Steven Comer

CS 1645

HW 5

18 Mar 2014

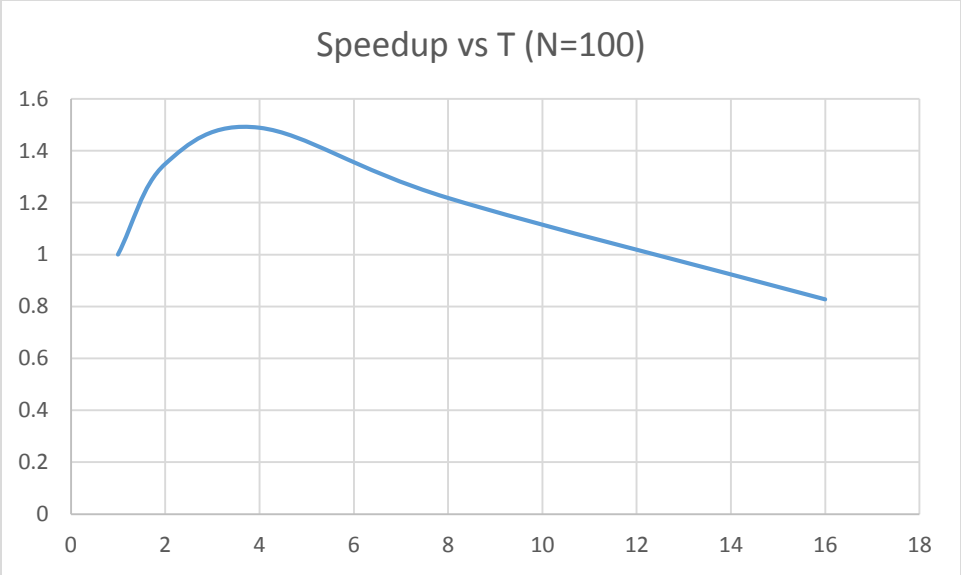
GAUSSIAN

1. This is a general strategy of the parallelization effort. Why did you choose those directives to parallelize the program?

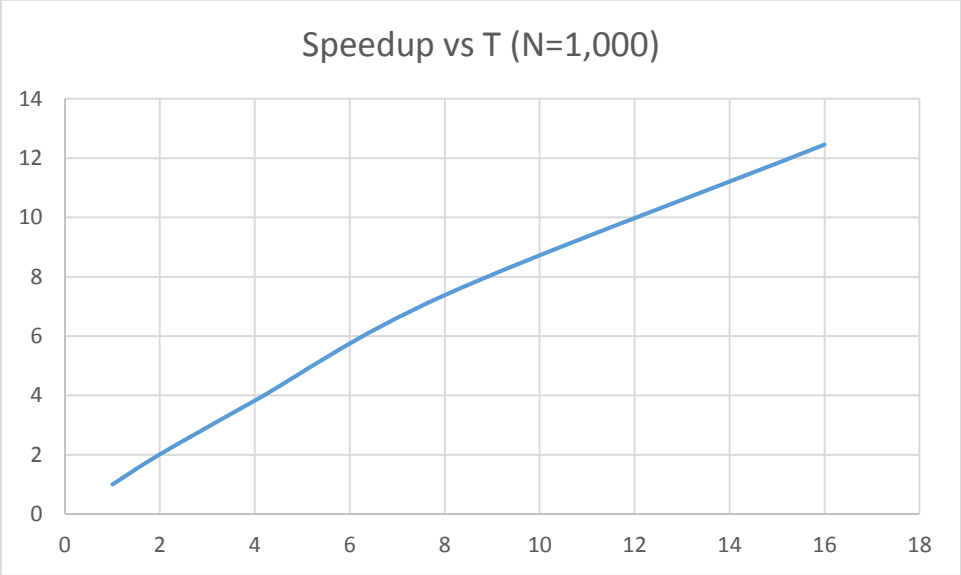
> This program contains four for loops. For this reason, I chose to use the directive "#pragma omp parallel for …" in order to parallelize the code. There were four different possible place to insert this directive. I tested multiple combinations and placements of the directives, while keeping in mind the data dependencies inherent in the program.

