ENGSCI 233 - Lab Worksheet

Performance

Date: May 3, 2022

Lab Introduction

Your code may not have many bugs, but if it's *too slow* to finish the job in a reasonable timeframe, why bother?

Your objective in this lab is to implement techniques for code profiling and optimisation. You will measure algorithm scaling, identify and eliminate runtime bottlenecks, make use of compiled language to run slow problems, and put your idle CPUs to work through parallelisation.

The following files have been made available to you:

- *functions_perflab.py*
- *task1_perflab.py*
- *task2_perflab.py*
- *task3_perflab.py*

Any classes or functions that you write/complete should include a suitable docstring. Note that writing test functions is entirely optional in this lab, but hopefully you will still write them where appropriate.

Task 1: Profiling

Background

The provided function `def matmul1` will calculate the matrix multiplication of two input matrices, A and B. The algorithm used in this function is commonly referred to as *naive matrix multiplication* i.e. the calculation is done in its simplest form, through a series of nested for loops.

The provided function `def multiply_square_matrices` will create a matrix A and a matrix B, both square matrices of identical order that are filled with uniformly distributed random numbers between -1 and 1. It will then call a matrix multiplication function, such as `matmul1`, to perform the calculation. It is similar to a *void function* in C i.e. it doesn't return anything. It is designed only to spend time multiplying matrices.

In this task, you will be using the Python profiling library `cProfile` to profile `multiply_square_matrices`.

Exercises

Complete the following exercises for this task:

1. Run the code provided in *task1_perflab.py*. This will profile the function call:

   ```
   multiply_square_matrices(10, 50)
   ```

   which performs ten separate matrix multiplication calculations on square matrices of order 50×50. The provided code will then print the run time results. Use these results to answer the following questions:

   (a) What is the main bottleneck in running `multiply_square_matrices`?

   (b) What should α be in $\mathcal{O}(n^{\alpha})$ for the naive matrix multiplication implemented in `matmul1`, where n is the number of matrix rows/columns? Justify your answer mathematically.

   If unsure, you could try profiling a few different calls to the function and get a sense of the performance scaling.

2. Complete the function `def profile_matmul` in *functions_perflab.py*. This function should take three inputs:

   - The integer number of matrix multiplications to perform.
   - A list of different problem sizes, n, to profile the matrix multiplication function with. Each n is equivalent to the number of rows/columns of A and B.
   - The matrix multiplication function to be profiled.

   This function should call `multiply_square_matrices` for each problem size, n_i, and use `cProfile` to find the total time taken. The function should return a list containing the total time taken for each problem size.

3. Complete the function `def plot_polynomial_performance` in *functions_perflab.py*.

   This function should take two inputs, a list of run times and a list of problem sizes. It should plot the logarithm of the run time against the logarithm of the problem size, using markers rather than lines to indicate individual data points.

   Use the NumPy function `np.polyfit` to fit an order 1 polynomial to the log-log data. Add this fitted polynomial as a line to your existing plot. One of the two fitted polynomial coefficients will be equivalent to α in $\mathcal{O}\left(n^{\alpha}\right)$ as measured for the algorithm - add this value to a legend in your plot.

Task 2: Optimisation

Background

In order for your computer to run Python code, an *interpreter* must convert each Python instruction into a lower level language that the CPU can understand (e.g. machine code). The CPU then runs the translated instruction and the interpreter can translate the next instruction. This process continues until either all instructions are finished, or there is an error.

If we were instead using C, C++ or Fortran to write our code, a *compiler* would translate the whole set of code into a lower level language ahead of time. This compilation process typically creates an executable in the process. As all translation has already been done, the CPU can work through the instructions without significant delay. It is for this reason you will often hear statements such as *"interpreted languages are slow, compiled languages are fast"*.

Python is generally very slow at executing many loops, as each one must be translated for the CPU. Fortunately, NumPy gives us access to pre-compiled code that can perform some calculations much faster than using a loop. Consider the following matrix multiplication, $A\,B = C$:

$$\begin{bmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{bmatrix} \begin{bmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{bmatrix} = \begin{bmatrix} a_{0,0}\,b_{0,0} + a_{0,1}\,b_{1,0} & a_{0,0}\,b_{0,1} + a_{0,1}\,b_{1,1} \\ a_{1,0}\,b_{0,0} + a_{1,1}\,b_{1,0} & a_{1,0}\,b_{0,1} + a_{1,1}\,b_{1,1} \end{bmatrix}$$

Notice that an element $c_{i,j}$ is equal to the dot product between row i of A and column j of B. The NumPy function `np.dot` utilises pre-compiled code and can be used to perform this calculation potentially much faster than performing a large number of iterations.

