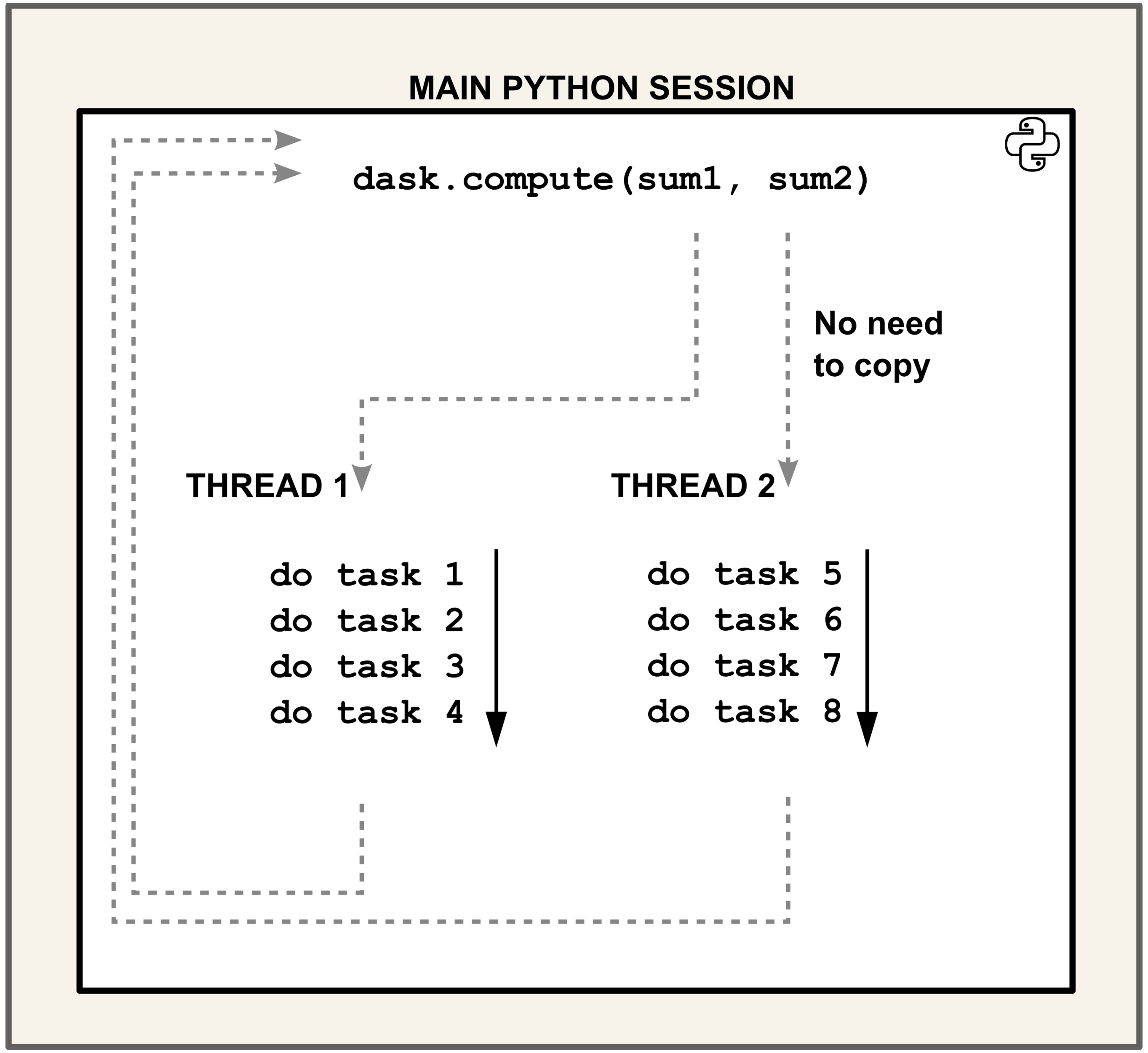
```
do task 5
do task 6
do task 7
do task 8
```

Copy answers back from processes

Multi-threading vs. parallel processing

```
# Run a sum on two big arrays
sum1 = delayed(np.sum)(big_array1)
sum2 = delayed(np.sum)(big_array2)

# Compute using threads
dask.compute(sum1, sum2)
```

- Fast using multi-threading



```
dask.compute(sum1, sum2)
```

```
do task 1
do task 2
do task 3
do task 4
```

```
do task 5
do task 6
do task 7
do task 8
```

