William Clift
Data Structures
March 2020

Matrix Multiplication Assignment

Concepts

The program is designed to compare the operation of matrix multiplication on randomly generated matricies in both single and multi-threaded functions. Using the `MatrixGenerate` class, we create square matricies and then multiply them in our function while keeping track of the runtime. This requires calculating the dot product of each row and column and putting that result in the corresponding cell of the product matrix. For the naive approach, we iterate through each row and column several times to calculate each dot product and place it in the resulting matrix. The multi-threaded function splits up each dot product into a separate thread and the executor service manages when threads become available to speedup the process. Below is our algorithm implementation in pseudocode.

Implementation

The implementation of the program used the format of the following pseudocode.

`multiply()` (Single-Threaded)

```
    //multiply square matricies 'a' and 'b'
    n is the length of a's rows
    m is the length of b's rows
    p is the length of b's columns

    // declare a new 'product' 2-Dimensional array (size [n][p])

    for i 0...n
      for j 0...p
        for k 0...m
          sum = sum + (a[i][k] * b[k][j])   //Sum the row of a and column of b and
    multiply
        product[i][j] = sum

    return product
```

`multiplyThreaded()` (Multi-Threaded)

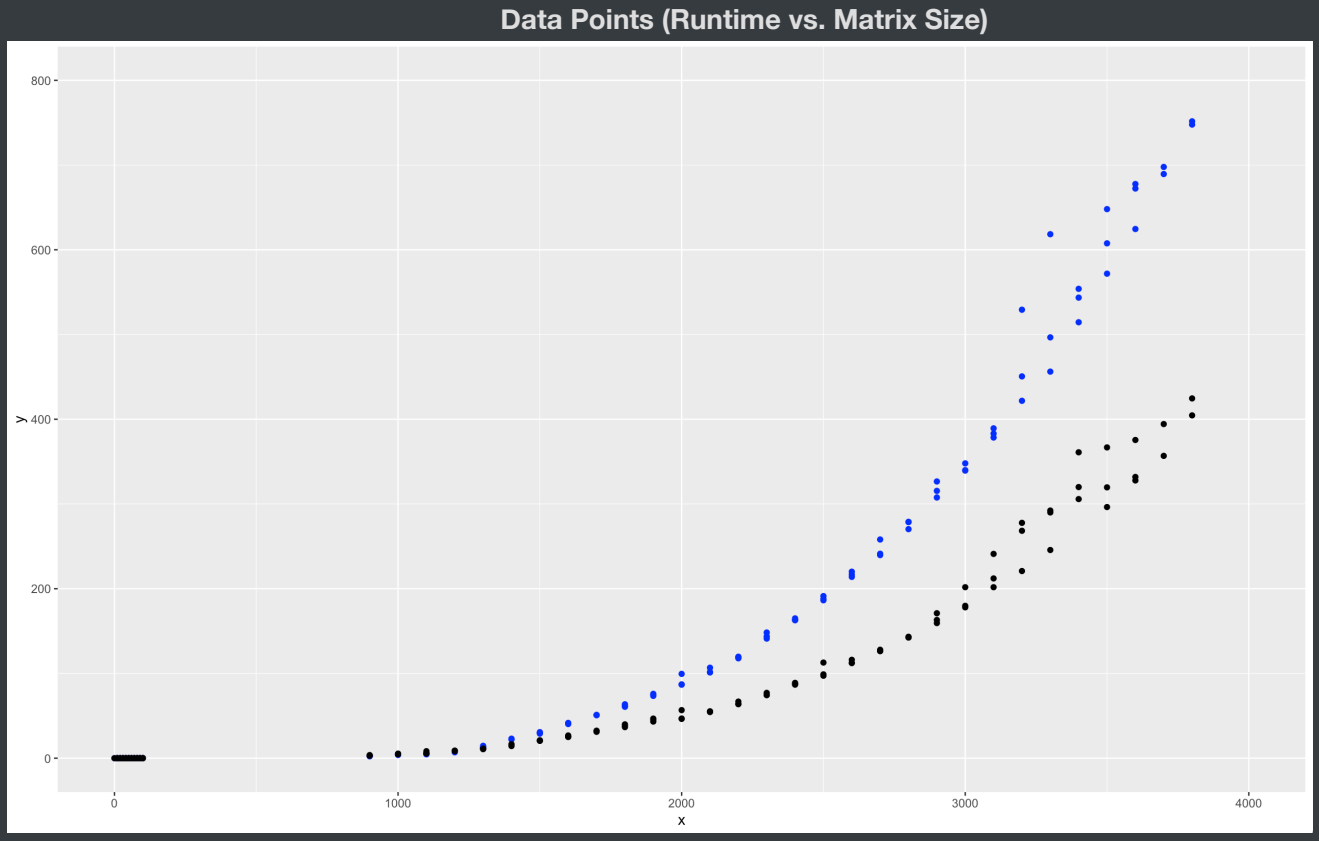
```
//multiply square matricies 'a' and 'b'
// instantiate the executor service (CachedThreadPool)
// declare a new 'product' 2-Dimensional array (size [n][p])

for i 0...n
  for j 0...p
      Runnable thread = new MyMatrixMultiply(a, b, newMatrix, i, j)
      // Calculate the dot product for the rows/columsn of newMatrix[i][j]

return newMatrix
```