2. This is a speedup analysis. Using 3 different interesting values of N and these values for T {1, 2, 4, 8, 16}, make a plot for speedup. Describe the curve.

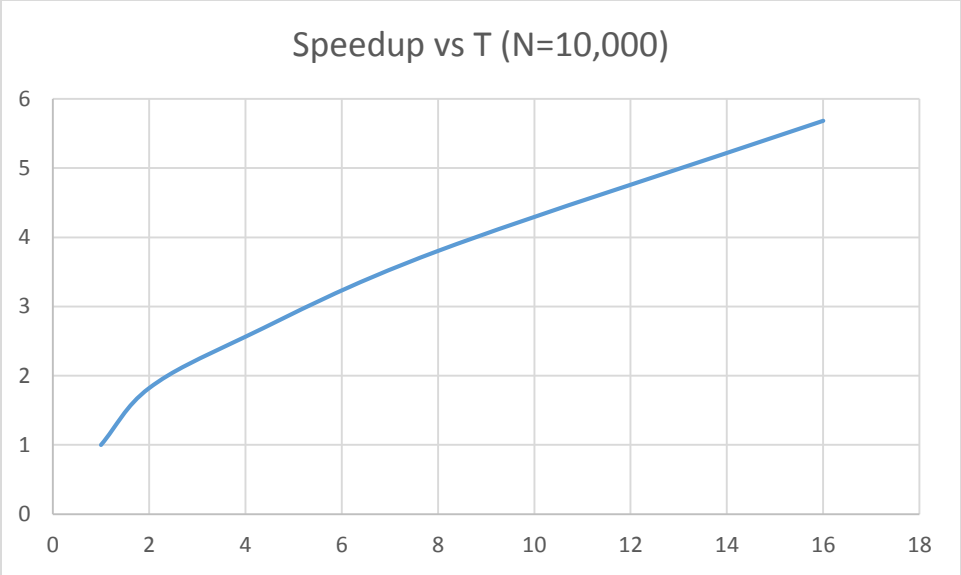
> I used three different values of N for the testing as well as {1, 2, 4, 8, 16} for the number of threads of execution. For N I used {100; 1,000; 10,000}. The curve of the speedup is best when N=1,000. See each plot for comments about the curve.



When N=100, the speedup increases at first, then decreases due to large overhead.



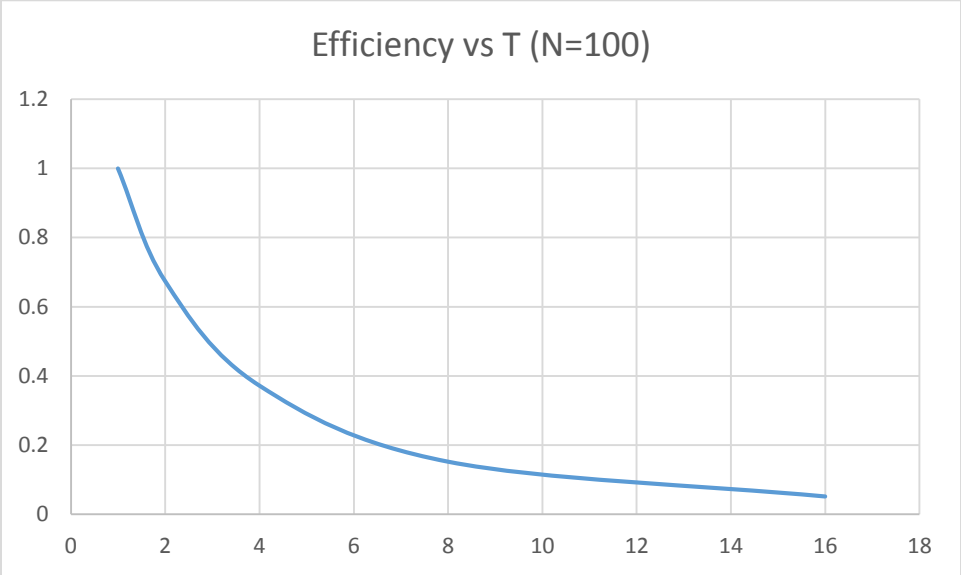
When N=1,000, speedup is approximately linear. It tapers off into sublinear space as N approaches 16.



