浙江大学爱丁堡大学联合学院
ZJU-UoE Institute

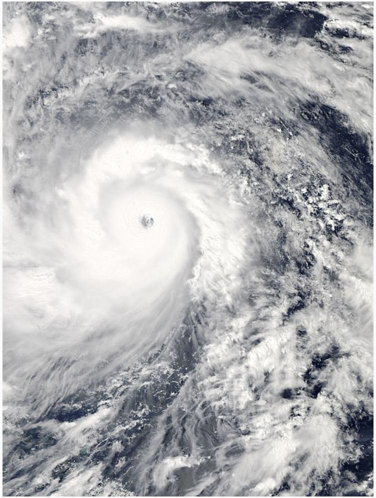
Dealing with large datasets - Part 2

Nicola Romanò - nicola.romano@ed.ac.uk

Last lecture we learned about...

- Big data, and the problems associated with it!
- Using HDF5 file format to store and efficiently access big datatets.

Danger, too much data ahead!



Souce: NASA, Public Domain

How not to drown in a sea of [image] data?

Problem-dependent solutions, today we will focus on:

- Hardware solutions
- Choice of file format
- Parallelization
- Lazy evaluation/loading
- ...

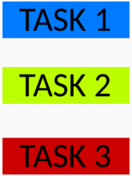
Learning objectives

At the end of this lecture you should be able to:

- Describe the concept of parallelization
- Implement simple parallelization in Python
- Use dask to access larger-than-memory arrays in Python
- Use dask to perform delayed computation

Improving computer specifics is not always possible and is not necessarily the best solution.

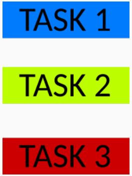
Imagine having three tasks to perform



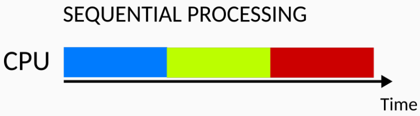
Parallelization

Improving computer specifics is not always possible and is not necessarily the best solution.

Imagine having three tasks to perform



We can perform the tasks in different ways

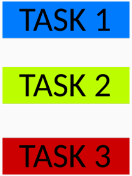


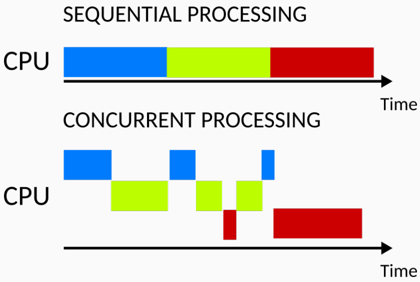
Parallelization

Improving computer specifics is not always possible and is not necessarily the best solution.

We can perform the tasks in different ways

Imagine having three tasks to perform

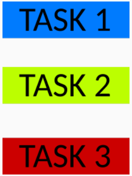




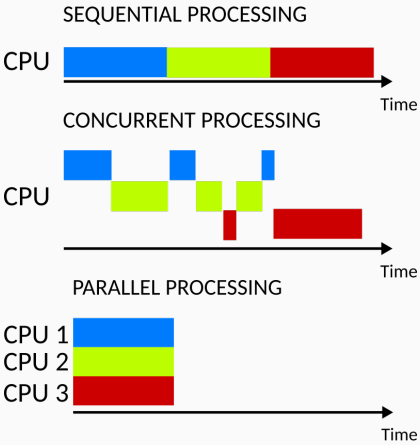
Parallelization

Improving computer specifics is not always possible and is not necessarily the best solution.

Imagine having three tasks to perform



We can perform the tasks in different ways



Sequential, concurrent, and parallel processing

A few considerations on different processing modalities.

- Concurrent processing can only work if the tasks are **independent**.
- If task 3 depends on the output of task 2, which depends on the output of task 1, then you need to execute them sequentially.
- Concurrent processing can be faster than sequential, even on a single processor, for example if tasks need to wait for data.
- Remember that there is a (small) overhead in switching between "tasks"

Threads and processes

- A **process** is the instance of a computer program executed by the operating system.
- A process can "spawn" multiple **threads**, each of which can execute an independent operation.
- Processes don’t share memory space, while threads do.