The GIL

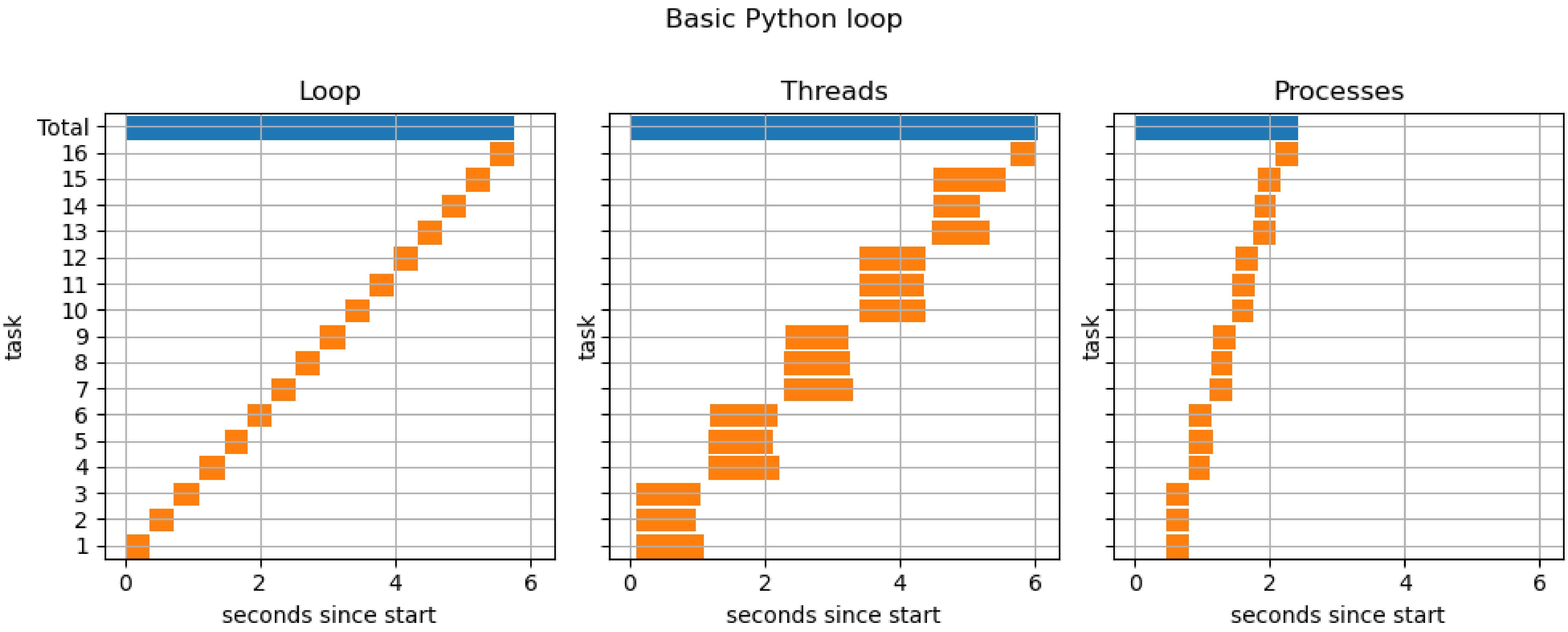
Global interpreter lock - only one thread can read the Python script at a time

```python
def sum_to_n(n):
    """Sums numbers from 0 to n"""
    total = 0
    for i in range(n+1):
        total += i
    return total
```

- Multi-threading won't help here
- Parallel processing will

```python
sum1 = delayed(sum_to_n)(1000)
sum2 = delayed(sum_to_n)(1000)
```