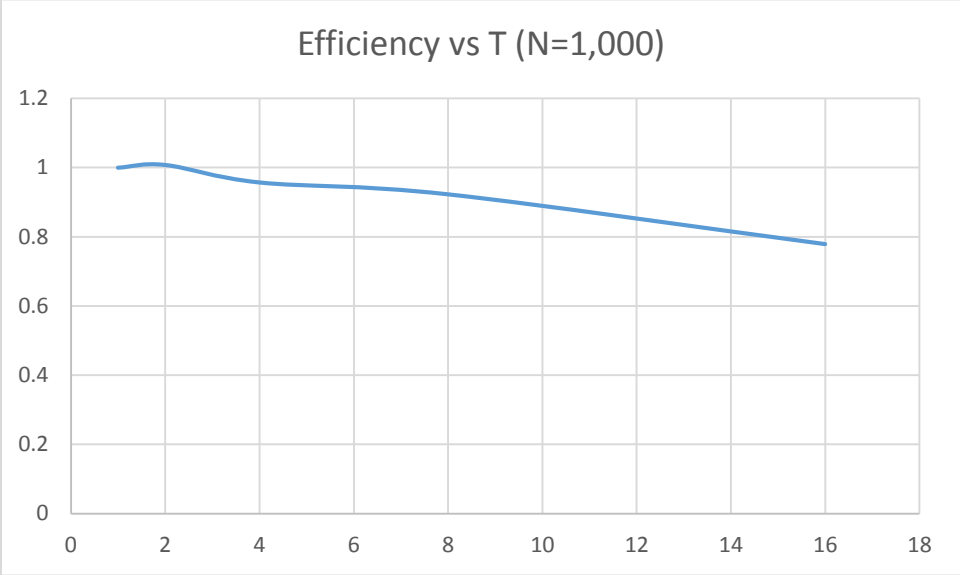
When N=10,000, the speedup starts off in a linear trend but becomes sublinear more quickly than when N=1,000. This is still better than N=100 though.

3. This is an efficiency analysis. Using 3 different interesting values of N and these values for T {1, 2, 4, 8, 16}, make a plot for efficiency. Describe the curve.

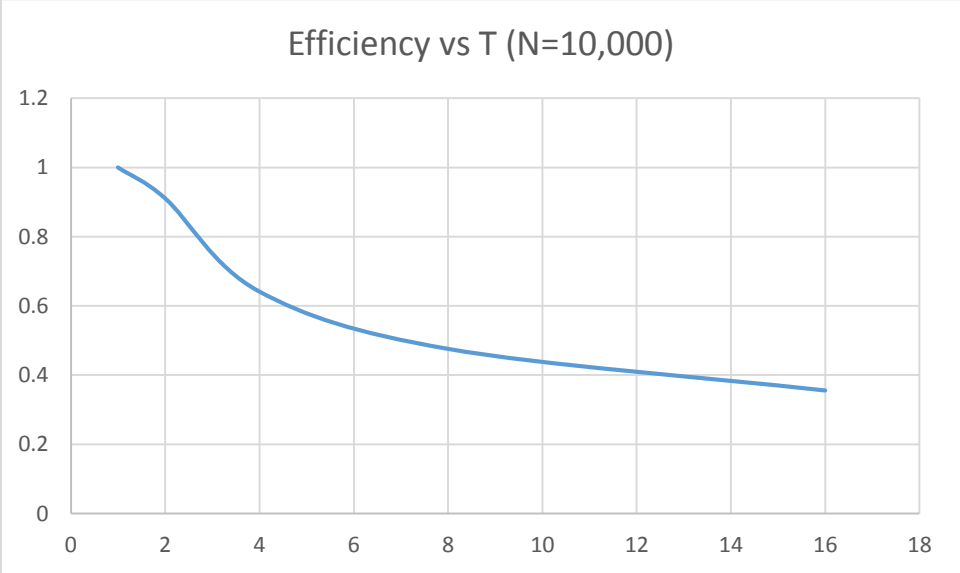
I used three different values of N for the testing as well as {1, 2, 4, 8, 16} for the number of threads of execution. For N I used {100; 1,000; 10,000}. The curve of the efficiency is best when N=1,000. See each plot for descriptions of the curve.