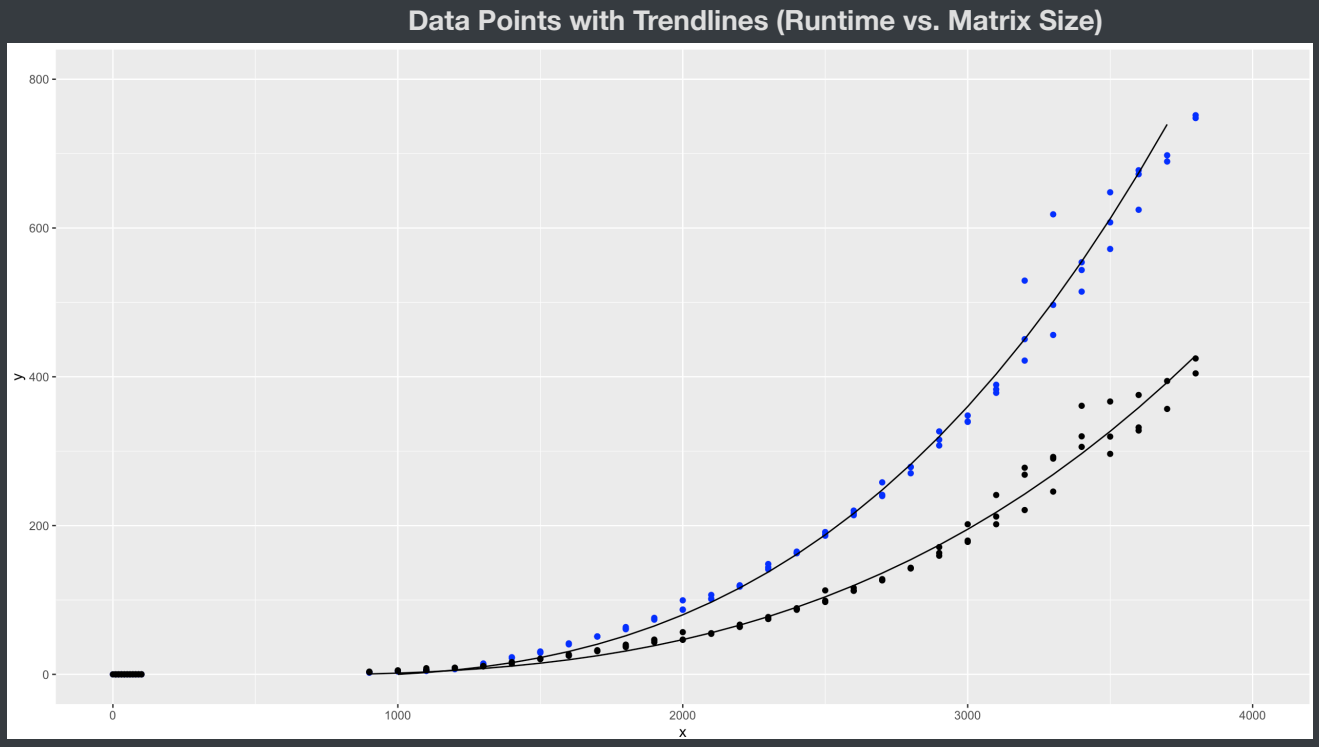
Empirical Analaysis

Shown on the graph below, we have the number of rows and columns in a randomly generated square matrix on the x-axis and the runtime as measured in seconds on the y-axis. The blue data-points refer to testing on the single-threaded `multiply()` program function, and the black data-points are from testing on the multi-threaded `multiplyThreaded()` program function.
Note: to generate the matricies, the multi-threaded `generateRandomMatrixThreaded(int n, int m)` was used in each and all of the tests. These graphs were made using R.

