Tanner Pflager
CS475
PA1 Report
9/13/2016

Jacobi 1D

Algorithm

Complexity: $O(n^2)$

Every iteration of the inner for loop touches 4 memory locations. One of those pointers gets written into, while the other 3 are read from. The memory access is regular in the inner for loop. However, every entry into the inner for loop accesses different memory.

Parallelization Approach

Shared Memory, Static Scheduling
Parallelized the for loop within the while loop.

Experimental Setup

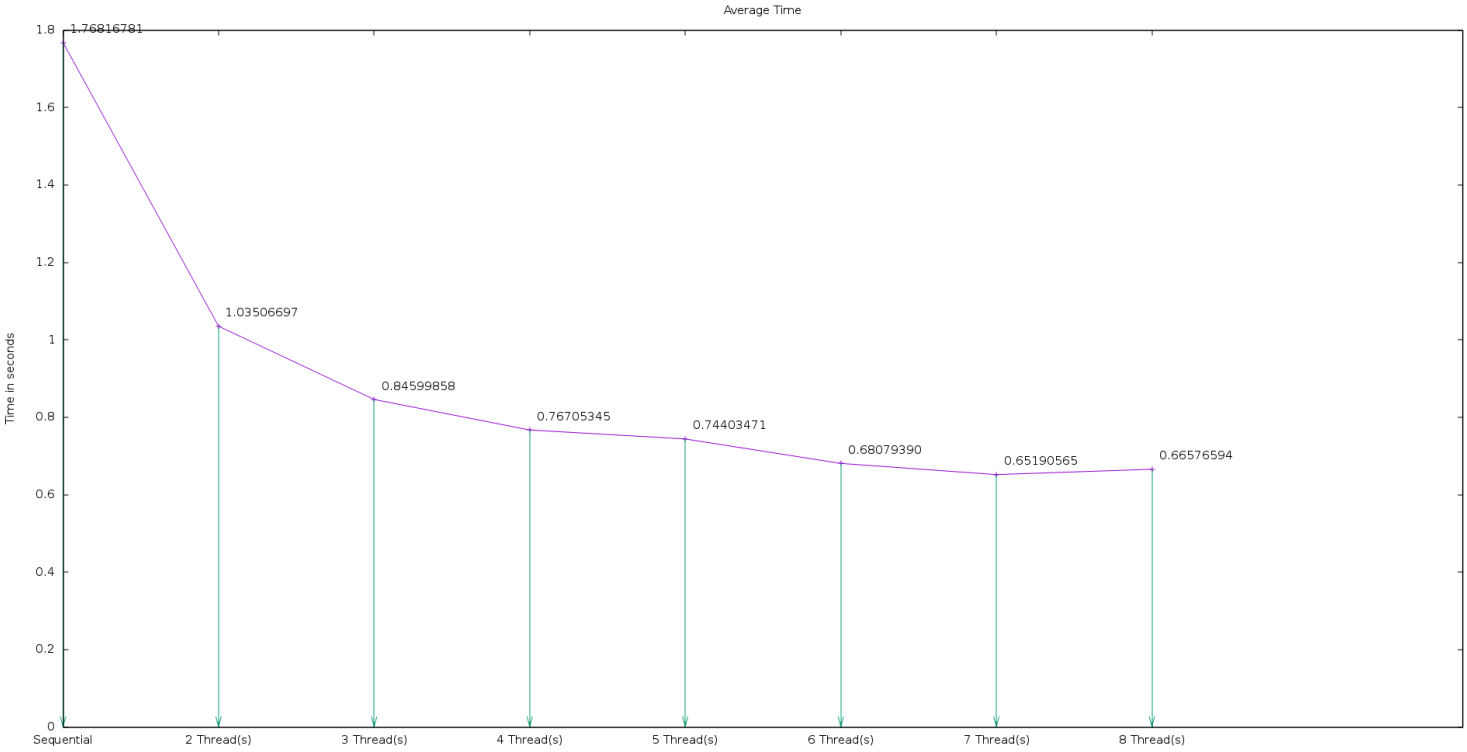
The testing was does on Lincoln. Lincoln has 8 hyper-threaded cores at 2.6 gigahertz, but can reach a maximum of 3.4 gigahertz. L1d cache and L1i cache are both 32K. The L2 cache is 256K and the L3 cache is 20480K. The CPUs architecture is x86_64.

There was a total of 30 tests done on the sequential program. Then tested the parallelized program with 1, 2, 3, 4, 5, 6, 7, and 8 threads, each 30 times.