For N=100, the curve is nowhere near constant. It quickly drops from 1, reflecting the large amount of overhead for fork/join calls and a relatively small problem size.



For N=1,000, the curve is close to 1 with a relatively small decreasing slope. It is much better than the N=100 and N=10,000 in comparison.



For N=10,000, the plot of efficiency resembles that of N=100. It quickly drops away from 1 and the slope begins to slow around an efficiency of 0.4.

4. This is a description of the performance bottlenecks. What is preventing the program from getting linear speedup?

There are four for loops in the program. The outermost loop (looping on k) has a data dependency with previous values of i. For this reason this loop cannot be parallelized. The second loop is too short to make efficient use of the parallelization. It results in too many fork/join calls. The fourth loop again suffers from the problem of frequent fork/join calls. The third loop that appears sequentially in the program is the only one that provides a good benefit in speedup and also does not have any data dependencies. The fact that the other three loops cannot be parallelized prevents this program from getting linear speedup.

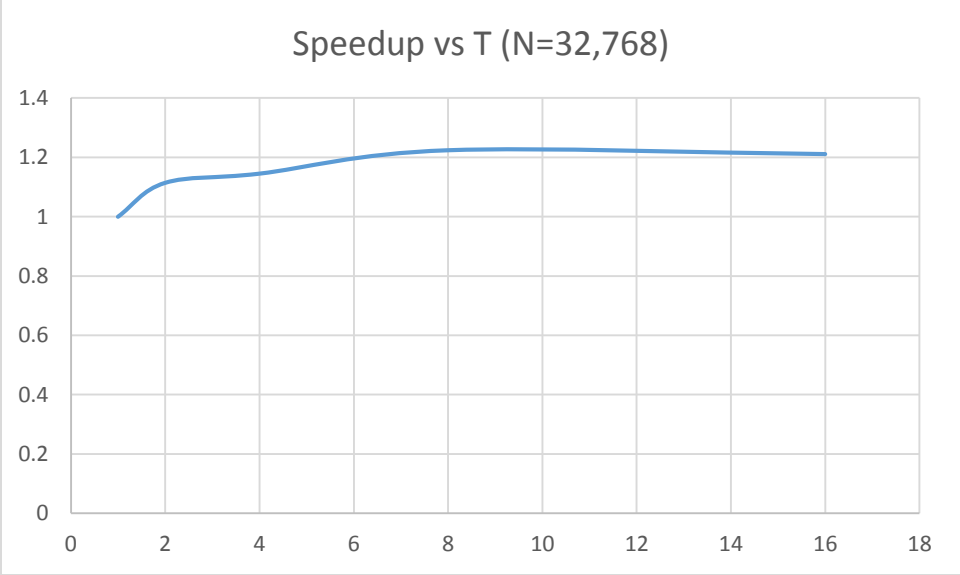
Fast-Fourier Transform (FFT)

1. This is a general strategy of the parallelization effort. Why did you choose those directives to parallelize the program?