Threads and processes

- A **process** is the instance of a computer program executed by the operating system.
- A process can "spawn" multiple **threads**, each of which can execute an independent operation.
- Processes don’t share memory space, while threads do.
- To avoid *race conditions*, where multiple threads can modify the same data at the same time, Python implements a **Global Interpreter Lock** (GIL), which prevents threads to run at the same time
- The GIL makes python *thread safe*, but does not allow running threads on multiple processors.

Threads and processes

- A **process** is the instance of a computer program executed by the operating system.
- A process can "spawn" multiple **threads**, each of which can execute an independent operation.
- Processes don’t share memory space, while threads do.
- To avoid *race conditions*, where multiple threads can modify the same data at the same time, Python implements a **Global Interpreter Lock** (GIL), which prevents threads to run at the same time
- The GIL makes python *thread safe*, but does not allow running threads on multiple processors.
- Processes are not affected by GIL.

- Use the *multiprocessing* module to create processes
- Use the *threading* module for threads or alternatively the *multiprocessing.dummy* module.
- *multiprocessing.dummy* is useful as it allows you to use the same exact code for creating processes but it creates threads instead!
- More examples in the practical!

Sequential execution

We execute a slow function sequentially

```
from time import sleep

def slow_function(n):
   # This stops execution for 1 second
   sleep(1)
return n*n

result = [slow_function(i) for i in range(4)]
print(result)
```

This takes (on my laptop with 4 cores) 4s and returns [0, 1, 4, 9]

Parallel execution

We execute a slow function in parallel

```
import multiprocessing as mp

# Start a "pool" of processes holding a maximum of
# processes equal to the number of CPU cores
pool = mp.Pool(mp.cpu_count())

result = pool.map(slow_function, range(4))
pool.close()
pool.join()
print(result)
```

Parallel execution

We execute a slow function in parallel

```
import multiprocessing as mp

# Start a "pool" of processes holding a maximum of
# processes equal to the number of CPU cores
pool = mp.Pool(mp.cpu_count())

result = pool.map(slow_function, range(4))
pool.close()
pool.join()
print(result)
```

We execute a slow function in parallel

```
import multiprocessing as mp

# Start a "pool" of processes holding a maximum of
# processes equal to the number of CPU cores
pool = mp.Pool(mp.cpu_count())

result = pool.map(slow_function, range(4))
pool.close()
pool.join()
print(result)
```

This takes (on my laptop with 4 cores) 1.17s and still returns [0, 1, 4, 9] (luckily!). Note that processes may finish in different order, but that does not affect the order of the output!

Clean up the pool!

What does this mean, at the end of the previous piece of code?

```
pool.close()
pool.join()
```

Clean up the pool!

What does this mean, at the end of the previous piece of code?

```
pool.close()
pool.join()
```

- Pools are reusable.
- You could call `pool.map` again on the same pool.

Clean up the pool!

What does this mean, at the end of the previous piece of code?

```
pool.close()
pool.join()
```

- Pools are reusable.
- You could call `pool.map` again on the same pool.
- You should call `pool.close()` when you are done with the pool.
- `pool.close` prevents any more tasks from being submitted to the pool.

Clean up the pool!

What does this mean, at the end of the previous piece of code?

```
pool.close()
pool.join()
```

- Pools are reusable.
- You could call `pool.map` again on the same pool.
- You should call `pool.close()` when you are done with the pool.
- `pool.close` prevents any more tasks from being submitted to the pool.
- `pool.join` should be called **after** `pool.close`
- `pool.join` makes the pool wait until all tasks are completed then exits.

Dealing with larger than memory arrays

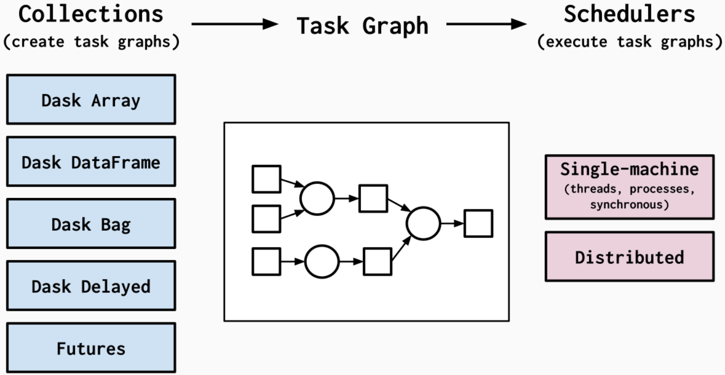
Consider a case when you wanted to load several large images in memory, 3Gb each.
It is likely that you won't be able to load such a large amount of data all at once.

