Homework 3

Adam Karl

April 29, 2021

1 Matrix Multiplication - Introduction

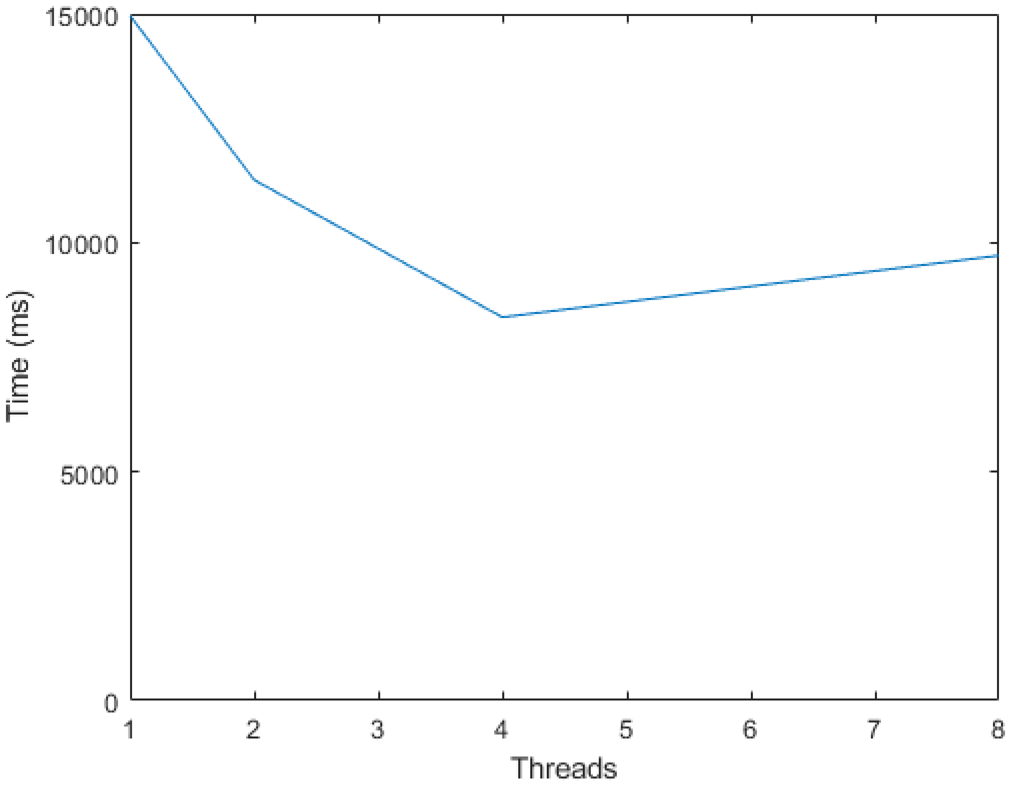
Matrix Multiplication is an essential function for any computing system. From solving systems of equations to network theory to computing graphics, many computational systems require a fast way to multiply matrices, and therefore all modern computers require a quick way to do the calculations.

A serial approach to matrix multiplication requires three nested loops: two for the two dimensions of the output matrix, and one to iterate over the row of the multiplicand as well as the column of the multiplier. The sum of the item-wise product of the multiplicand's row and multiplier's column represents a single element in the solution matrix.