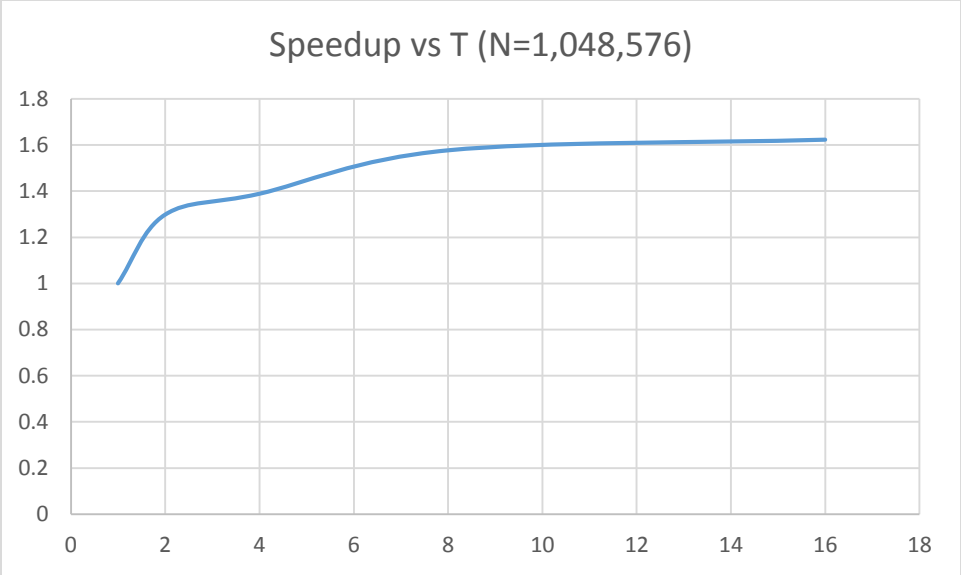
This program also contains four for loops. We converted the variables ‘wr’ and ‘wi’ into arrays (I called them ‘wwr’ and ‘wwi’, respectively) so that they can be used as fast lookups and some calculation can be removed from the nested for loops (loops 3 & 4 are the nested loops). Again I used the “#pragma omp parallel for …” directive to parallelize this program. This fits because it allows the compiler to effectively deal with the for loops that already exist in the code.

2. This is a speedup analysis. Using 3 different interesting values of N and these values for T {1, 2, 4, 8, 16}, make a plot for speedup. Describe the curve.

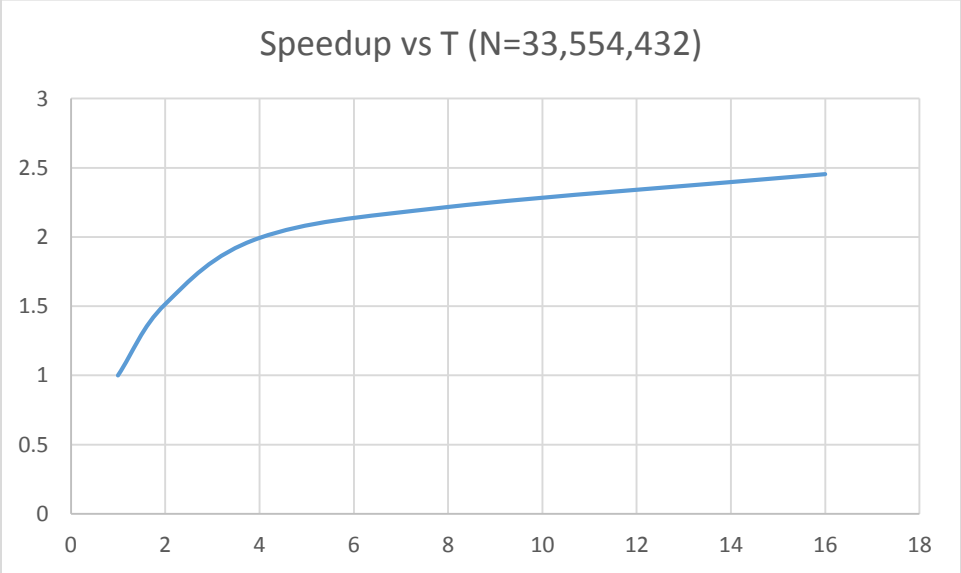
I used three different values of N for the testing as well as {1, 2, 4, 8, 16} for the number of threads of execution. For N, I used {32,768; 1,048,576; 33,554,432}.