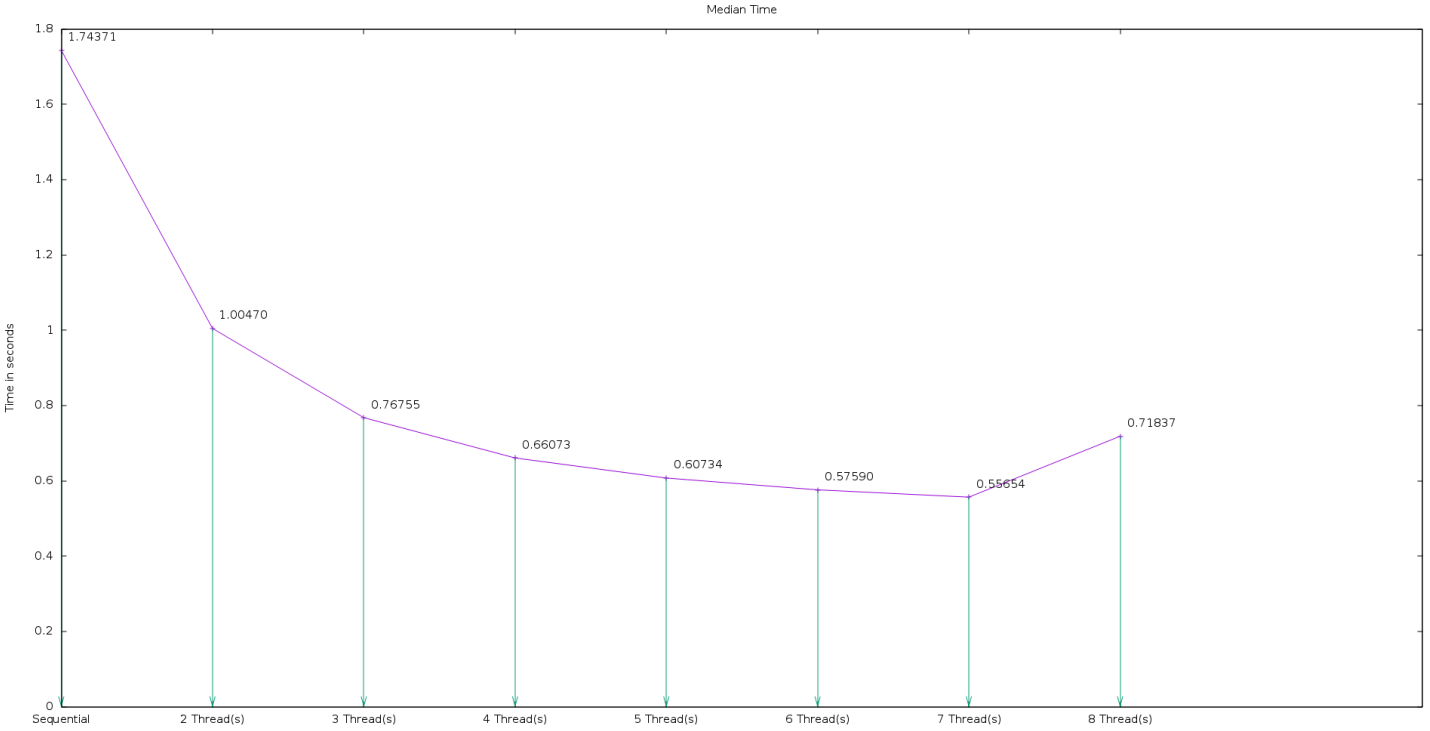
Results

When looking at the results, this program did benefit from parallelization. It did not obtain ideal speedup, but it did however achieve linear speedup. Speedup and efficiency started to taper off after 4 threads. I also, noticed that in both the median and average execution times, 8 threads was slower than 7 threads. Possibly, the problem size is not large enough to reap benefit from more than 7 threads.

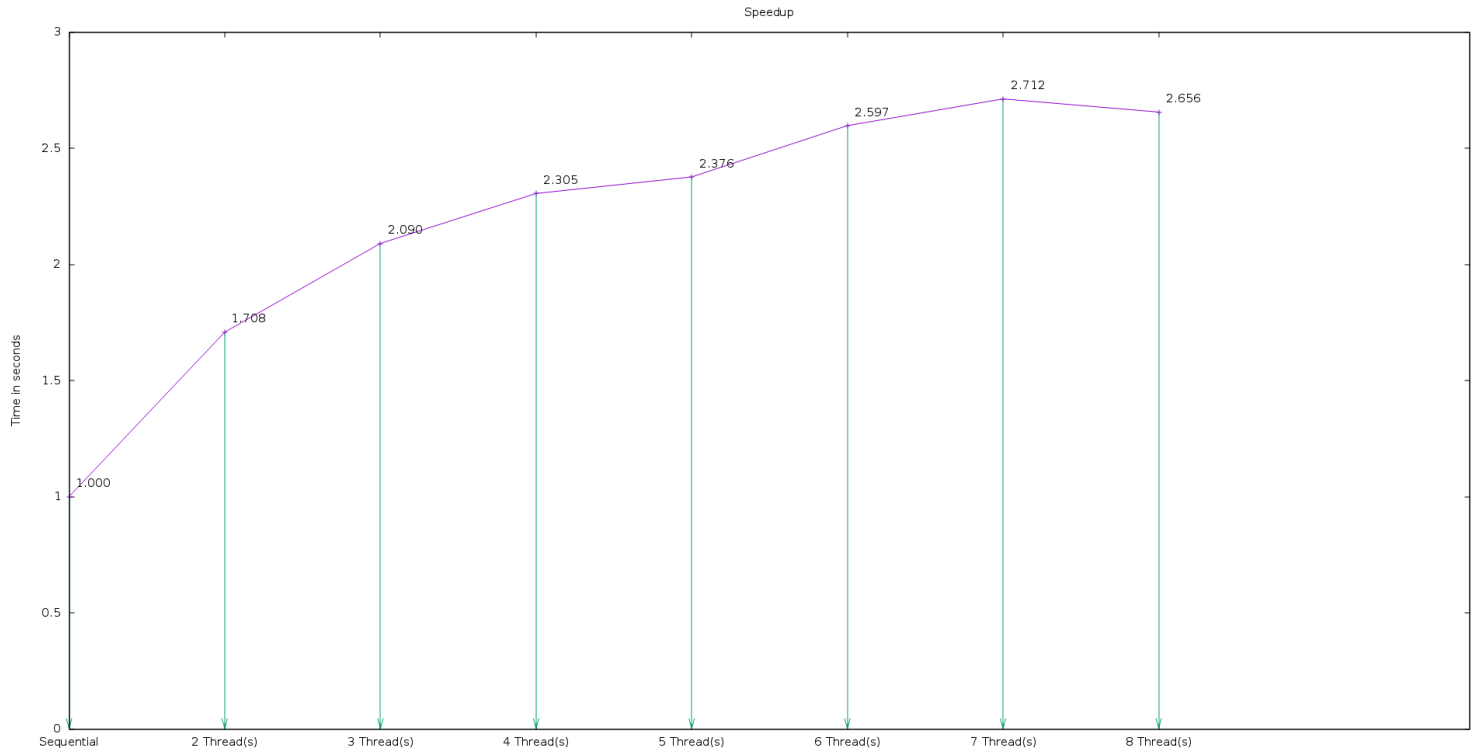
Average Execution Time (in seconds)