2 Matrix Multiplication - Solution

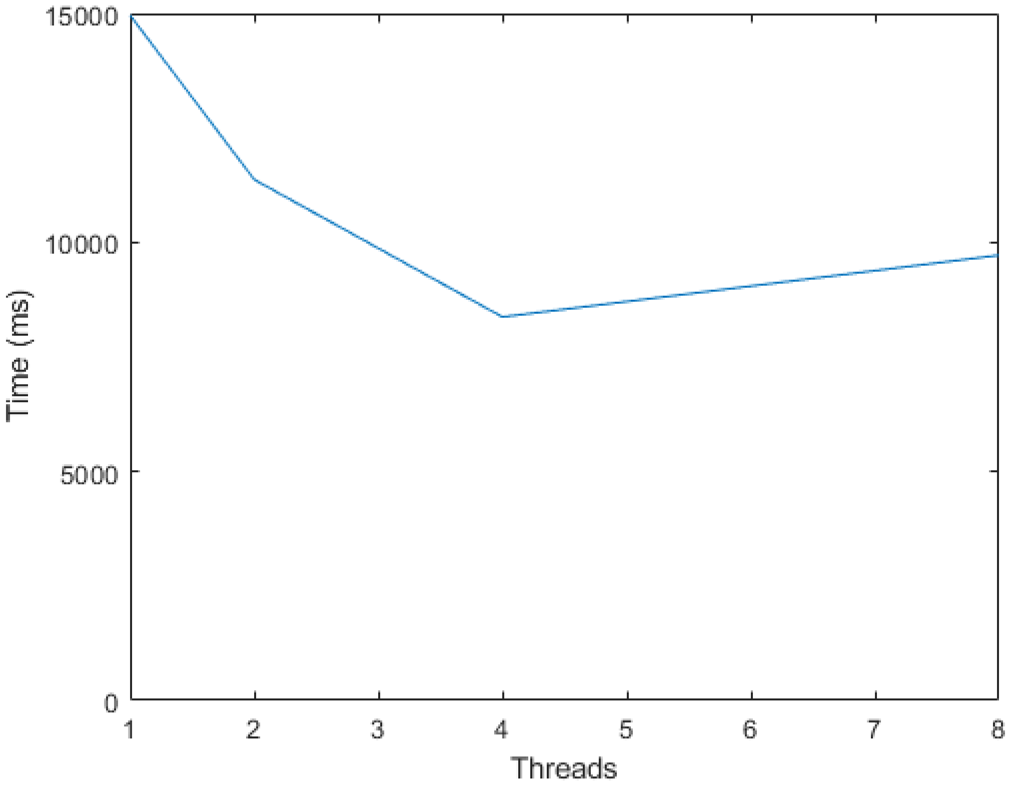
My solution for parallelizing matrix multiplication was simply to divide and conquer, each of k processors computing 1/k of the solution matrix. This solution had a very simple implementation: just divide the solution matrix evenly between the threads (I implemented this by row using the thread id), then have each thread run the same three nested loops as the serial solution to calculate the value of that element in the solution matrix. Since our problem also includes adding matrix B in the calculation, I looked up and added the value from the B matrix inside two of the loops (obviously not inside the element-wise sum of products loop). Since there is no write overlap between the threads, we can write each element to the solution matrix without worry.

3 Matrix Multiplication - Experimental Evaluation



parallel solution

Then, I tried scaling the problem up to 2048x2048 matrices, and plotted the threads vs runtime.



2048x2048 matrices

4 Matrix Multiplication - Conclusions

I managed to achieve significant improvement up to 4 threads, but past 4 cores the runtime increases significantly. Part of this is likely due to the fact that my linux machine only actually has 4 cores, but it's possible that there is significant downtime while different threads are waiting to write to the global variable. Either way, there seem to be diminishing returns when adding threads, and I was only able to (on my machine) reduce the runtime by about half, no matter how many threads I added. With 4 threads, the speedup is 1.79 and the efficiency is 0.44.

When I scaled up the problem, the times increased, but the shape of the graphs stayed about the same, indicating that my multithreaded runtime grows at the same rate as the sequential solution. With 4 threads, the speedup is 1.76 and the efficiency is 0.44.

5 Softmax - Introduction

The Softmax function transforms a vector of real numbers into a normalized probability distribution, all with values between 0 and 1, that sums to 1. Many neural networks work with normalized distributions and use the softmax function to normalize data for them. Normalized distributions also have many uses in data science and statistics.