For N=32,768, the speedup curve is relatively constant. It provides a slight benefit, but not nearly what it should. This is partially due to large overhead for a small N.



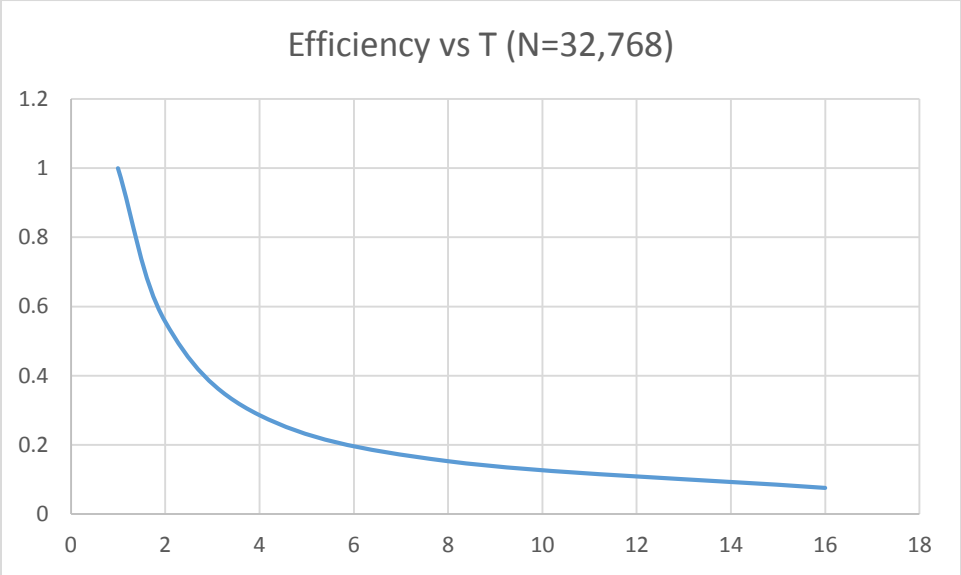
For this value of N, the curve starts off in a linear fashion but quickly becomes constant. This is not good for the speedup which should be an increasing line with an approximate slope of 1 if it is linear.



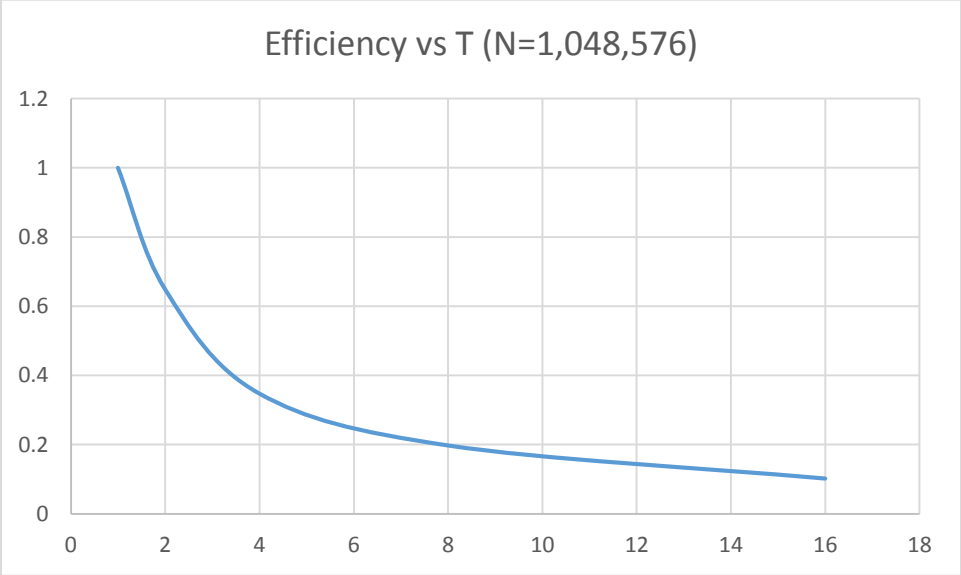
For this value of N, the curve is linear for a slightly longer period than the last plot. However this one as well still begins to grow very slowly as T approaches 16.