The *dask* Pyton library helps solving this problem!

Dealing with larger than memory arrays

Consider a case when you wanted to load several large images in memory, 3Gb each.
It is likely that you won't be able to load such a large amount of data all at once.

The *dask* Pyton library helps solving this problem!



Delaying operation

```
from dask import delayed

def slow_function(n):
    # This stops execution for 5 seconds
    sleep(1)
    return n * n

result = delayed(slow_function)(1)
```

Delaying operation

```
from dask import delayed

def slow_function(n):
   # This stops execution for 5 seconds
   sleep(1)
   return n * n

result = delayed(slow_function)(1)
```

This takes 1.4 ms as the code **is not executed**.
To effectively perform the operation we use

```
result.compute()
```

Which now takes 1 second to run.

Task graphs

Dask puts operations in a task graphs and executes them in an optimised way when needed.

```
def inc(x):
   sleep(1)
   return x + 1

def add(x, y):
   sleep(1)
   return x + y

x = delayed(inc)(1)
y = delayed(inc)(2)
z = delayed(add)(x, y)
```

Dask puts operations in a task graphs and executes them in an optimised way when needed.

```
def inc(x):
   sleep(1)
   return x + 1

def add(x, y):
   sleep(1)
   return x + y

x = delayed(inc)(1)
y = delayed(inc)(2)
z = delayed(add)(x, y)
```

This takes <1ms to run. Only once we run

```
z.compute()
```

The calculation is performed, and takes 2 seconds

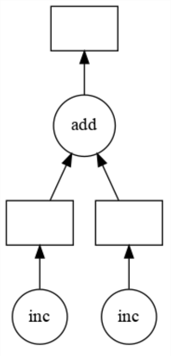
Dask puts operations in a task graphs and executes them in an optimised way when needed.

```
def inc(x):
   sleep(1)
   return x + 1

def add(x, y):
   sleep(1)
   return x + y

x = delayed(inc)(1)
y = delayed(inc)(2)
z = delayed(add)(x, y)
```

This takes <1ms to run. Only once we run

```
z.compute()
```

The calculation is performed, and takes 2 seconds



Task graph can be visualized using `z.visualize("output.png")`

You have a 1000-frame long video, and want to calculate the mean value every 10th frame

Numpy

```
import numpy as np
x = np.random.randint(0, 255,
  size=(1000, 1024, 1024))
y = x[::10].mean()
```

Dask

```
import numpy as np
import dask.array as da

#  1 billion elements
x = da.random.randint(0, 255,
  size=(1000, 1024, 1024),
  chunks=(100, 64, 64))
y = x[::10].mean()
```

The dask array container

You have a 1000-frame long video, and want to calculate the mean value every 10th frame

Numpy

```
import numpy as np
x = np.random.randint(0, 255,
  size=(1000, 1024, 1024))
y = x[::10].mean()
```

Runs in 12.8 s

Dask

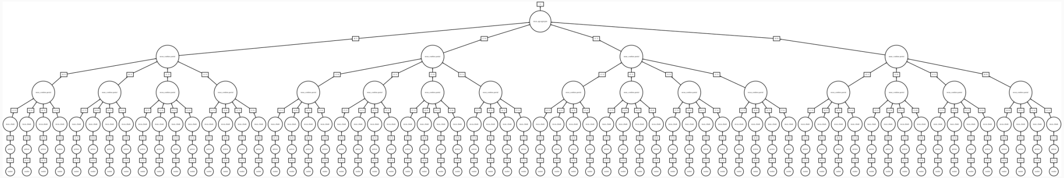
```
import numpy as np
import dask.array as da

#  1 billion elements
x = da.random.randint(0, 255,
  size=(1000, 1024, 1024),
  chunks=(100, 64, 64))
y = x[::10].mean()
```

Runs in 0.55 seconds

```
y.compute()
```

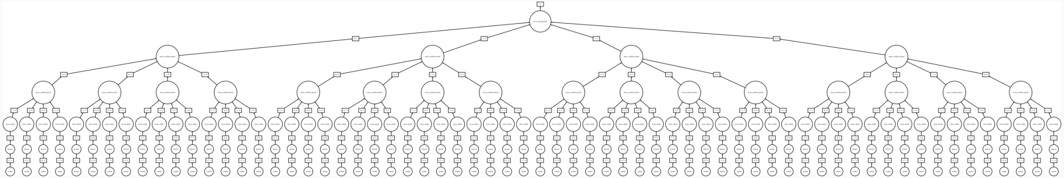
Runs in 3.66 s

What's happening under the hood...