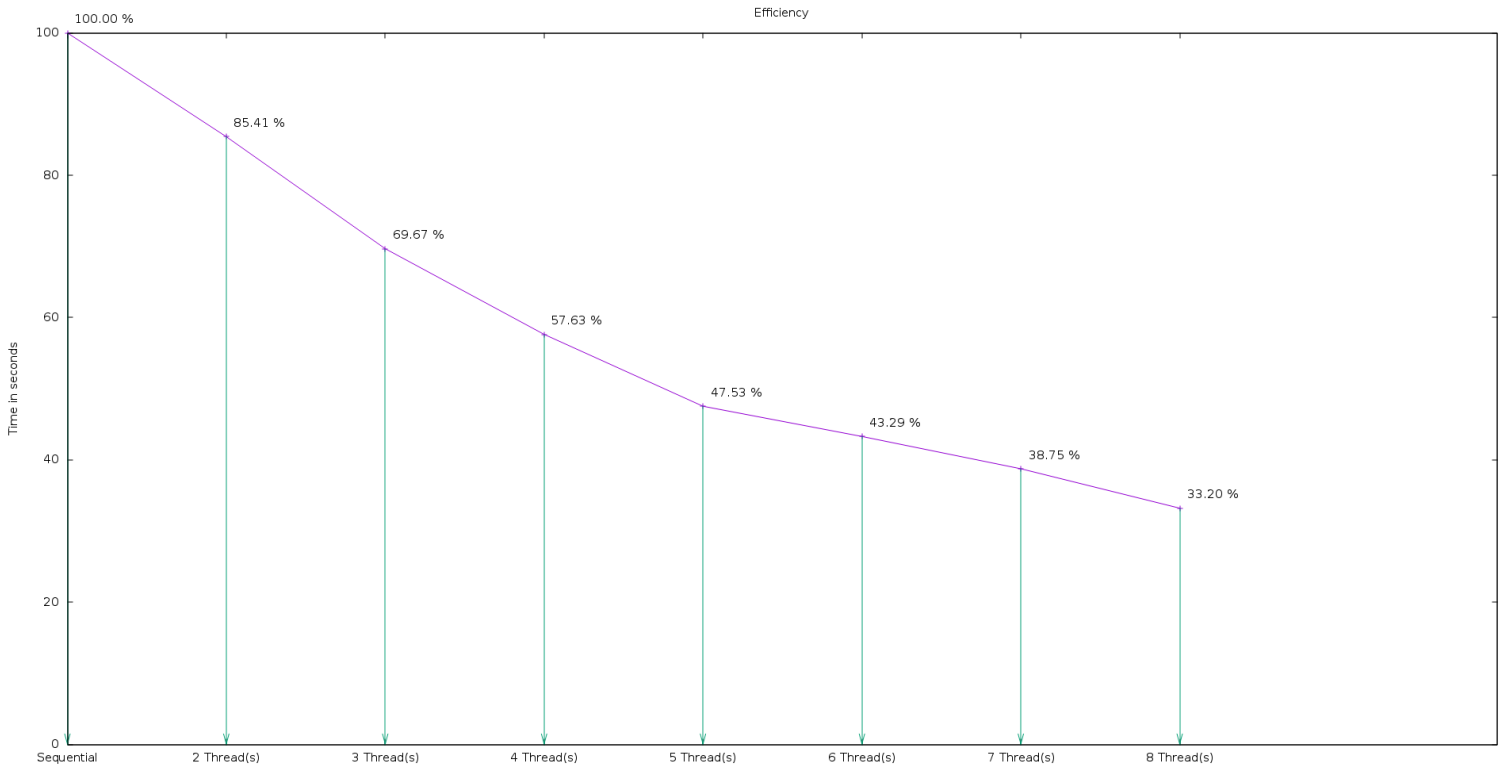
Median Execution Time (in seconds)



Speedup



Efficiency



In conclusion, linear speedup was achieved. However, diminishing returns started to take place as early as 4 threads. This is likely due to fork / join overhead but could also be blamed on the problem size was not large enough to benefit from utilizing more cores.

Jacobi 2D

Algorithm

Complexity: $O(n^3)$

Every iteration of the most inner for loop touches 9 memory locations. One of those pointers gets written into, while the other 8 are read from. The memory access is irregular because there is a large cap in between used memory addresses.

Parallelization Approach

Shared Memory, Static Scheduling
Parallelized the outer most for loop within the while loop.

Experimental Setup

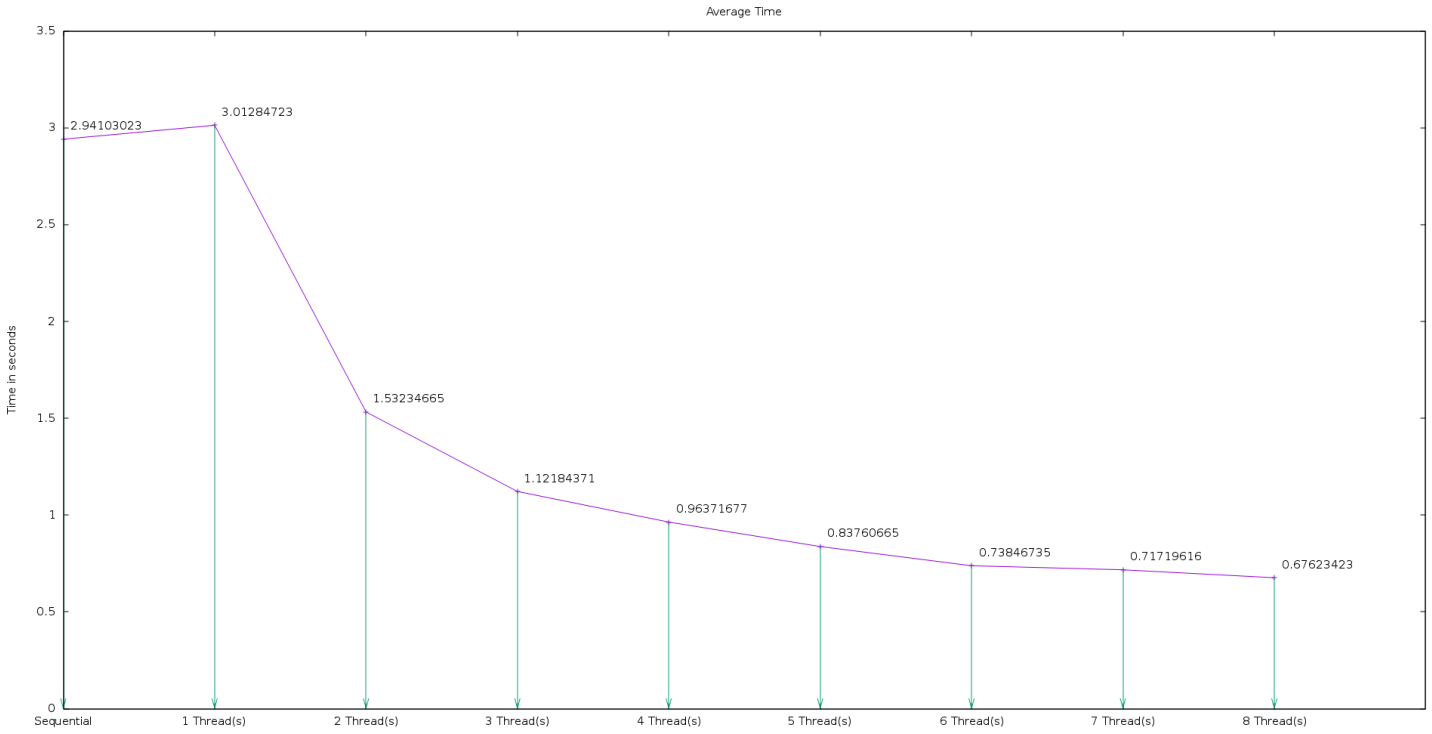
The testing was does on Lincoln. Lincoln has 8 hyper-threaded cores at 2.6 gigahertz, but can reach a maximum of 3.4 gigahertz. L1d cache and L1i cache are both 32K. The L2 cache is 256K and the L3 cache is 20480K. The CPUs architecture is x86_64.

