Performance Analysis Report for Thread Based Pi Calculator

Lucas Sudol
Sebastian Kula

Table of Contents

Correctness Analysis

*Compiler optimizations were not used in compiling this assignment, as for repeat tests with the same iteration sizes, the compiler would optimize the algorithm for computing pi. This would include saving the computed value of pi for a specific iteration, making all other computations have a O(1) complexity, drastically changing the timing results of the program.

Testing Correctness of Multithreaded Application

Test Case 1: Checking serial calculations with single threaded application
Iteration sizes: 10,100,1000,10000, 100000, 1000000, 10000000, 100000000

Test runs a serial program, then single threaded multi-threaded application with each iteration size, and compares Pi from both programs to 20 decimal points. Test is to verify the correctness of the iteration size of pi calculations between programs.

Test Passed

Test Case 2: Checking serial calculations with size 8 thread pool
Iteration sizes: 10,100,1000,10000, 100000, 1000000, 10000000, 100000000
Task size: 8
Thread pool: 8

Test runs the multi-threaded program with a thread pool of 8, and 8 tasks with the different iteration sizes. It runs a serial calculation of each iteration size, and compares it to the results of the multi-threaded application. This comparison tolerance is done to 20 decimal points. This test is used to verify correct calculations alongside a large thread pool.

Test Passed

Test Case 3: Checking serial calculations with size 4 thread pool, and varying task ordering
Iteration sizes: 10,100,1000,10000, 100000, 1000000, 10000000, 100000000
Task size: 16
Thread pool: 4

Test runs the multi-threaded program with a thread pool of 4, and 16 tasks with 8 different iteration sizes. It runs a serial calculation of each iteration size, and compares it to the results of the multi-threaded application. This comparison tolerance is done to 20 decimal points. This test is used to verify accuracy in the program when multiple threads have different iteration sizes, and may have to wait for other threads before program shutdown.

Test Passed
Test Report

```
 Test Case 1 with 1 thread(s):
  iterations:   10            Parallel Pi = 3.0418396189      Serial Pi = 3.0418396189       PASS
  iterations:   100           Parallel Pi = 3.1315929036      Serial Pi = 3.1315929036       PASS
  iterations:   1000          Parallel Pi = 3.1405926538      Serial Pi = 3.1405926538       PASS
  iterations:   10000         Parallel Pi = 3.1414926536      Serial Pi = 3.1414926536       PASS
  iterations:   100000        Parallel Pi = 3.1415826536      Serial Pi = 3.1415826536       PASS
  iterations:   1000000       Parallel Pi = 3.1415916536      Serial Pi = 3.1415916536       PASS
  iterations:   10000000      Parallel Pi = 3.1415925536      Serial Pi = 3.1415925536       PASS
  iterations:   100000000     Parallel Pi = 3.1415926436      Serial Pi = 3.1415926436       PASS
Test Case 2 with 8 thread(s):
  iterations:   10            Parallel Pi = 3.0418396189      Serial Pi = 3.0418396189       PASS
  iterations:   100           Parallel Pi = 3.1315929036      Serial Pi = 3.1315929036       PASS
  iterations:   1000          Parallel Pi = 3.1405926538      Serial Pi = 3.1405926538       PASS
  iterations:   10000         Parallel Pi = 3.1414926536      Serial Pi = 3.1414926536       PASS
  iterations:   100000        Parallel Pi = 3.1415826536      Serial Pi = 3.1415826536       PASS
  iterations:   1000000       Parallel Pi = 3.1415916536      Serial Pi = 3.1415916536       PASS
  iterations:   10000000      Parallel Pi = 3.1415925536      Serial Pi = 3.1415925536       PASS
  iterations:   100000000     Parallel Pi = 3.1415926436      Serial Pi = 3.1415926436       PASS
Test Case 3 with 4 thread(s):
  iterations:   10            Parallel Pi = 3.0418396189      Serial Pi = 3.0418396189       PASS
  iterations:   100           Parallel Pi = 3.1315929036      Serial Pi = 3.1315929036       PASS
  iterations:   1000          Parallel Pi = 3.1405926538      Serial Pi = 3.1405926538       PASS
  iterations:   10000         Parallel Pi = 3.1414926536      Serial Pi = 3.1414926536       PASS
  iterations:   100000        Parallel Pi = 3.1415826536      Serial Pi = 3.1415826536       PASS
  iterations:   1000000       Parallel Pi = 3.1415916536      Serial Pi = 3.1415916536       PASS
  iterations:   10000000      Parallel Pi = 3.1415925536      Serial Pi = 3.1415925536       PASS
  iterations:   100000000     Parallel Pi = 3.1415926436      Serial Pi = 3.1415926436       PASS
  iterations:   10            Parallel Pi = 3.0418396189      Serial Pi = 3.0418396189       PASS
  iterations:   100           Parallel Pi = 3.1315929036      Serial Pi = 3.1315929036       PASS
  iterations:   1000          Parallel Pi = 3.1405926538      Serial Pi = 3.1405926538       PASS
  iterations:   10000         Parallel Pi = 3.1414926536      Serial Pi = 3.1414926536       PASS
  iterations:   100000        Parallel Pi = 3.1415826536      Serial Pi = 3.1415826536       PASS
  iterations:   1000000       Parallel Pi = 3.1415916536      Serial Pi = 3.1415916536       PASS
  iterations:   10000000      Parallel Pi = 3.1415925536      Serial Pi = 3.1415925536       PASS
  iterations:   100000000     Parallel Pi = 3.1415926436      Serial Pi = 3.1415926436       PASS
```