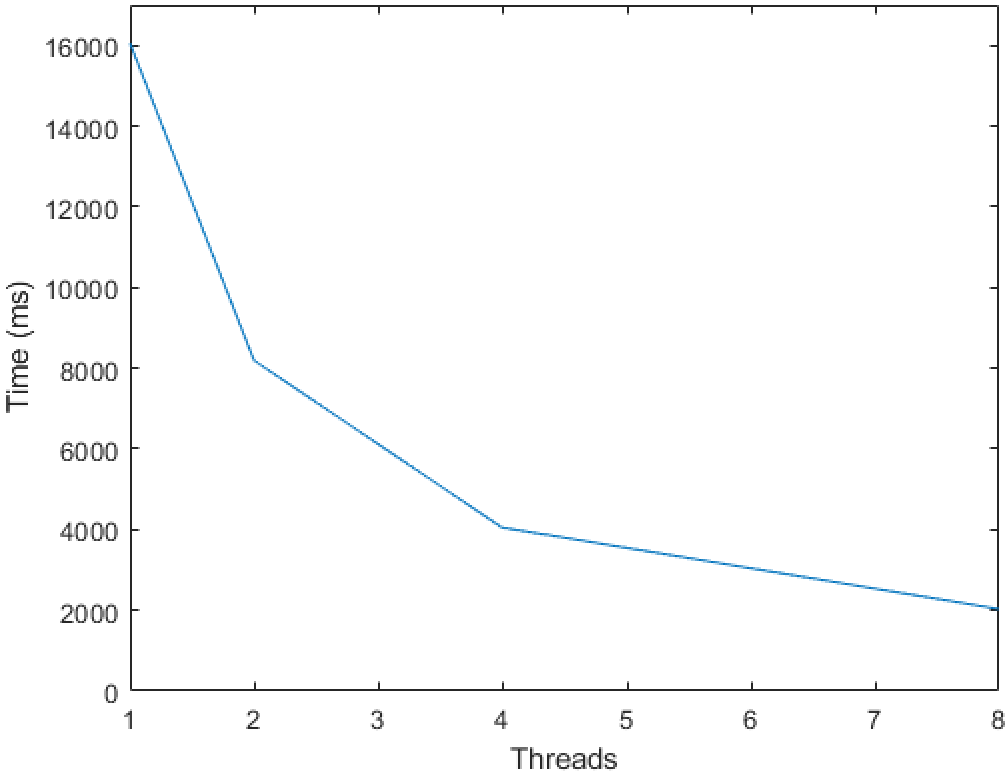
The serial softmax function iterates over the array to find the sum of their exponentials, then iterates over it again to divide each element by that sum. It is a linear algorithm.

6 Softmax - Solution

I initially tried to optimize the Softmax portion of the problem by splitting up both the exponentials loop and the corrections loop evenly between all k threads. Thread 0 would deal with indices 0 through N/k-1, thread 1 would deal with indices N/k through 2N/k-1, etc. However, when I tested my program with this solution, I found that there was no discernible improvement. Every time the program is run, there's a certain range for the runtime, and it was a 50-50 whether my multithreaded approach would even beat the unmodified program. Therefore, I decided to scrap my progress and focus on the matrix multiplication and evaluation portion of the program.

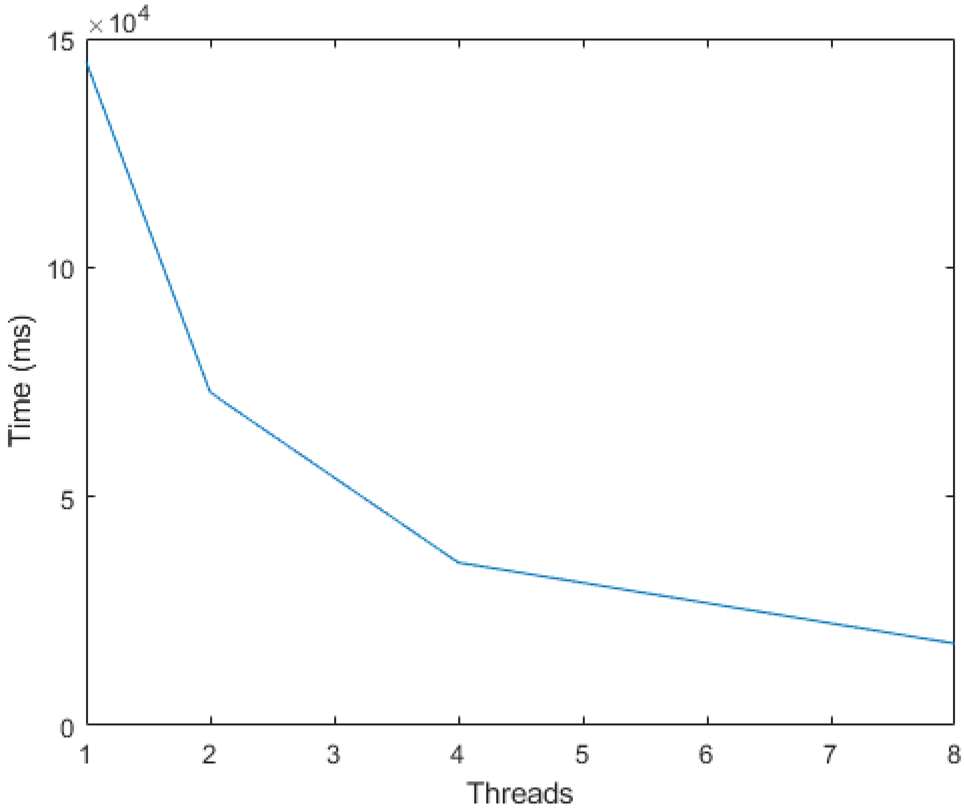
By implementing a solution very similar to the matrix multiplication problem, I was able to significantly speed the program up. With k threads, thread i would be responsible for elements i, i+k, i+2k, i+3k ... within the range of the output array Y. By doing this, the work was evenly spread around all the threads in order to improve the runtime the most.

