

ARCHER2 for Data Scientists

Practical 3: Dask

Adrian Jackson

1 Introduction

In this exercise we are going to try using Dask to parallelise some Python programs

2 Dask

Recall that there is an optimised Python module available on ARCHER2:

```
module load cray-python
```

Now try running the following Python using dask:

```
import dask.array as da
x = da.random.random((10000, 10000), chunks=(1000, 1000))
print(x)
print(x.compute())
print(x.sum())
print(x.sum().compute())
```

This should demonstrate that dask is both straight forward to implement simple parallelism, but also lazy in that it does not compute anything until you force it to with the `.compute()` function.

You can also try out dask DataFrames, using the following code:

```
import dask.dataframe as dd
df = dd.read_csv('surveys.csv')
df.head()
df.tail()
df.weight.max().visualize()
```

The `visualize()` function call may not work for you if you do not have the `graphviz` or `dot` libraries installed, do not worry about this if it does not work.

You can try using different block sizes when reading in the csv file, and then undertaking an operation on the data, as follows:

```
df = dd.read_csv('surveys.csv', blocksize="10000")
df.weight.max().compute()
```

Experiment with varying block sizes, although you should be aware that making your block size too small is likely to cause poor performance (the block size affects the number of bytes read in at each operation).

You can also experiment with Dask Bags to see how that functionality works:

```
import dask.bag as db
from operator import add
b = db.from_sequence([1, 2, 3, 4, 5], npartitions=2)
print(b.compute())
```

3 Dask Delayed

Dask delayed lets you construct your own task graphs/parallelism from Python functions. Try parallelising the code below using the `.delayed` function or the `@delayed` decorator:

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

def double(x):
    return x * 2

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

data = [1, 2, 3, 4, 5]

output = []
for x in data:
    a = inc(x)
    b = double(x)
    c = add(a, b)
    output.append(c)

total = sum(output)
print(total)
```

If you are struggling with this task you can ask for help, and there are example solutions available in the repository for this course.

4 Mandelbrot Exercise

The code below calculates the members of a Mandelbrot set using Python functions:

```
import sys
import time
import numpy as np
import matplotlib.pyplot as plt

def mandelbrot(h, w, maxit=20, r=2):

    """Returns an image of the Mandelbrot fractal of size (h,w)."""
    start = time.time()
```

```

x = np.linspace(-2.5, 1.5, 4*h+1)
y = np.linspace(-1.5, 1.5, 3*w+1)
A, B = np.meshgrid(x, y)
C = A + B*1j
z = np.zeros_like(C)
divtime = maxit + np.zeros(z.shape, dtype=int)

for i in range(maxit):
    z = z**2 + C

    diverge = abs(z) > r                                # who is diverging
    div_now = diverge & (divtime == maxit)              # who is diverging now
    divtime[div_now] = i                                # note when
    z[diverge] = r                                       # avoid diverging too much

end = time.time()

return divtime, end-start

h = 2000
w = 2000

mandelbrot_space, time = mandelbrot(h, w)

plt.imshow(mandelbrot_space)

print(time)

```

Your task is to parallelise this code using Dask Array functionality. Using the base python code above, extend it with Dask Array for the main arrays in the computation. Remember you need to specify a chunk size with Dask Arrays, and you will also need to call `compute` at some point to force Dask to actually undertake the computation. Note, depending on where you run this you may not see any actual speed up of the computation. You need access to extra resources (compute cores) for the calculation to go faster. If in doubt, submit a python script of your solution to the ARCHER2 compute nodes to see if you see speed up there.