Memory Leaks and Race Condition Test
Test Case: Run valgrind with greater task size then thread pool, and varying iteration sizes
Iteration sizes: 10,100,1000,10000, 100000, 1000000, 10000000, 100000000
Task size: 16
Thread pool: 4

```
valgrind --fair-sched=yes ./A1 valgrindTestCase.txt true
valgrind --tool=drd --fair-sched=yes ./A1 valgrindTestCase.txt true
```

Test runs a multithreaded application with thread pool size of 4, and 16 tasks of 8 different iteration sizes. This test is used to determine if there are any memory leaks or race conditions within the program. Program is executed twice, one time for each type of valgrind test. Varying iteration sizes were used to create differences in the timings between threads, to ensure some threads complete tasks before others. This is done in order to test closing of the program, when it hangs on the completion of remaining threads. Checking if all threads completed before memory cleanup and program exit.

Test Passed
Test Report

```
MEMORY REPORT
socs@cis3090:~/workdir/ThreadBasedScheduler$ valgrind --fair-sched=yes ./A1 valgrindTestCase.txt true
==6228== Memcheck, a memory error detector
==6228== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==6228== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==6228== Command: ./A1 valgrindTestCase.txt true
==6228==
Thread 2 completed computed Pi using 10 iterations, the result is 3.04183961892940324390
Thread 2 completed computed Pi using 1000 iterations, the result is 3.14059265383979413500
Thread 3 completed computed Pi using 100 iterations, the result is 3.13159290355855368659
Thread 4 completed computed Pi using 10000 iterations, the result is 3.14149265359003448950
Thread 2 completed computed Pi using 100000 iterations, the result is 3.14158265358971977577
Thread 2 completed computed Pi using 10 iterations, the result is 3.04183961892940324390
Thread 2 completed computed Pi using 100 iterations, the result is 3.13159290355855368659
Thread 2 completed computed Pi using 1000 iterations, the result is 3.14059265383979413500
Thread 2 completed computed Pi using 10000 iterations, the result is 3.14149265359003448950
Thread 2 completed computed Pi using 100000 iterations, the result is 3.14158265358971977577
Thread 3 completed computed Pi using 1000000 iterations, the result is 3.14159165358977432447
Thread 2 completed computed Pi using 1000000 iterations, the result is 3.14159165358977432447
Thread 4 completed computed Pi using 10000000 iterations, the result is 3.14159255358979150330
Thread 3 completed computed Pi using 10000000 iterations, the result is 3.14159255358979150330
Thread 1 completed computed Pi using 100000000 iterations, the result is 3.14159264358932599492
Thread 2 completed computed Pi using 100000000 iterations, the result is 3.14159264358932599492
==6228==
==6228== HEAP SUMMARY:
==6228==     in use at exit: 0 bytes in 0 blocks
==6228==   total heap usage: 33 allocs, 33 frees, 11,366 bytes allocated
==6228==
==6228== All heap blocks were freed -- no leaks are possible
==6228==
==6228== For lists of detected and suppressed errors, rerun with: -s
==6228== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

DRD REPORT