7 Softmax - Experimental Evaluation



parallel solution

Then, I tried scaling the problem up to 2048x2048 matrices, and plotted the threads vs runtime.



2048x2048 matrices

8 Softmax - Conclusions

I managed to achieve significant improvement up to 8 threads. With 4 threads, the speedup is 3.96 and the efficiency is 0.991. With 8 threads, the speedup is 7.90 and the efficiency is 0.988.

Although I did make some significant improvements to the matrix vector multiplication part of the problem, I was not able to find any way to noticeably reduce the runtime of the softmax function. When analyzing the program, it is obvious why this is the case: matrix multiplication has cubic runtime while the softmax function has linear runtime. While I'm sure that it is possible to make the softmax function run slightly faster, any improvement will be dwarfed in comparison by the number of computations necessary for the matrix multiplication portion of the problem. The number of computations necessary for the matrix evaluation portion of the problem is in the billions (cubic with 1000-side matrices), the number of computations for the softmax portion is in the thousands (linear with 2 loops through a 1000-side array).

With this in mind, my submitted code does not have multithreading for the softmax portion of the problem, as that part is insubstantial.

When I scaled up the problem, the times increased, but the shape of the graphs stayed about the same, indicating that my multithreaded runtime grows at the same rate as the sequential solution. With 4 threads, the speedup is 3.99 and the efficiency is 0.9975. With 8 threads, the speedup is 7.67 and the efficiency is 0.959.

9 Trapezoidal Rule - Introduction

In calculus, the Trapezoidal rule is an approximation technique used for calculating the area described by a definite integral. The idea of the Trapezoidal rule is to envision the area under the curve as made up of many small trapezoids, with two corners on the x-axis and the other two corners at the evaluation of the function at the corresponding x-values (these evaluations are also known as Riemann sums).

The trapezoidal approximation does leave some margin for error. If the function has a positive curve such as x^2, then every trapezoid will slightly underestimate the area, and can add up to a significant difference between the approximation and actual solution. A function with a negative curve will have the opposite problem and be overestimated by the trapezoidal rule. However, no matter the case, adding more trapezoids will always improve the approximation, and as the number of sections approaches infinity the approximation will approach the correct solution.

10 Trapezoidal Rule - Solution

While I began implementing the parallel solution, I also cut out a significant amount of duplicate calculations in the program by recognizing that aside from the very first and very last Riemann sum, all other heights factor into the equation twice. Therefore, I simplified the approximation from:

$$\sum_{k=0}^{N-1} \frac{f(x_k) + f(x_{k+1})}{2} \Delta x_k$$

to:

$$\Delta x(\frac{f(x_0)}{2} + f(x_1) + f(x_2) + \cdots + f(x_{N-2}) + f(x_{N-1}) + \frac{f(x_{N-3})}{2})$$