There was a total of 30 tests done on the sequential program. Then tested the parallelized program with 1, 2, 3, 4, 5, 6, 7, and 8 threads, each 30 times.

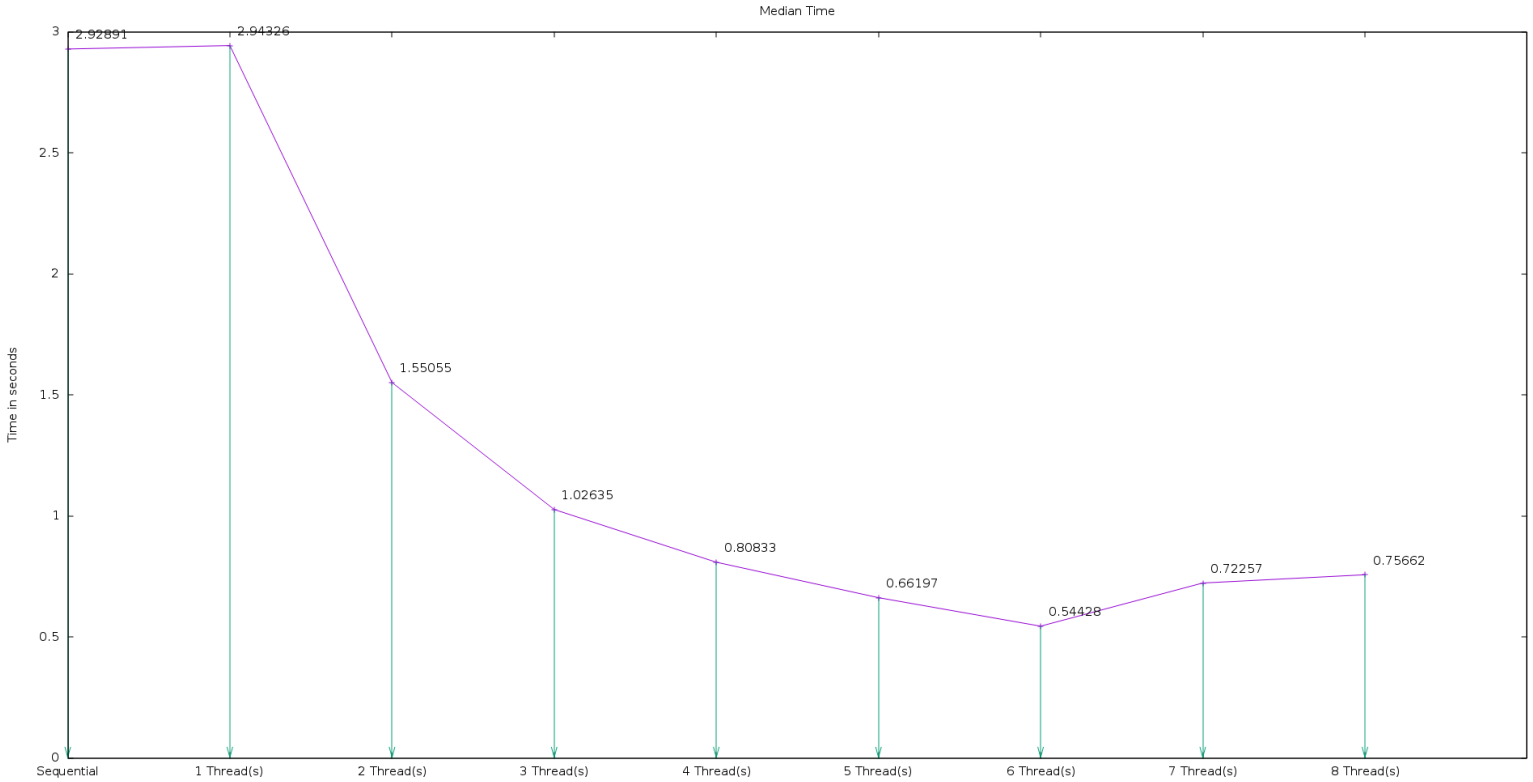
Results

When looking at the results this program benefited substantially from parallelization excluding the use of a single thread in the parallelized version. Execution time in the sequential version was 2.94 seconds and with 8 threads in the parallelized version was 0.67 seconds. With the use of 8 threads the efficiency was around 50%. Ideal speedup was almost achieved when just 2 threads were used, but overall linear speedup was achieved.