```
socs@cis3090:~/workdir/ThreadBasedScheduler$ valgrind --tool=drd --fair-sched=yes ./A1 valgrindTestCase.txt true
==6551== drd, a thread error detector
==6551== Copyright (C) 2006-2020, and GNU GPL'd, by Bart Van Assche.
==6551== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==6551== Command: ./A1 valgrindTestCase.txt true
==6551==
Thread 2 completed computed Pi using 10 iterations, the result is 3.04183961892940324390
Thread 2 completed computed Pi using 1000 iterations, the result is 3.14059265383979413500
Thread 3 completed computed Pi using 100 iterations, the result is 3.13159290355855368659
Thread 4 completed computed Pi using 10000 iterations, the result is 3.14149265359003448950
Thread 2 completed computed Pi using 100000 iterations, the result is 3.14158265358971977577
Thread 2 completed computed Pi using 10 iterations, the result is 3.04183961892940324390
Thread 2 completed computed Pi using 100 iterations, the result is 3.13159290355855368659
Thread 2 completed computed Pi using 1000 iterations, the result is 3.14059265383979413500
Thread 2 completed computed Pi using 10000 iterations, the result is 3.14149265359003448950
Thread 2 completed computed Pi using 100000 iterations, the result is 3.14158265358971977577
Thread 3 completed computed Pi using 1000000 iterations, the result is 3.14159165358977432447