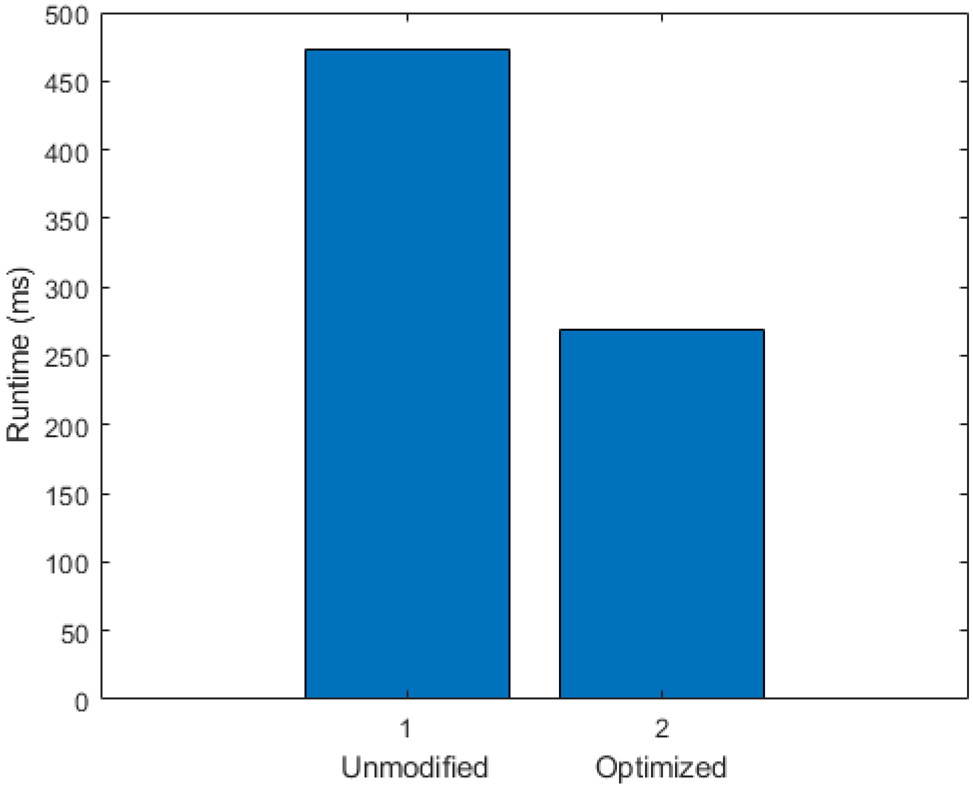
This is much more efficient since now, each height is calculated only once, rather than once for the left trapezoid and once for the right trapezoid. However, I had to put in special stipulations for the leftmost and rightmost Riemann sums, since they should only be calculated once as they only have one trapezoid touching them. It's important that I mention this improvement, as it cuts out almost half of the runtime for the problem on my machine while still returning the same solution, dwarfing the improvements of the parallelization.

In order to parallelize the serial solution, I then divided up the N Riemann sums between the k threads evenly, such that thread 0 will calculate sums 0 through N/k-1, thread 1 will calculate sums N/k through 2N/k-1, and so on, with the last thread calculating sums N-N/k through N. As stated before, since the very first and very last Riemann sums are only counted once in the equation (and by default my program counts each sum twice), I had to implement a check to see if the current thread is the first thread, and if so to subtract the value of the first Riemann sum from the running total. A similar check was added for the last Riemann sum of the last thread.

After calculating the trapezoidal approximations for the x-value range designated to each thread, the thread will add that area to a global area variable that all threads have access to. After all threads add to this variable and subsequently terminate, we will have calculated the Trapezoidal rule approximation of the integral in parallel.

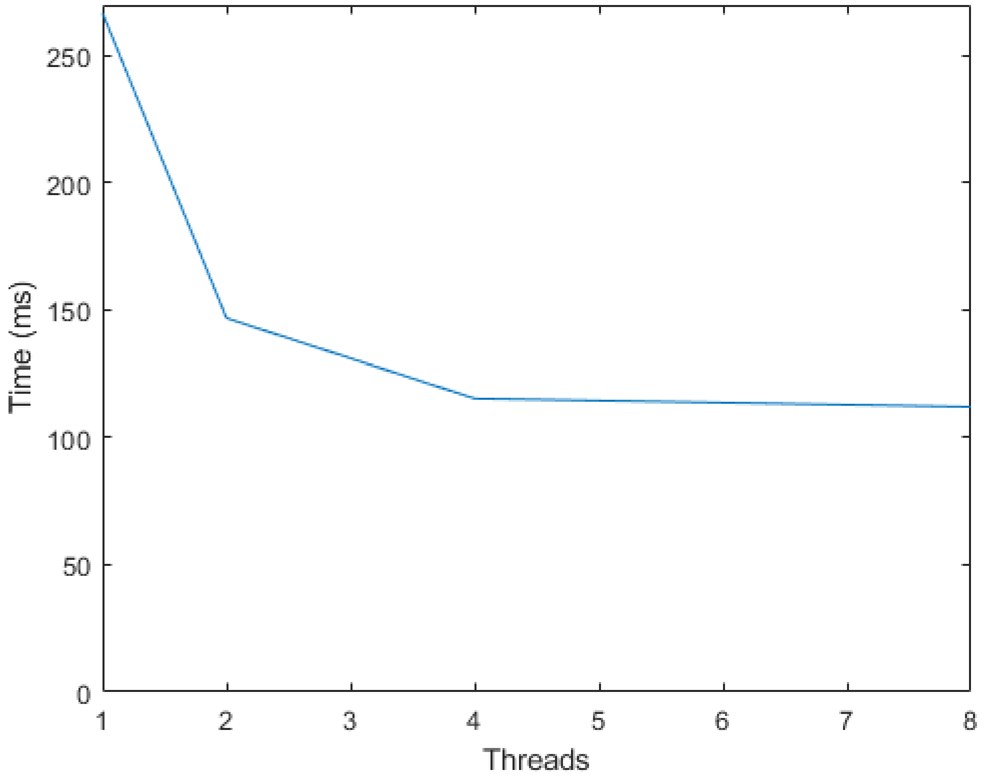
11 Trapezoidal Rule - Experimental Evaluation