Example timings - GIL

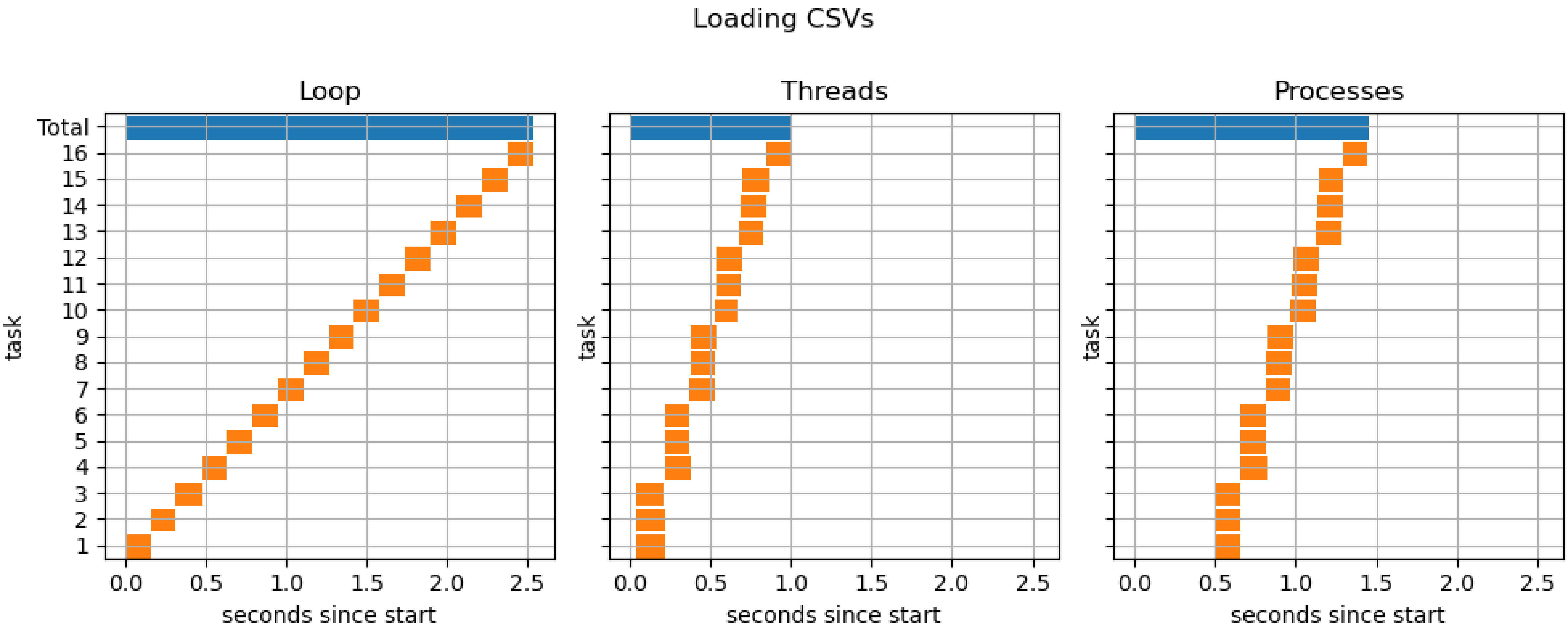


Functions which release the GIL

- E.g. the `pd.read_csv()` function releases the GIL

```
df1 = delayed(pd.read_csv)('file1.csv')
df2 = delayed(pd.read_csv)('file2.csv')
```

Example timings - Loading data



Summary

Threads

- Are very fast to initiate
- Share memory space with main session
- No memory transfer needed
- Limited by the GIL, which allows one thread to read the code at once

Processes

- Take time and memory to set up
- Have separate memory pools
- Very slow to transfer data between themselves and to the main Python session
- Each have their own GIL and so don't need to take turns reading the code

Let's practice!