Thread 2 completed computed Pi using 1000000 iterations, the result is 3.14159165358977432447
Thread 4 completed computed Pi using 10000000 iterations, the result is 3.14159255358979150330
Thread 3 completed computed Pi using 10000000 iterations, the result is 3.14159255358979150330
Thread 1 completed computed Pi using 100000000 iterations, the result is 3.14159264358932599492
Thread 2 completed computed Pi using 100000000 iterations, the result is 3.14159264358932599492
==6551==
==6551== For lists of detected and suppressed errors, rerun with: -s
==6551== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 272 from 37)
```

Analysis

With the results of the prior test cases, it can be concluded that the results of the multithreaded applications calculation of Pi are correct. Since the threads in the multithreaded program simply use the serial algorithm, errors can only result from fault in the message passing or any prevalent race conditions. Using different task sizes with thread numbers tested the messaging system, and each calculation was confirmed with the serai calculation for that same iteration size. Finally, the valgrind was used to check for memory leaks and race conditions, which none were reported for the given test cases.These tests all confirmed that the calculations of Pi for any given iteration size were correct, and that the threads were receiving the correct iteration sizes to calculate. Since the calculations are dependent on the same algorithm used in the serial version, no further testing was required to validate the parallelization of the tasks.

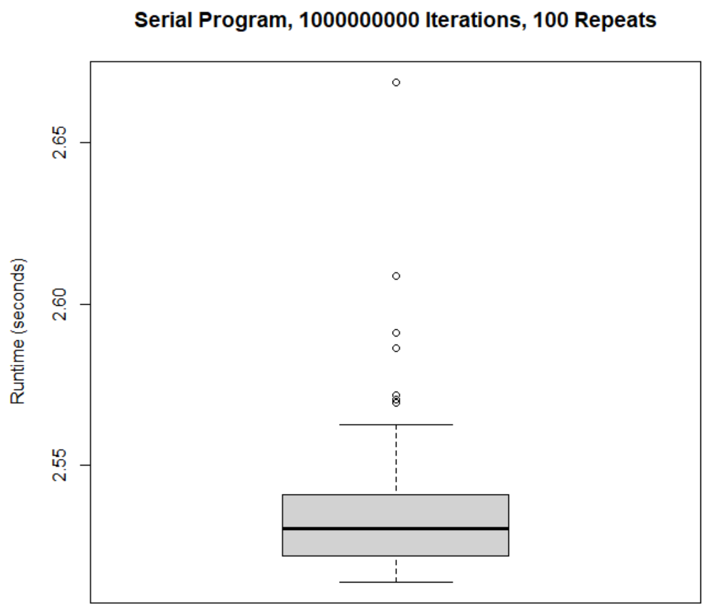
Performance Analysis

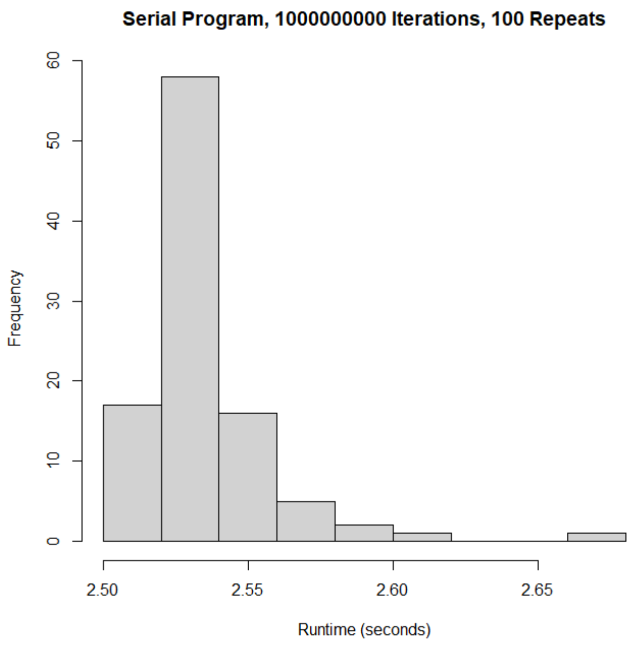
Computer Processor Information

Processor: AMD Ryzen™ 7 5800X
Core Count: 8
Thread Count: 16
SMT (Simultaneous Multithreading) - Yes

Each test was performed using 100 repetitions, and then the median of the sample was taken as result. This is due to the fact of the data being skewed, leading to the median being a better representation of the data, as mean results are affected by outliers.

Example: Variation in Serial Program Sample





It is important to consider skewness when timing programs, as it results from there only being factors that can "slow" the program down, not speed it up. This leads to timings being for the most part near their maximum, and have outliers when things start to slow down the cpu scheduler. These considerations are taken into account when using the median instead of the mean, as its result is not as skewed due to the presence of outliers.

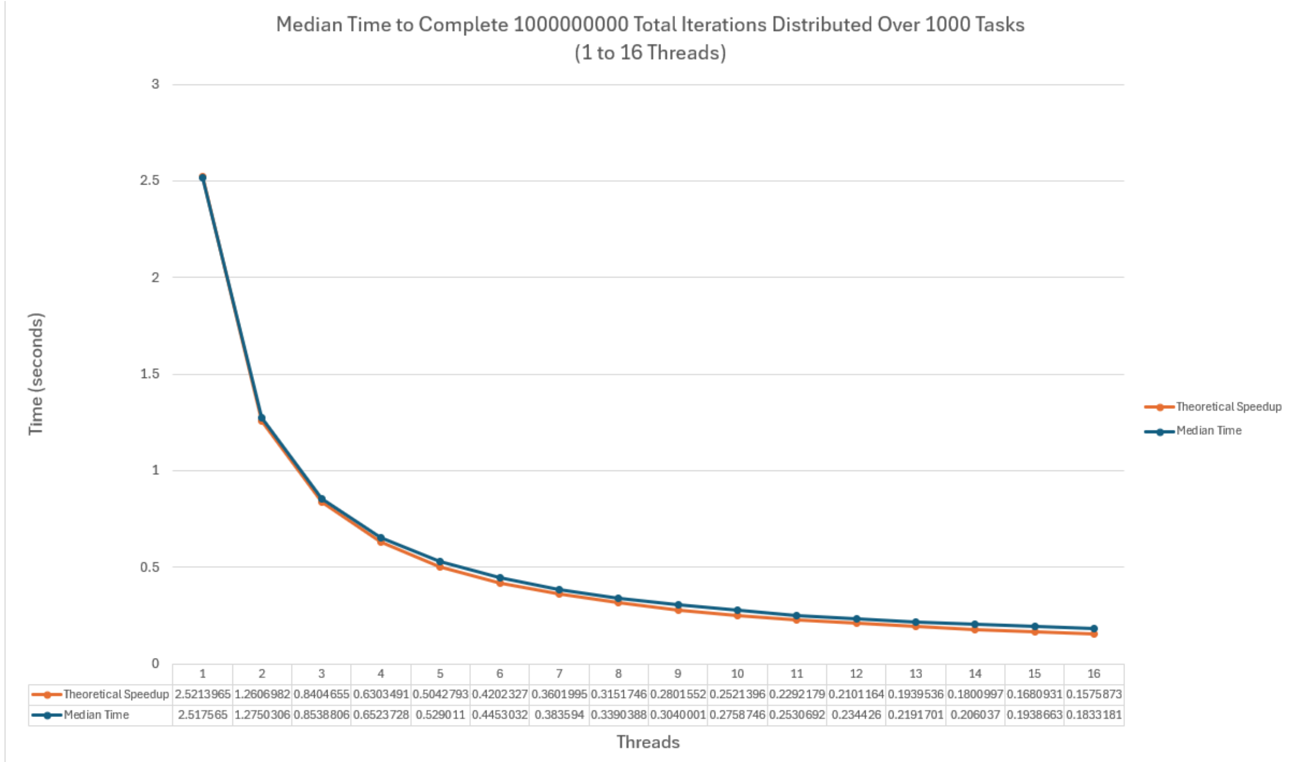
Analyzing Performance in <u>Increasing Thread Count</u>

Case 1: 1 000,000,000 Iterations, 1000 Tasks (Total iterations split into 10000 tasks)