My initial modification in order to cut out duplicate Riemann sums saved a significant amount of time, cutting the runtime from 978ms to 268ms.



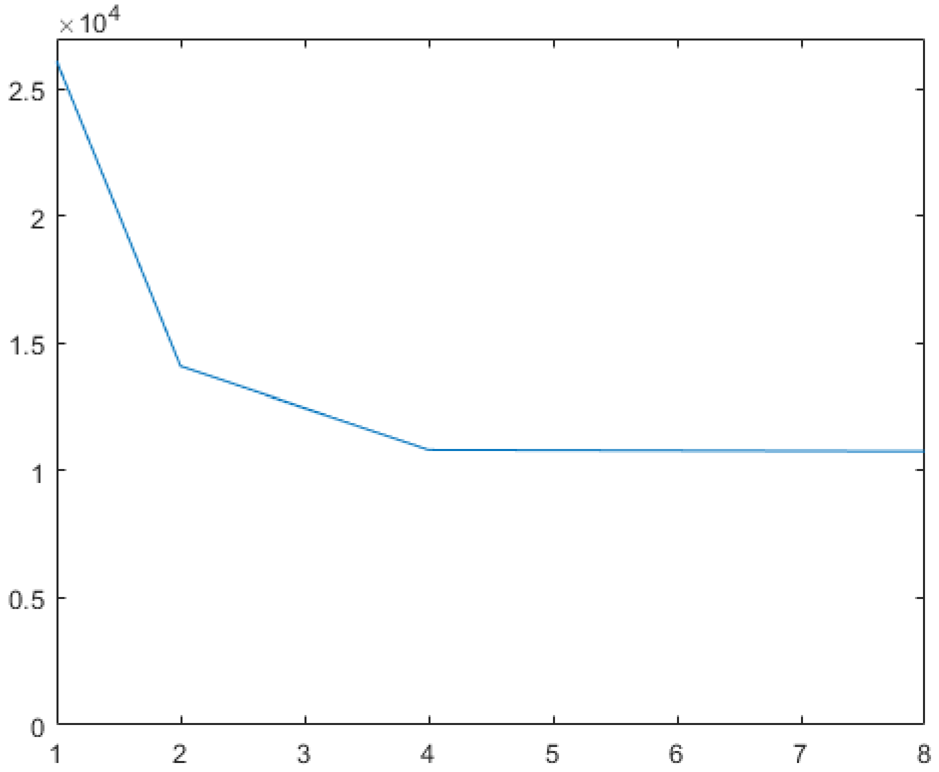
pre-parallelization optimizations

After parallelizing, here are my results for the number of threads vs time up to 8 threads:



parallel solution

Then I increased the number of trapezoids by a factor of 100 and reran the tests with up to 8 threads:



parallel solution - 100x as many calculations

12 Trapezoidal Rule - Conclusions

I managed to achieve significant improvement up to 4 threads, but from 4 to 8 threads there is no significant change. Part of this is likely due to the fact that my linux machine only actually has 4 cores, but it's possible that there is significant downtime while different threads are waiting to write to the global variable. Either way, there seem to be diminishing returns when adding threads. With 4 threads, the speedup is 2.31 and the efficiency is 0.578.

Notice how when I multiplied the size of the problem by 100, the shape of the graph stayed approximately the same. When I scaled up the problem, the times increased, but the shape of the graphs stayed about the same, indicating that my multithreaded runtime grows at the same rate as the sequential solution. With 4 threads, the speedup is 2.41 and the efficiency is 0.603.