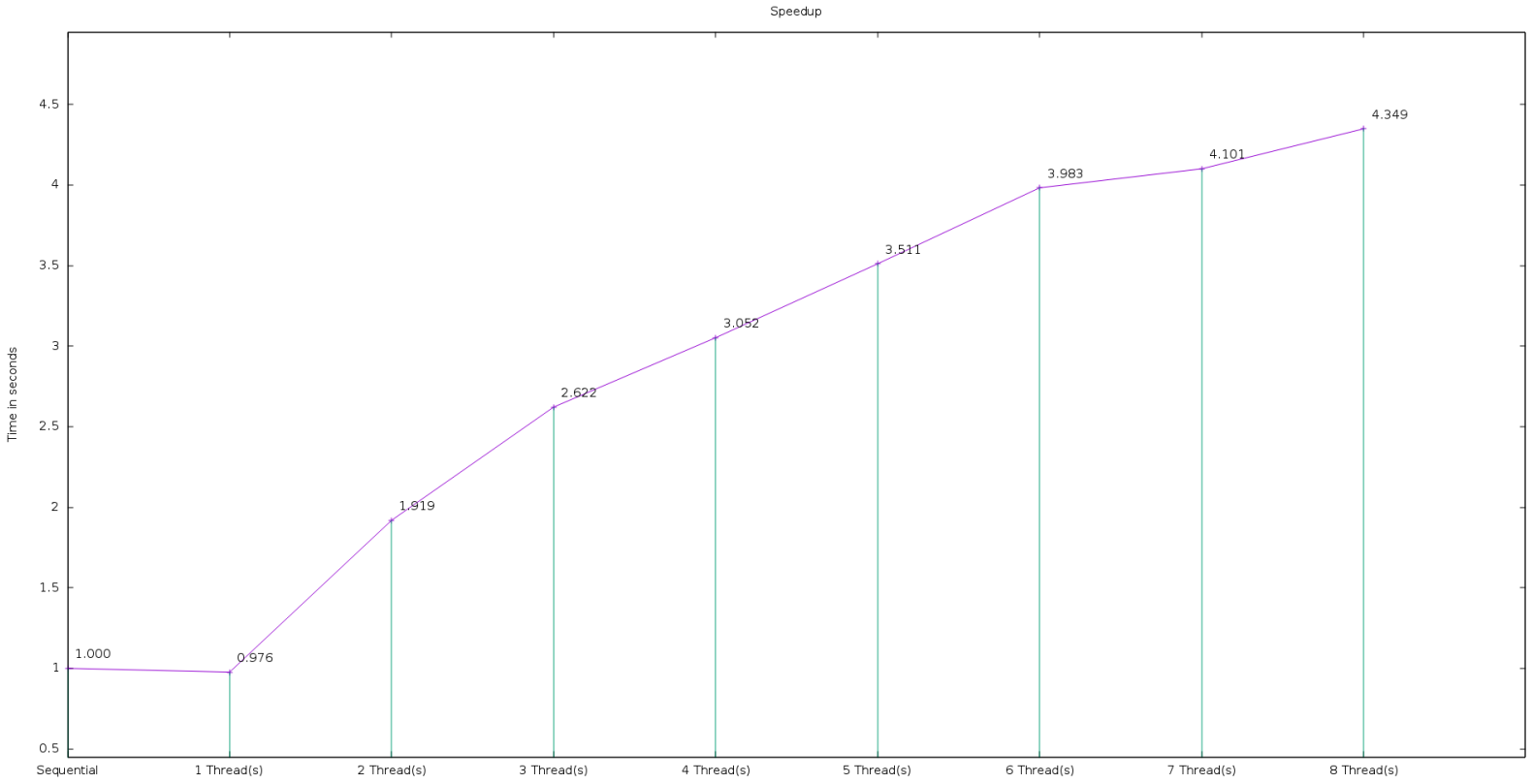
Average Execution Time (in seconds)



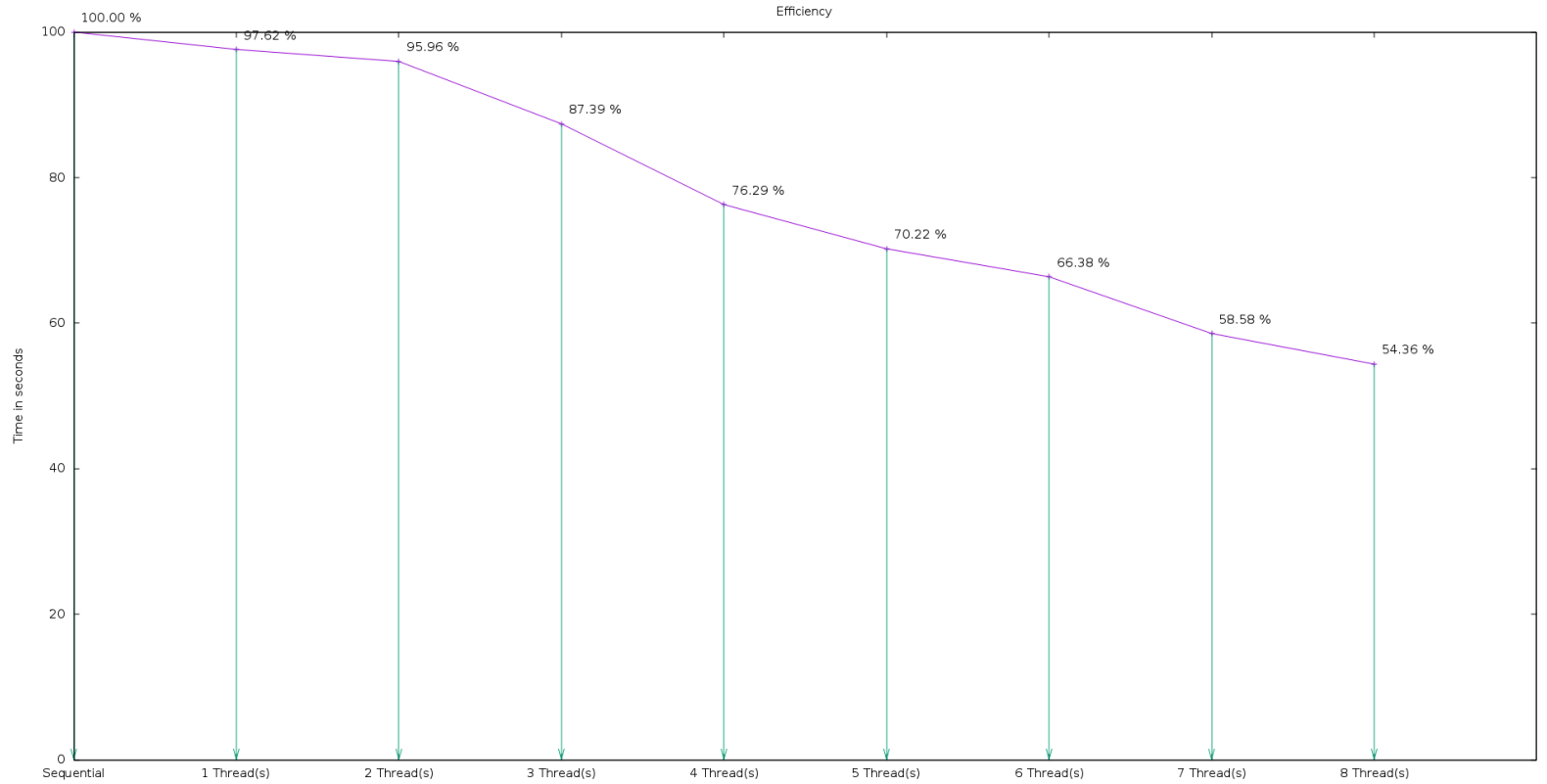
Median Time (in seconds)