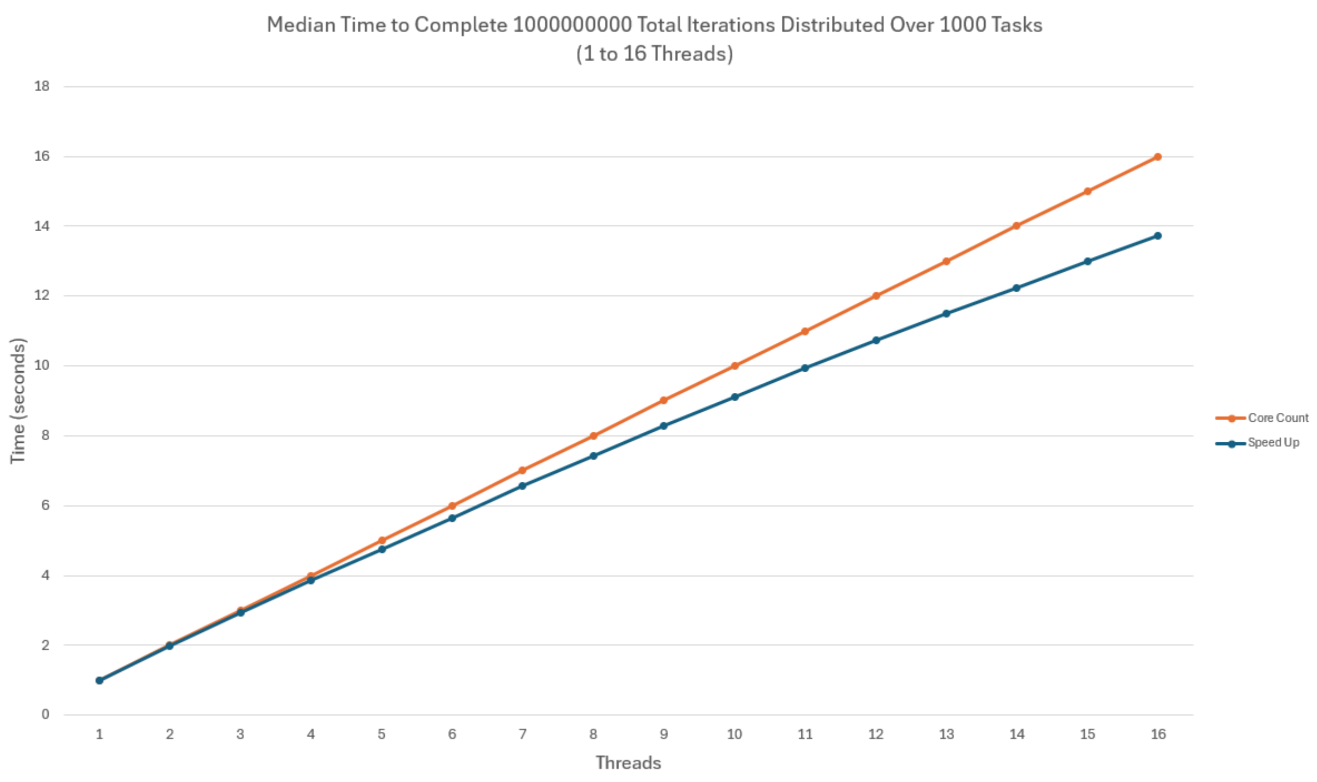
Tasks = 1000

Thread Counts 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

Repeats = 100, Plot median results



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Theoretical Speedup	2.5213965	1.2606982	0.8404655	0.6303491	0.5042793	0.4202327	0.3601995	0.3151746	0.2801552	0.2521396	0.2292179	0.2101164	0.1939536	0.1800997	0.1680931	0.1575873
Median Time	2.517565	1.2750306	0.8538806	0.6523728	0.529011	0.4453032	0.383594	0.3390388	0.3040001	0.2758746	0.2530692	0.234426	0.2191701	0.206037	0.1938663	0.1833181



Case 1 Average of results: 0.5540974

Case 1 Analysis

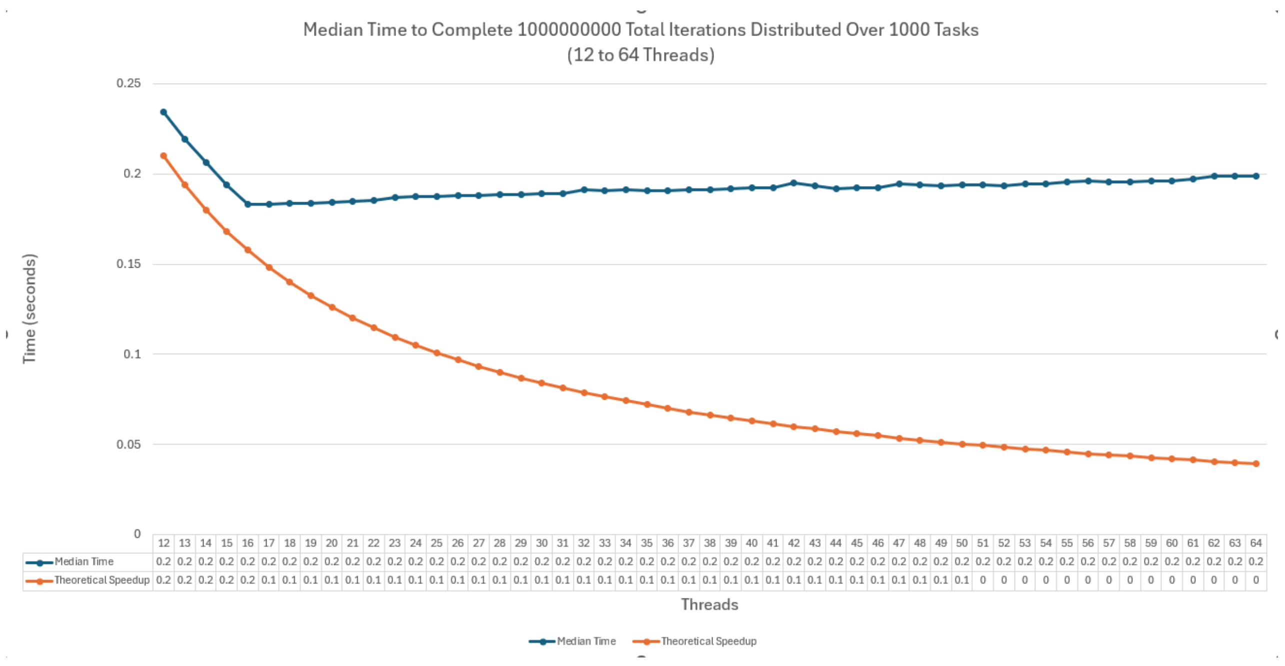
This test was used to determine how increasing thread counts impact performance for a set workload. In this experiment, 1,000,000,000 iterations were evenly split over 1000 tasks. Since this test was run on a 8 core, 16 thread CPU, this was also used to reveal the performance uplifts of hyperthreading. In the first graph, it is evident how increasing core counts drastically decreases performance time, but with not as major of an effect as thread counts increase. This is a result of the logarithmic complexity, as the workload is continuously divided by thread counts, its overall effects on performance uplifts begin to diminish. For example, when dividing a 100 by 2, it