PARALLEL PROGRAMMING WITH DASK IN PYTHON

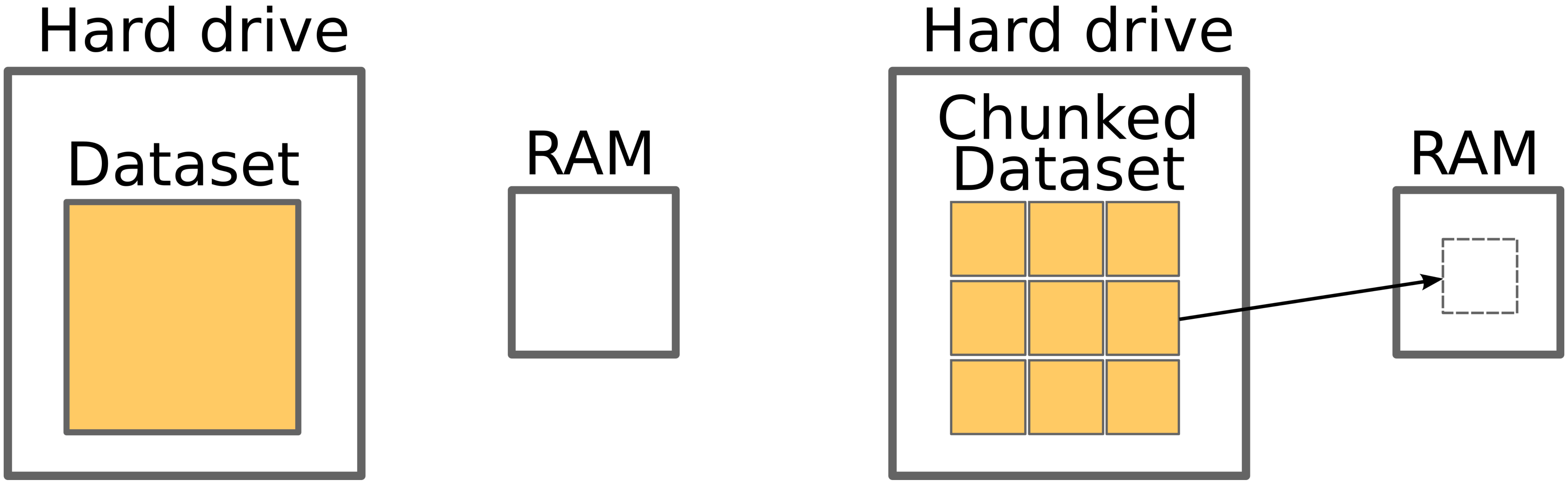
Building delayed pipelines

PARALLEL PROGRAMMING WITH DASK IN PYTHON

James Fulton
Climate Informatics Researcher

datacamp

Chunks of data



Spotify songs dataset

```
files = [
   '2005_tracks.csv',
   '2006_tracks.csv',
   '2007_tracks.csv',
   '2008_tracks.csv',
   '2009_tracks.csv',
   '2010_tracks.csv',
   ...
   '2020_tracks.csv',
]
```

Spotify songs dataset

```
                       name  duration_ms release_date  ...
0     Aldrig (feat. Carmon)       247869   2019-01-01  ...
2  2019 - The Year to Build       288105   2019-01-01  ...
3                 Na zawsze       186812   2019-01-01  ...
4         Humo en la Trampa       258354   2019-01-01  ...
5                     Au Au       176000   2019-01-01  ...
...                     ...          ...          ...  ...
```

Analyzing the data

```python
import pandas as pd

maximums = []

for file in files:
    # Load each file
    df = pd.read_csv(file)
    # Find maximum track length in each file
    max_length = df['duration_ms'].max()
    # Store this maximum
    maximums.append(max_length)

# Find the maximum of all the maximum lengths
absolute_maximum = max(maximums)
```

Analyzing the data

```python
import pandas as pd

maximums = []

for file in files:
    # Load each file
    df = delayed(pd.read_csv)(file) # <------- delay loading
    # Find maximum track length in each file
    max_length = df['duration_ms'].max()
    # Store this maximum
    maximums.append(max_length)

# Find the maximum of all the maximum lengths
absolute_maximum = max(maximums)
```

Analyzing the data

```python
import pandas as pd

maximums = []

for file in files:
    # Load each file
    df = delayed(pd.read_csv)(file) # <-------- delay loading
    # Find maximum track length in each file
    max_length = df['duration_ms'].max()
    # Store this maximum
    maximums.append(max_length)

# Find the maximum of all the maximum lengths
absolute_maximum = delayed(max)(maximums) # <-------- delay max() function
```

Using methods of a delayed object

```python
import pandas as pd

maximums = []

for file in files:
    df = delayed(pd.read_csv)(file)
    # Use the .max() method
    max_length = df['duration_ms'].max()

    maximums.append(max_length)


absolute_maximum = delayed(max)(maximums)
```

```python
print(max_length)
```

```
Delayed('max-0602855d-3ee6-4c43-a4d2-...')
```

- Delayed object methods and properties return new delayed objects

```python
print(df.shape)
print(df.shape.compute())
```

```
Delayed('getattr-bc1e8838ab...')
(11907, 12)
```

Using methods of a delayed object

```python
import pandas as pd

maximums = []

for file in files:
    df = delayed(pd.read_csv)(file)
    # Use a method which doesn't exist
    max_length = df['duration_ms'].fake()

    maximums.append(max_length)


absolute_maximum = delayed(max)(maximums)
```

```python
print(max_length)
```

```
Delayed('max-6c026036-5daf-4b2-...')
```

- Methods aren't run until after `.compute()` is used

```python
print(max_length.compute())
```

```
...
AttributeError: 'Series' object has no
attribute 'fake'
```

Using methods of a delayed object

```
import pandas as pd

maximums = []

for file in files:
    df = delayed(pd.read_csv)(file)

    max_length = df['duration_ms'].max()
    # Add delayed object to list
    maximums.append(max_length)

# Run delayed max on delayed objects list
absolute_maximum = delayed(max)(maximums)
```

`maximums` is a list of delayed objects

```
print(maximums)
```

```
[Delayed('max-80b...'),
Delayed('max-fa15d...',
...]
```

Computing lists of delayed objects

```python
import pandas as pd

maximums = []

for file in files:
    df = delayed(pd.read_csv)(file)

    max_length = df['duration_ms'].max()
    # Add dalayed object to list
    maximums.append(max_length)

# Compute all the maximums
all_maximums = dask.compute(maximums)
```

```python
print(all_maximums)
```

```
([2539418, 4368000, ...
... 4511716, 4864333],)
```