Speedup



Efficiency



In conclusion, linear speedup was achieved. However, almost achieving ideal speedup with 2 threads and then tapering off shortly after 3 threads may mean the parallelization strategy is flawed. Since this algorithm is using irregular memory accesses, it is very likely that is susceptible to cache misses and possibly can be improved further. This program benefited more so than Jacobi 1D likely because the complexity is much greater.

Mat Vec

Algorithm

Complexity: O(n)

Originally the complexity was $O(n^2)$, but was optimized by getting rid of the nested for loop. Instead of a nested for loop, two separate for loops were used. Also, optimized memory use by using a single vector rather than 3. The other 2 vectors were just holding values that were replaced by a simple equation.

Parallelization Approach

Shared Memory, Static Scheduling

Experimental Setup

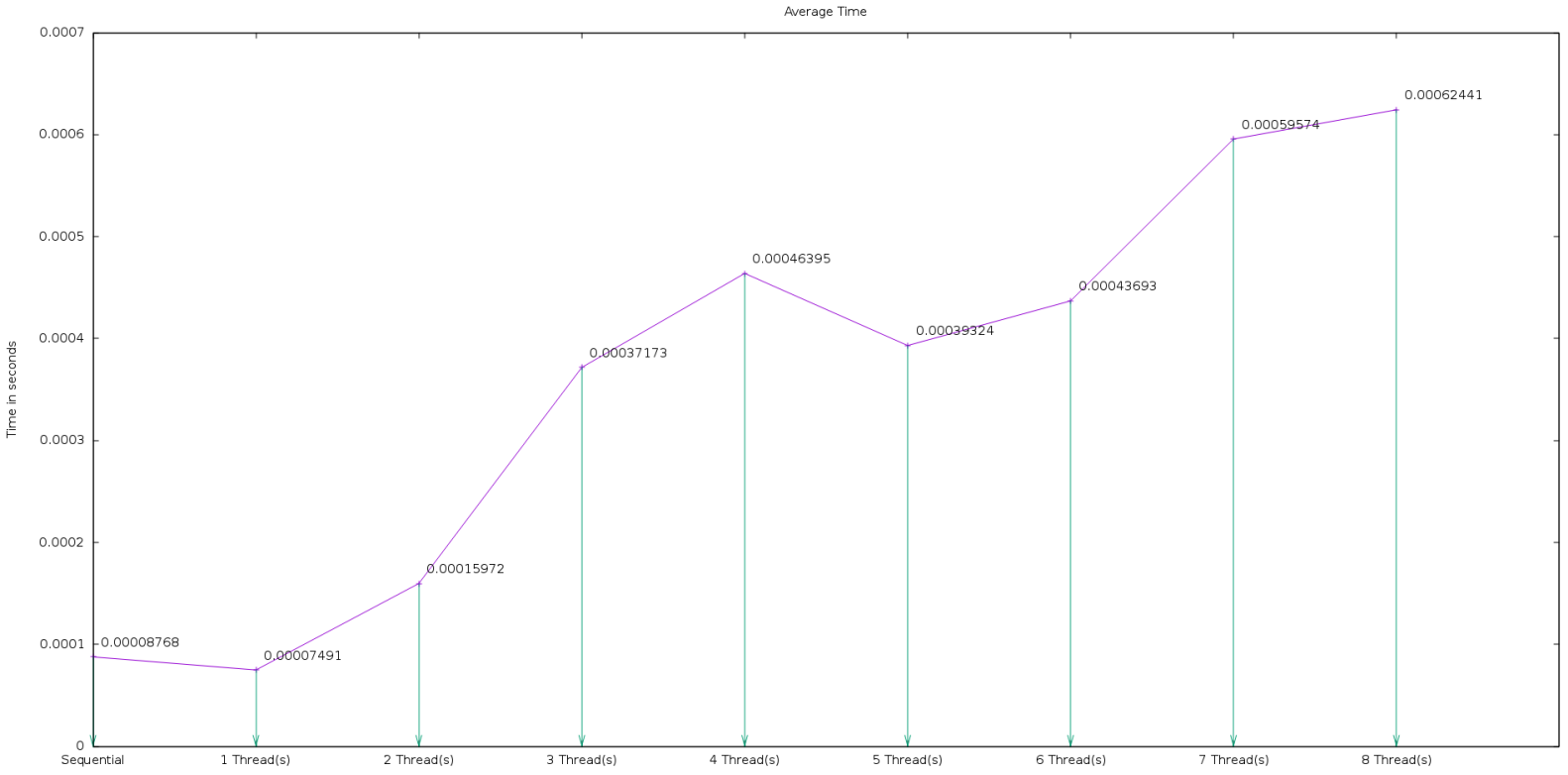
The testing was does on Lincoln. Lincoln has 8 hyper-threaded cores at 2.6 gigahertz, but can reach a maximum of 3.4 gigahertz. L1d cache and L1i cache are both 32K. The L2 cache is 256K and the L3 cache is 20480K. The CPUs architecture is x86_64.

There was a total of 100 tests done on the sequential program. Then tested the parallelized program with 1, 2, 3, 4, 5, 6, 7, and 8 threads, each 100 times. Attempted to use 30 tests, but wanted to ensure the results I saw were accurate, so 100 tests each were used.