is a 50% decrease, but when further dividing the remainder 2 it is only a 25% decrease. To analyze the effects of multithreading, it is better shown in potential speedup, shown in chart two. Once past 8 threads, the speedup trend still follows that of the potential, although at a lower rate. This demonstrates that hyperthreading does play a major role in performance uplift, as when plotting the potential vs actual speedup past 16 threads in case 2, there is a major plateau in actual speedup as the program steps outside the processor hyperthread count.

Case 2: 1 000,000,000 Iterations, 1000 Tasks, Greater number of threads > processor

Tasks = 1000

Thread Counts 12,13,14…64

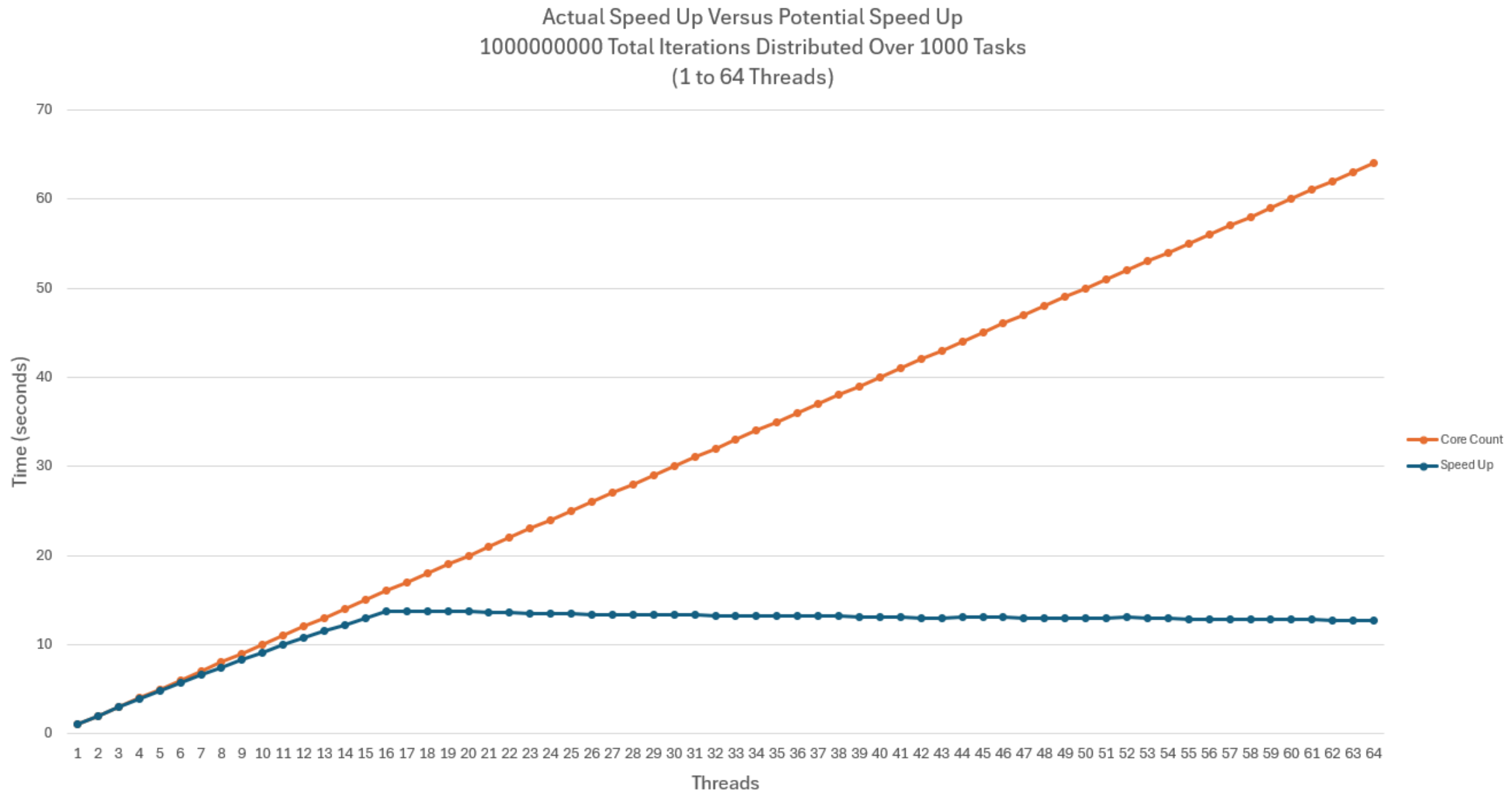
Repeats = 100, Plot median results



	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38
Median Time	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2
Theoretical Speedup	0.2	0.2	0.2	0.2	0.2	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1

	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
Median Time	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2
Theoretical Speedup	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Speedup Plot for 1-64 Threads



Case 2 Analysis:
