ARCHER2 for Data Scientists

Practical 4: 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 blocksizes 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 blocksizes, although you should be aware that making your block size too small is likely to cause poor performance (the blocksize 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.

If you are struggling with this parallelisation exercise, there is a solution available for you in the `dask-mandel.py` file.

5 Pi Exercise

The code below calculates Pi using a function that can split it up into chunks and calculate each chunk separately. Currently it uses a single chunk to produce the final value of Pi, but that can be changed by calling `pi_chunk` multiple times with different inputs. This is not necessarily the most efficient method for calculating Pi in serial, but it does enable parallelisation of the calculation of Pi using multiple copies of `pi_chunk` called simultaneously.

```
import time
import sys

# Calculate pi in chunks
# n     - total number of steps to be undertaken across all chunks
# lower - the lowest number of this chunk
# upper - the upper limit of this chunk such that i < upper
def pi_chunk(n, lower, upper):
    step = 1.0 / n
    p = step * sum(4.0/(1.0 + ((i + 0.5) * (i + 0.5) * step * step)) for i
in range(lower, upper))
```

```
    return p


# Number of slices
num_steps = 10000000

print("Calculating PI using:\n  " + str(num_steps) + " slices")

start = time.time()

# Calculate using a single chunk containing all steps
p = pi_chunk(num_steps, 1, num_steps)

stop = time.time()

print("Obtained value of Pi: " + str(p))
print("Time taken: " + str(stop - start) + " seconds")
```

For this exercise, your task is to implemented the above code on ARCHER2, and then parallelise using Dask. There are a number of different ways you could parallelise this using Dask, but we suggest using the Futures `map` functionality to run the `pi_chunk` function on a range of different inputs. Futures map has the following definition:

```
Client.map(func, *iterables[, key, workers, ...])
```

Where `func` is the function you want to run, and then the subsequent arguments are inputs to that function. To utilise this for the Pi calculation, you will first need to setup and configure a Dask `Client` to use, and also create and populate lists or vectors of inputs to be passed to the `pi_chunk` function for each function run that Dask launches.

If you run Dask with processes then it is possible that you will get errors about forking processes, such as these:

```
    An attempt has been made to start a new process before the
    current process has finished its bootstrapping phase.

    This probably means that you are not using fork to start your
    child processes and you have forgotten to use the proper idiom
    in the main module:
…
```

In that case you need to encapsulate your code within a main function, using something like this:

```
if __name__ == "__main__":
…
```

If you are struggling with this exercise then there is a solution available for you in the `dask-pi.py` file.