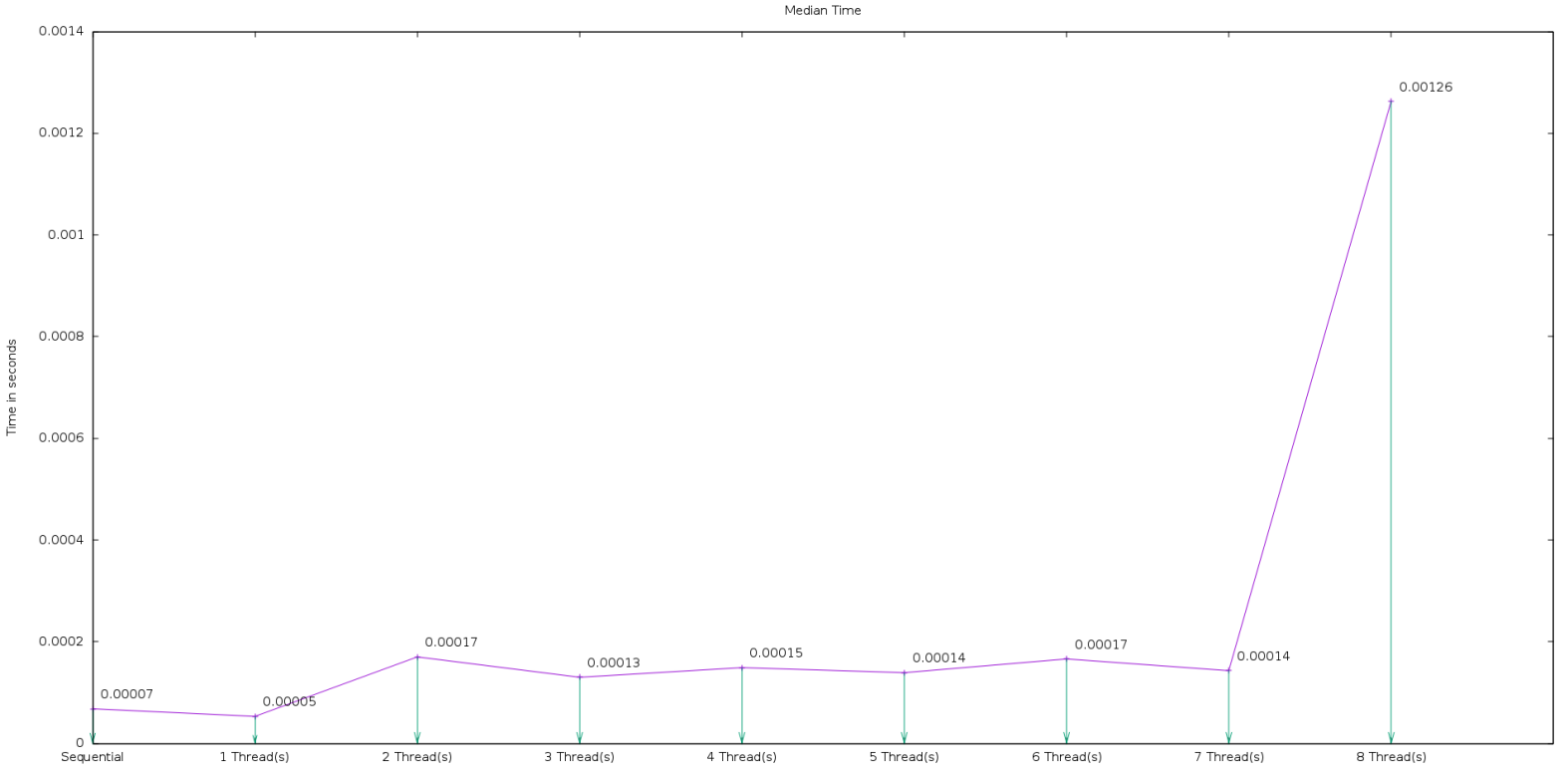
Results

The results showed that no speedup was achieved with parallelization. The sequential program had the fastest execution times, excluding the parallelization with 1 thread used. Since linear speedup was not achieved, efficiency also was poor and quickly approached zero.

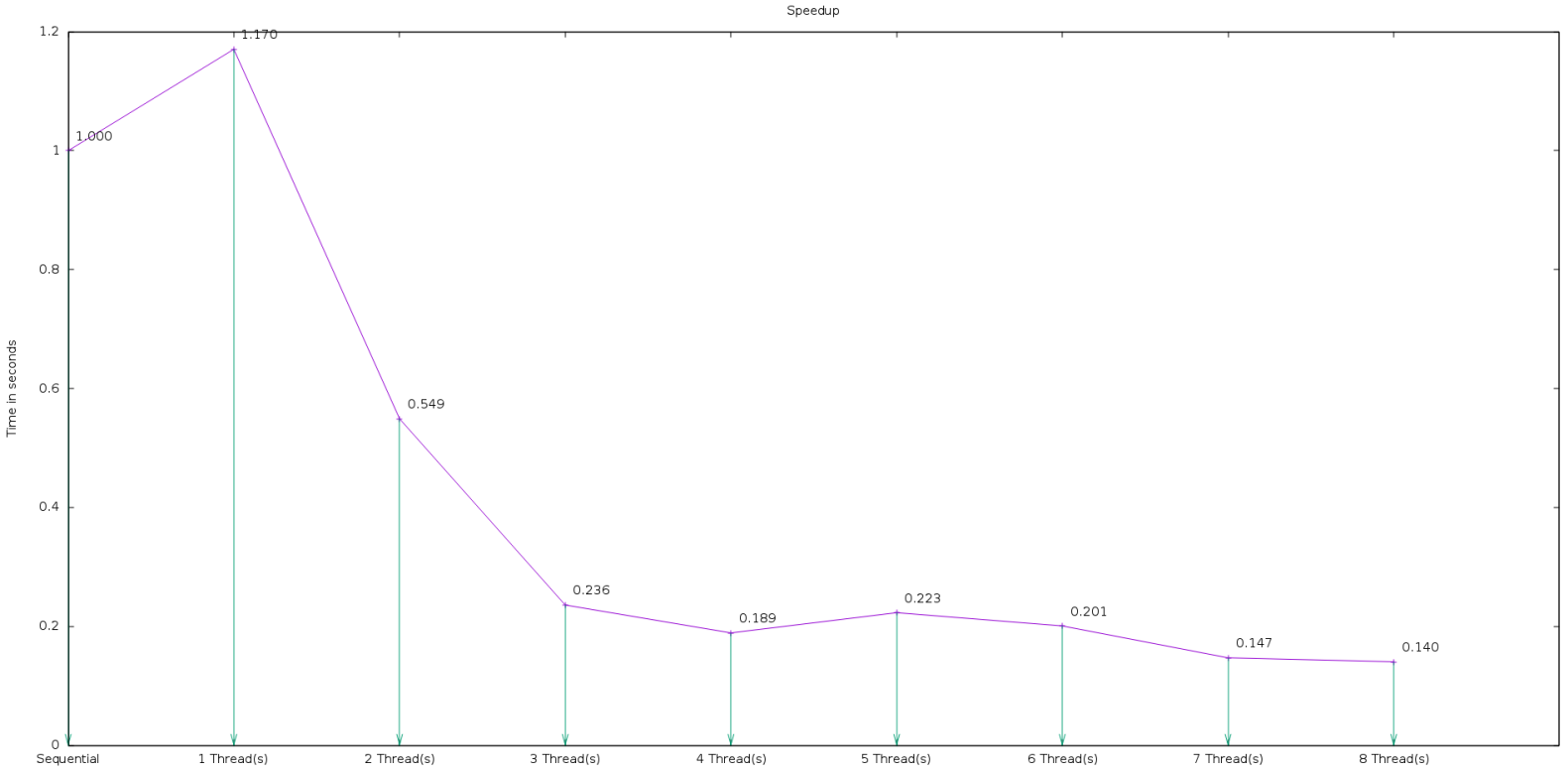
Average Execution Time (in seconds)