Exercises

Complete the following exercises for this task:

1. Complete the function `def matmul2` in *functions_perflab.py*. It should follow a similar approach to `matmul1`, but use `np.dot` to eliminate the need for the final for loop. This should result in a good performance improvement.

2. Write code in *task2_perflab.py* that utilises your `profile_matmul` and `plot_polynomial_algorithm` from Task 1 to profile the following matrix multiplication functions:

   - The provided `matmul1`
   - Your completed `matmul2`
   - The NumPy `np.matmul`

   It is recommended that you perform 10 matrix multiplications, with a range of problem sizes (i.e. rows/columns of A and B) equal to 2^n where $n = 3, 4, \ldots, 7$.

   Use these results to answer the following questions:

   (a) Rank the three matrix multiplication function in terms of their performance. Explain why these rankings make sense.

   (b) Find the approximate relationship between run-time and problem size for each of your algorithm profiling results. Are these equivalent to the Big-O notation for performance? It may be helpful when answering to read up on the Big-O performance of current matrix multiplication algorithms.

Task 3: Parallel Implementation

Background

One way of improving the performance would be to use a more computationally efficient algorithm, one that has better scaling with the problem size. One of the first better than naive matrix multiplication algorithms developed was the Strassen algorithm (1969) with $\mathcal{O}\left(n^{2.8074}\right)$ (under certain conditions). However, this can be quite difficult to implement at this level of education/experience, so instead we will be looking for an easier way to speed up the process of multiplying matrices.

Most modern computers contain more than one CPU. By default, our Python code will only be run on a single CPU. The `multiprocessing` library supports the spawning of processes (e.g. running a function), which can be distributed out to a pool of available CPUs. Our aim in this task is to divide out a large number of matrix multiplications to multiple CPUs. We can time how long the processes take and use these results to calculate the parallel speed-up and efficiency achieved. Since each student will have a different hardware configuration, your results may vary!

Exercises

Complete the following exercises for this task:

1. Write appropriate docstrings and internal comments for the functions:

   - `def time_serial_multiply_square_matrices`
   - `def time_parallel_multiply_square_matrices`

   I had originally intended for you to write your own versions of these two functions, but the parallel one in particular was somewhat unintuitive to write. By documenting the provided functions, you should develop a good understanding of how they work.

2. Run the code provided in *task3_perflab.py*, which uses the provided functions to determine the run time for both serial and parallel execution of many matrix multiplications. You may want to tweak the following values to better suit your own personal computer:

   - `multiplications` - total number of matrix multiplications to perform.
   - `n` - problem size i.e. number of rows/columns in A and B.
   - `matmul` - the matrix multiplication function to use. By default we could just use `matmul1`.
   - `verbose` - display recorded times to screen, may be useful for debugging.

   Note that if you are having difficulty running this code on Windows, perhaps try the following replacement:

   ```
   with multiprocessing.Pool(ncpu) as pool:
   ```

to

```
with multiprocessing.Pool(ncpu, limit_cpu) as pool:
```

This attempts to schedule the processes with a less restrictive condition on CPU priority. I personally was having difficulties getting `multiprocessing` to schedule and execute my processes. If it's a complete disaster for some students, we can provide an exemplar set of data to continue the next exercise with.

Once you are happy with a set of timing results (ideally you want results that show some improvement from moving from serial to parallel, though not necessarily improved speedup every time the number of CPUs increases), you can hardcode these for use in the next exercise (you don't need to run them again, it can be very time-consuming).

3. Complete the function `def plot_runtime_ncpu`.

   This function should take two inputs, a list of run times and a list of problem sizes. It should plot the parallel speedup and parallel efficiency as two subplots on a single figure.

4. Answer the following questions:

   (a) Would you consider your parallel speedup to be sublinear or superlinear? Briefly explain your answer.

   (b) Why might the speedup get worse as the pool size increases? If not sure, try running the timing functions with a much smaller number of matrix multiplications to see what happens to the run time between serial and parallel.

Submission Instructions

For this lab, you should submit the following:

- *functions_perflab.py*
- *task1_perflab.py*
- *task2_perflab.py*
- *task3_perflab.py*

- A PDF or Word file containing your answers to the Task 1, 2 and 3 questions. These should be supported by relevant plots produced by the plotting functions you were instructed to complete.

You do **not** need to submit any other files. Do **not** modify the file names, but don't worry if Canvas appends a `-1` or `-2` to the end.

Remember, all submissions are compared against each other and those from previous years. Copying someone else's code and changing the variable names constitutes academic misconduct by *both* parties.