0099_dask_delayed

November 25, 2018

1 Parallel processing functions and loops with dask 'delayed' method

For a full SciPy conference video on dask see: https://www.youtube.com/watch?v=mqdglv9GnM8

Dask is a Python library that allows parts of program to run in parallel in separate cpu threads to speed up the program.

Here we will look at using dask to run a normal function in parallel when we need to call the function more than once in one part of a program. We will mimic a slow function by using the Python sleep() method to make the function take on second each time it is run. Normally it would take 3 seconds to run this function 3 times, but here we will see that with dask all three calls to the function will be complete in one second (assuming you have at least a dual core, 4-thread cpu).

We will first import our required libraries.

```
In [1]: from time import time # to time the program
        from time import sleep # to mimic a slow function
        from dask import delayed # to allow parallel computation
```

Next we define a normal function (there is no use of dask at this this point). Here we will write a function that returns the square of the number passed to the function, but we'll add a 1 second sleep in it to mimic a longer running function.

```
In [2]: # Define a function normally
        def my_function(x):
            # mimic a slow function with sleep for 1 secons
            sleep(1)
            return x*2
```

Now we will call that function three times.

Normally this would take three seconds as each function must complete before the next one can start. But by using the decorator 'delayed' we mark this as a function call that may be run in parallel with others.

Note the syntax amendment. We would normally call this function with my_function(x), but we amend the syntax to delayed(my_function)(x).

We then calculate the sum of the three returned numbers from our function. But when using dask this does not actually give us our answer. If we print the type of this object we see that it is a 'delayed' object. To get the actual result we must then use the .compute() method as shown below.

Then we see how long these three 1 second function calls take. If you have a processor with at least 2 CPUS and 4 threads you should see it takes close to one second rather than three!

```
In [3]: # Record time at start of run
        start = time()

        # Run function in parallel when calling three times
        # Syntax of my_function(x) is replaced with delayed(my_function)(x)
        a = delayed(my_function)(1)
        b = delayed(my_function)(2)
        c = delayed(my_function)(3)

        # Total will sum results. But at this point we generate a 'delayed' object
        total = a + b + c

        # Show object type of total
        print ('Object type of total', type(total))

        # To get the result we must use 'compute':
        final_result = total.compute()

        # Calculate time taken and print results
        time_taken = time()-start
        print ('Process took %0.2f seconds' %time_taken)
        print('Final result ',final_result)
```

```
Object type of total <class 'dask.delayed.Delayed'>
Process took 1.01 seconds
Final result  12
```

1.1 Using dask 'delayed' in a loop

We can also use dask delayed to parallel process data in a loop (so long as an interation of the loop does not depend on previous results). Here we will call our function 10 times in a loop. Note the use of .compute again to get the actual result. This would take 10 seconds without dask. On a 4-cour/8-thread CPU it takes two seconds!

```
In [4]: start = time()

        # Example loop will add results to a list and calulate total
        results = []
        for i in range(10):
            # Call normal function with dask delayed decorator
            x = delayed(my_function)(i)
            results.append(x)

        total = sum(results)
        final_result = total.compute()

        # Calculate time taken and print results
        time_taken = time()-start
        print ('Process took %0.2f seconds' %time_taken)
        print('Final result ',final_result)
```

```
Process took 2.01 seconds
Final result  90
```