Computing lists of delayed objects

```python
import pandas as pd

maximums = []

for file in files:
    df = delayed(pd.read_csv)(file)

    max_length = df['duration_ms'].max()

    maximums.append(max_length)

# Compute all the maximums
all_maximums = dask.compute(maximums)[0]
```

```python
print(all_maximums)
```

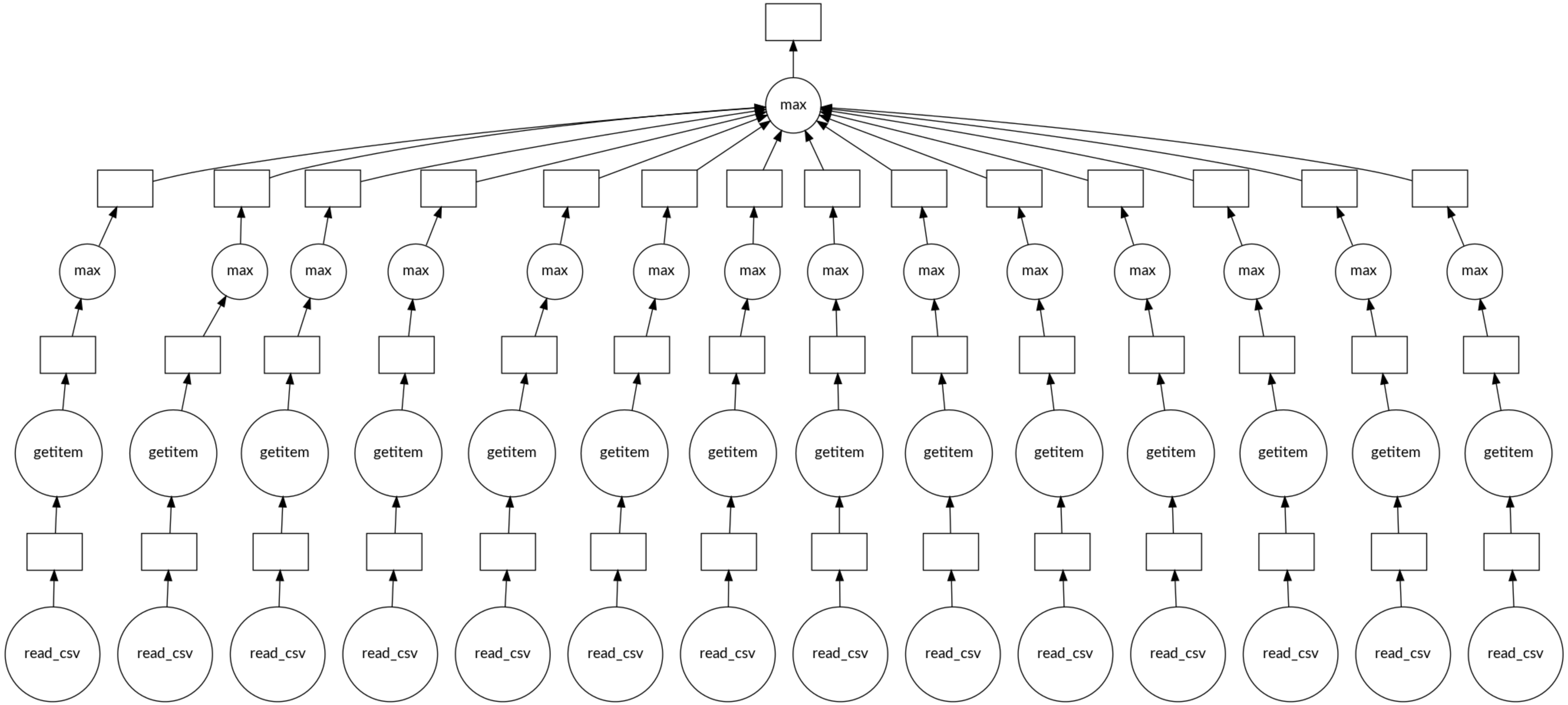
```
[2539418, 4368000, ...
... 4511716, 4864333]
```

To delay or not to delay

```python
def get_max_track(df):
    return df['duration_ms'].max()

for file in files:
    df = delayed(pd.read_csv)(file)
    # Use function to find max
    max_length = get_max_track(df)

    maximums.append(max_length)



absolute_maximum = delayed(max)(maximums)
```

Deeper task graph

```
absolute_maximum.visualize()
```



Let's practice!

PARALLEL PROGRAMMING WITH DASK IN PYTHON