No, you're not supposed to be able to read this image...

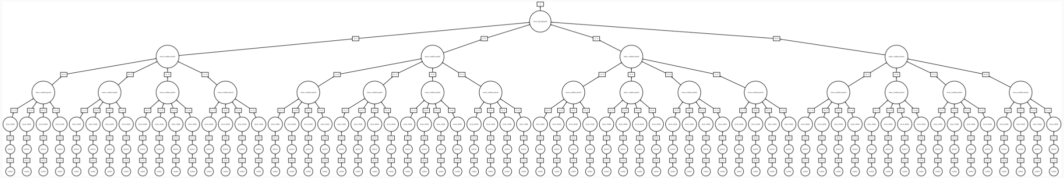
What’s happening under the hood...



No, you’re not supposed to be able to read this image...

- Dask is splitting your task into many subtasks
    - Generate a subset of the random numbers
    - Calculate partial means
    - Pulling those means together
    - Pulling those new means together
    - ...
    - Get the final value!

What's happening under the hood...



No, you're not supposed to be able to read this image...

- Dask is splitting your task into many subtasks
  - Generate a subset of the random numbers
  - Calculate partial means
  - Pulling those means together
  - Pulling those new means together
  - ...
  - Get the final value!
- Subtasks can run on different CPUs/GPUs, even on different computers!
- Subtasks will run from bottom to top, but may run in any left/right order.
- These are simpler and faster to handle and fit into memory

Delayed loading of images with Dask

Let's delay imread

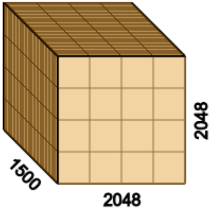
```
from skimage.io import imread
from dask import delayed

# 1500 images of a 2048x2048 movie
filenames = [f"images/image{i:04d}.tif"
   for i in range(1500)]
delayed_img = [delayed(imread)(fn)
   for fn in filenames]
dask_arrays = [da.from_delayed(im,
   shape=(2048, 2048), dtype=np.uint8)
   for im in delayed_img]
```

Delayed loading of images with Dask

Let’s delay imread

```
from skimage.io import imread
from dask import delayed

# 1500 images of a 2048x2048 movie
filenames = [f"images/image{i:04d}.tif"
   for i in range(1500)]
delayed_img = [delayed(imread)(fn)
   for fn in filenames]
dask_arrays = [da.from_delayed(im,
   shape=(2048, 2048), dtype=np.uint8)
   for im in delayed_img]
stack = da.stack(dask_arrays, axis=0)
# We can also modify chunk size
stack = stack.rechunk((10, 512, 512))
```

	Array	Chunk
Bytes	50.33 GB	20.97 MB
Shape	(1500, 2048, 2048)	(10, 512, 512)
Count	2400 Tasks	2400 Chunks
Type	float64	numpy.ndarray



I/O on HDF5 files with Dask

You have an HDF file with the following structure

```
|
|-> Control |-> Experiment 1
|           |-> Experiment 2
|           |-> ...
|
|-> Treatment |-> Experiment 1
              |-> Experiment 2
              |-> ...
```

I/O on HDF5 files with Dask

You can read all the experiments using

```
import numpy as np
import dask.array as da
controls = [h5py.File("experiments.h5", "r")[f"/Control/Experiment i"]
   for i in np.arange(1, 10)]
control_data = [da.from_array(c, chunks=(10, 128, 128))
   for c in controls]
```

All experiments are loaded lazily, so this takes only a few ms to run.
Data can be accessed rapidly and in parallel through dask only when needed.

Summary

- We have only seen the tip of the iceberg.
- Big datasets are becoming ever more common and easy to access
- Think in advance about the challenges these data present and the strategies to overcome them
- Parallel processing can help greatly speed up your computations (even more when using GPUs)
- Chunked storage and processing of big arrays (or other data types), e.g. using dask, is an efficient way of dealing with large datasets