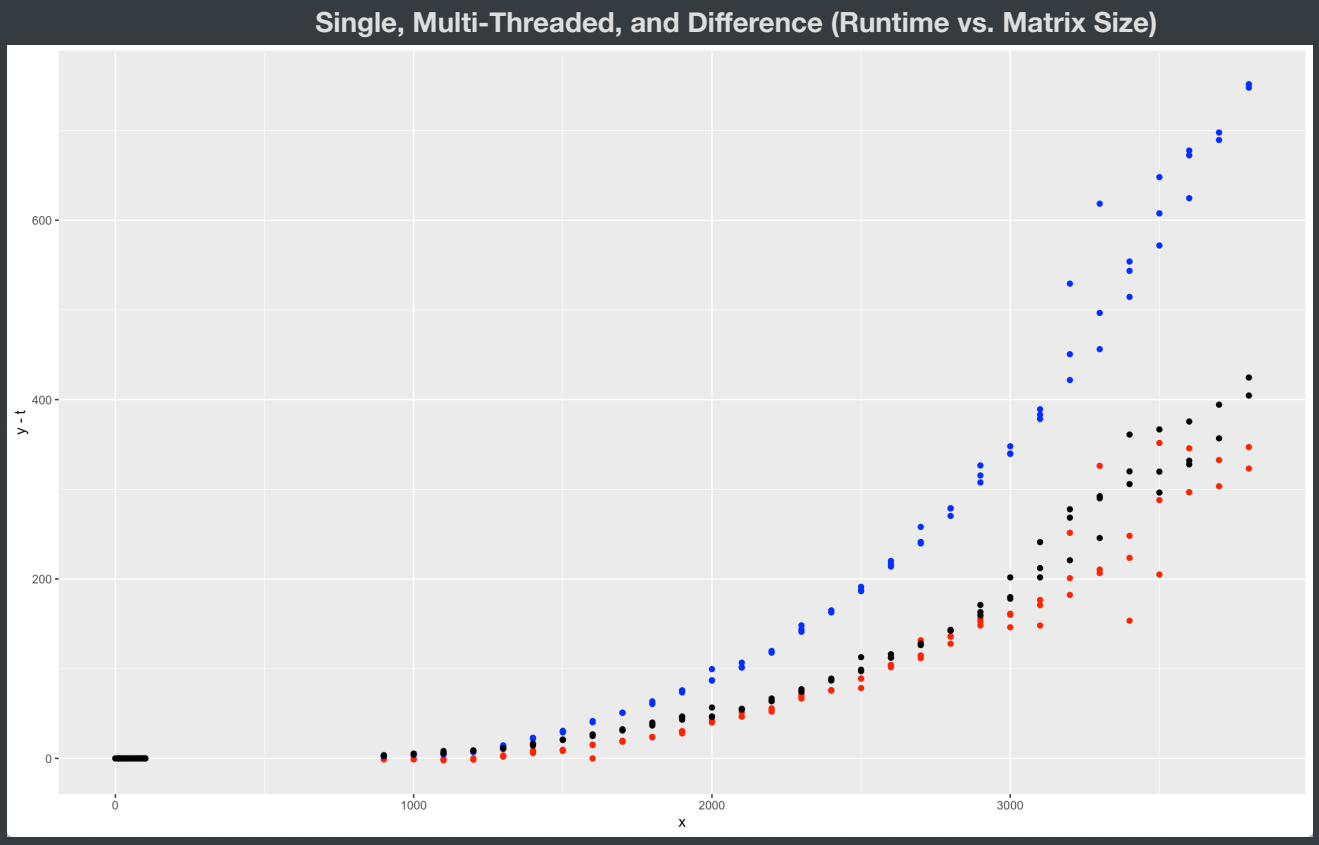


And here, once we add the regression lines, we see that these two processes both follow cubic function lines.
In blue, for the single-threaded program, we describe these data using approximately
$y = x^3/50000000 - x^2/50000$
and in black, for the multi-threaded program, we descibe these data using approximately
$y = x^3/100000000 - x^2/120000$.