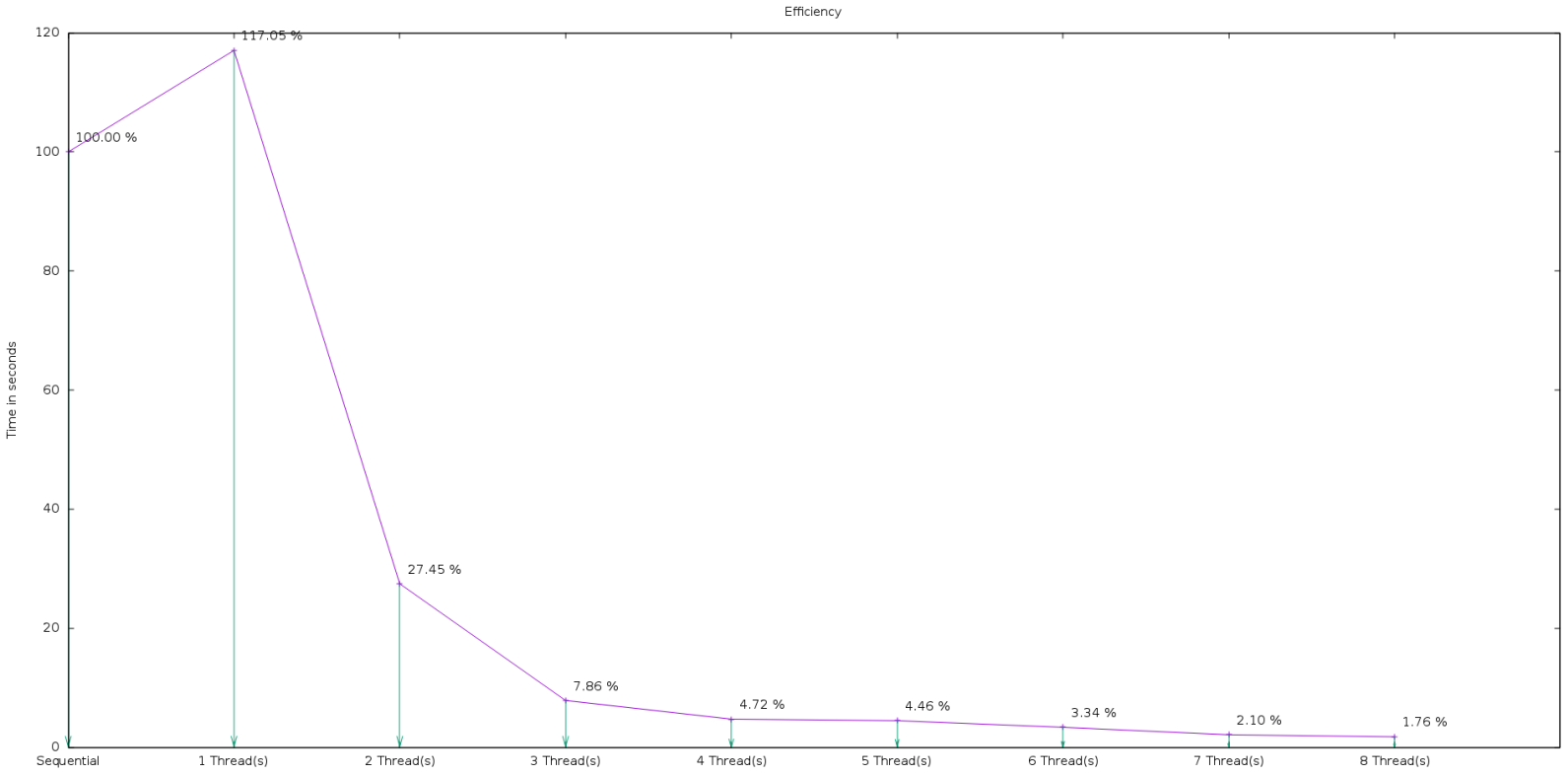
Median Execution Time (in seconds)



Speedup



Efficiency



In conclusion, the optimization of the sequential version of Mat Vec, did not leave enough work for the executable to benefit from parallelization. With a large problem size the results may have been different. The results showed that the parallel version was slower, but many of the values could be subject to a margin of error. All the average execution times were within one one thousandth of each other. A lesson learned from these results is that just because a problem is “easily parallelized” does not mean you will get better performance from parallelization.