3. This is an efficiency analysis. Using 3 different interesting values of N and these values for T {1, 2, 4, 8, 16}, make a plot for efficiency. Describe the curve.

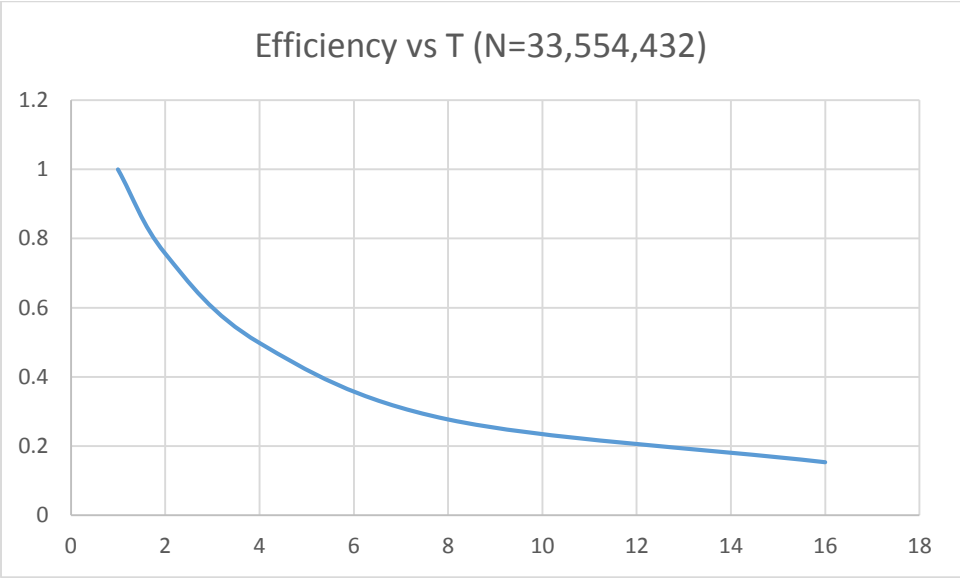
I used three different values of N for the testing as well as {1, 2, 4, 8, 16} for the number of threads of execution. For N I used {32,768; 1,048,576; 33,554,432}. The curve of the efficiency is best when N = 33,554,432. See each plot for descriptions of the curve.



For N = 32,768, the efficiency is nowhere near 1. It starts off at 1 but quickly decreases. It slows the rate of decay such that it appears to approach 0.1 as T grows larger.



For N=1,048,576, the plot is very similar to the one above. The only noticeable difference is that it appears to be settling on a value slightly higher than for N=32,768. Again, this plot is nowhere near the constant 1 that we would hope for.



For N=33,554,432, the plot again grows in a negative manner and is not linear. It is the best of the 3 selected values of N however it is still nowhere near constant 1. This parallelization effort is not very efficient.

4. This is a description of the performance bottlenecks. What is preventing the program from getting linear speedup?

There are four for-loops in this program as well as in Gaussian. This program starts with 3 for-loops in its original form, but we extracted an additional loop to calculate the ‘wr’ and ‘wi’ values ahead of time to speed up the nested loop slightly. The first two loops are both single level and did not bring much, if any, benefit to the performance of the program when they were parallelized. The first loop has a data dependency as well. The bottleneck of this program is that the bulk of the calculation occurs within the innermost for-loop (the fourth loop in the program). This creates a runtime on the order of N^2. We add some speedup by parallelizing the outer of the nested loops, but it is not enough to achieve linear speedup.