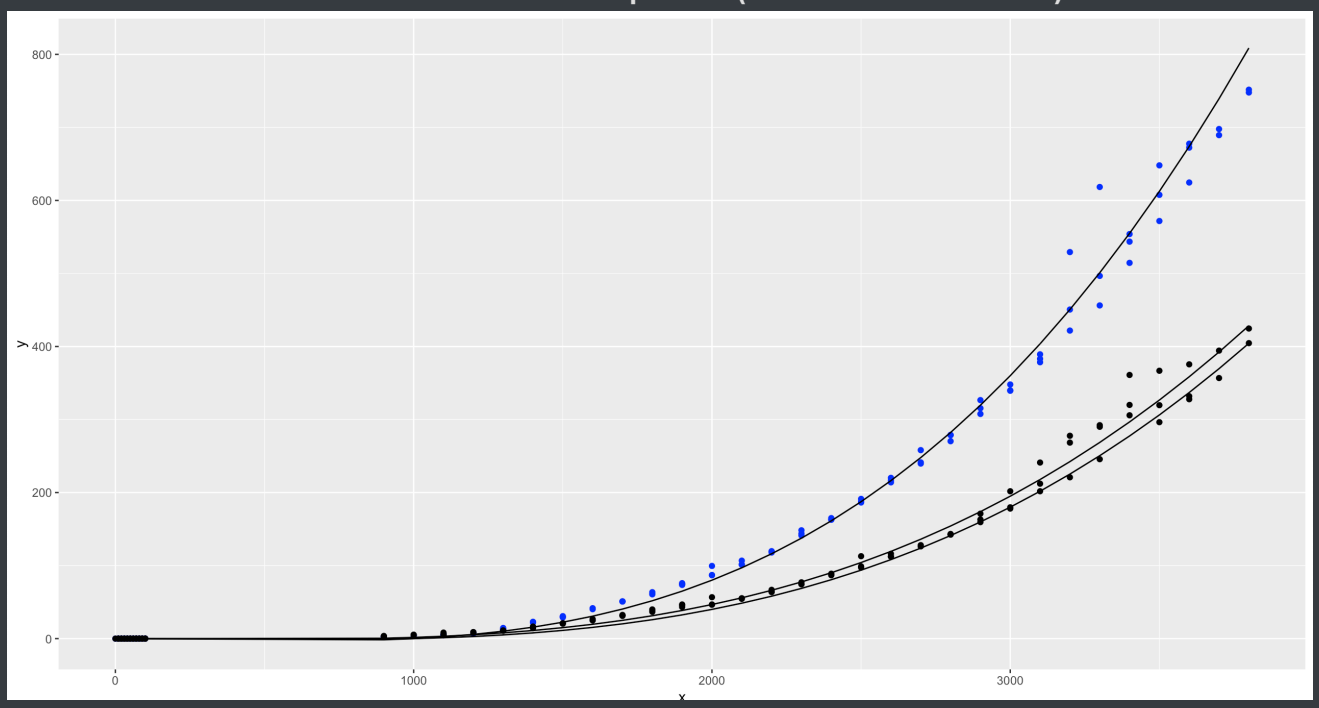
But, it is clear that while they follow different functions, the algorithm for both the single and multi-threaded are cubic functions. If we were to have m number of processors in a machine running this program, we would have a coefficient of $\frac{1}{m}$ that divides the equation for multi-threaded. As the number of processors approaches N, we would get $O(N^2)$ as the runtime. However, the machine used to do this analysis has 4 cores and thus a constant $\frac{1}{4}$ coefficient, which is ignored in the "Big-O" notation. This is why we see no overall improvement on the scaling of the program. Thus, we have $O(N^3)$ for both single and multi-threaded. Below are the data with cubic trendlines.



We can add the difference between the single-threaded (blue) and the multi-threaded (black) to get new data points (below in red). Notice that these red points overlap quite a bit with the multi-threaded points (black). Mathematically, this might indicate that the trend from single to multi-threaded is twice as fast.



Let us experiment by taking the trendline of the single-threaded function (blue) and dividing it in half to create a new line. If the multi-threaded function is actually twice as fast as single threaded, then this new line should approximate the ulti-threaded data points (black). Let us add the line which divides the blue trendline in half.



As we can see, this is a pretty good approximation. In fact, it follows closely to the line we used to approximate the multi-threaded data points to begin with. By this, we can conclude with confidence that the multi-threaded algorithm gives us a speed up by a factor of 2.

Conclusion

Through the development of this program, we observed the benefits but also the challenges of multi-threading parrallelism. At first, I had trouble understanding and implementing the executor service procedure to incorporate the multi-threading into my program. The algorithm to complete this involved `MyMatrixMultiply()` using the executor service to startup a thread and call `run()` which calls on the `dotProduct()` function. Once successfully done, we found that with small matricies, we actually get a slow down from single to multiple threads. But, at around 1000 rows and 1000 columns, the parallel algorithm starts to perform much faster. We found that this speedup is approximately doubled. with large matricies. However, our runtime analysis gave us $O(N^3)$ for both algorithms.