Upon analyzing the data and graphs from this case, it becomes clear that the overhead increases as the number of threads grows. At a certain point, the number of threads being managed exceeds the available system cores, and if multithreading is supported, it eventually surpasses the number of supported threads. In this case, the processor supports hyperthreading, meaning that spawning more than 16 threads does not yield significant performance gains. Beyond this threshold, the overhead of managing additional threads increases while the benefits of parallelization diminish. This is evident in the graph above, as after 16 threads the relative improvement of the program plateaus. This indicates a cease in the benefits of parallelization beyond what the system can support. The cost associated with managing more and more cores also continues to grow, meaning at a certain point, depending on the program and problem being solved, a point may be reached where performance is actually lost compared to instances where a smaller amount of threads are utilized. This may be observed in the first graph above, as the median compute time trends upwards after the system threshold of 16 threads is reached.

Case 3: 1 000,000,000 Iterations, <u>Thread Based Task Count</u> (Extra)

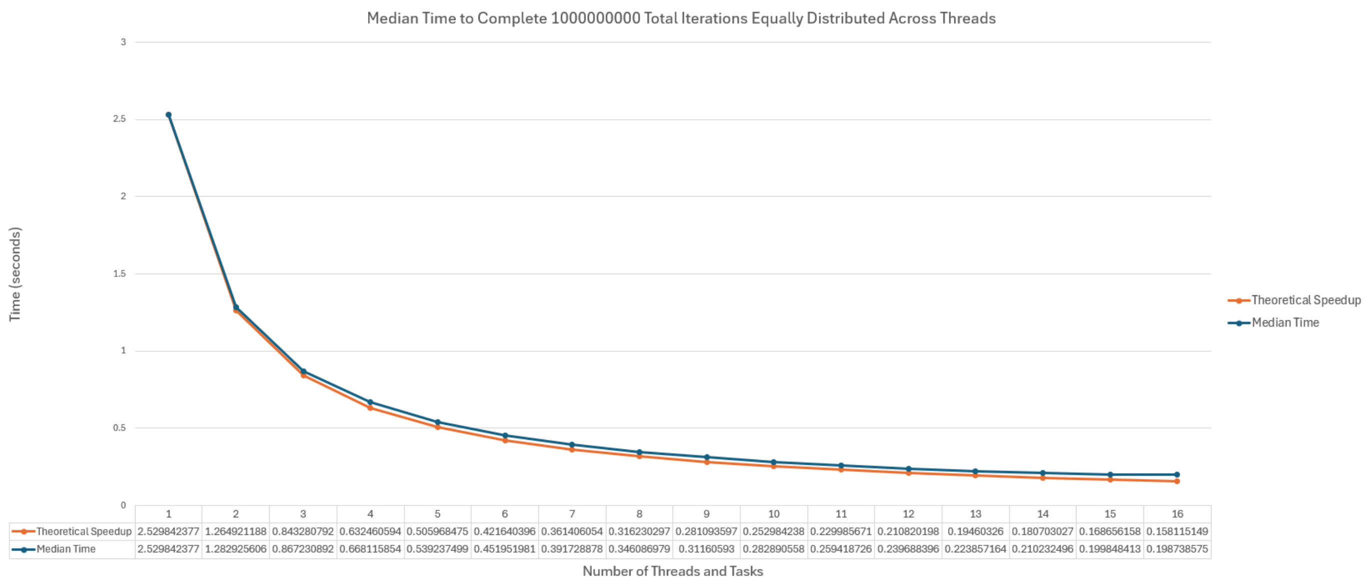
Tasks = Iterations / Thread Count + 1 (if remainder from division)

Thread Counts 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

Repeats = 100, Plot median results

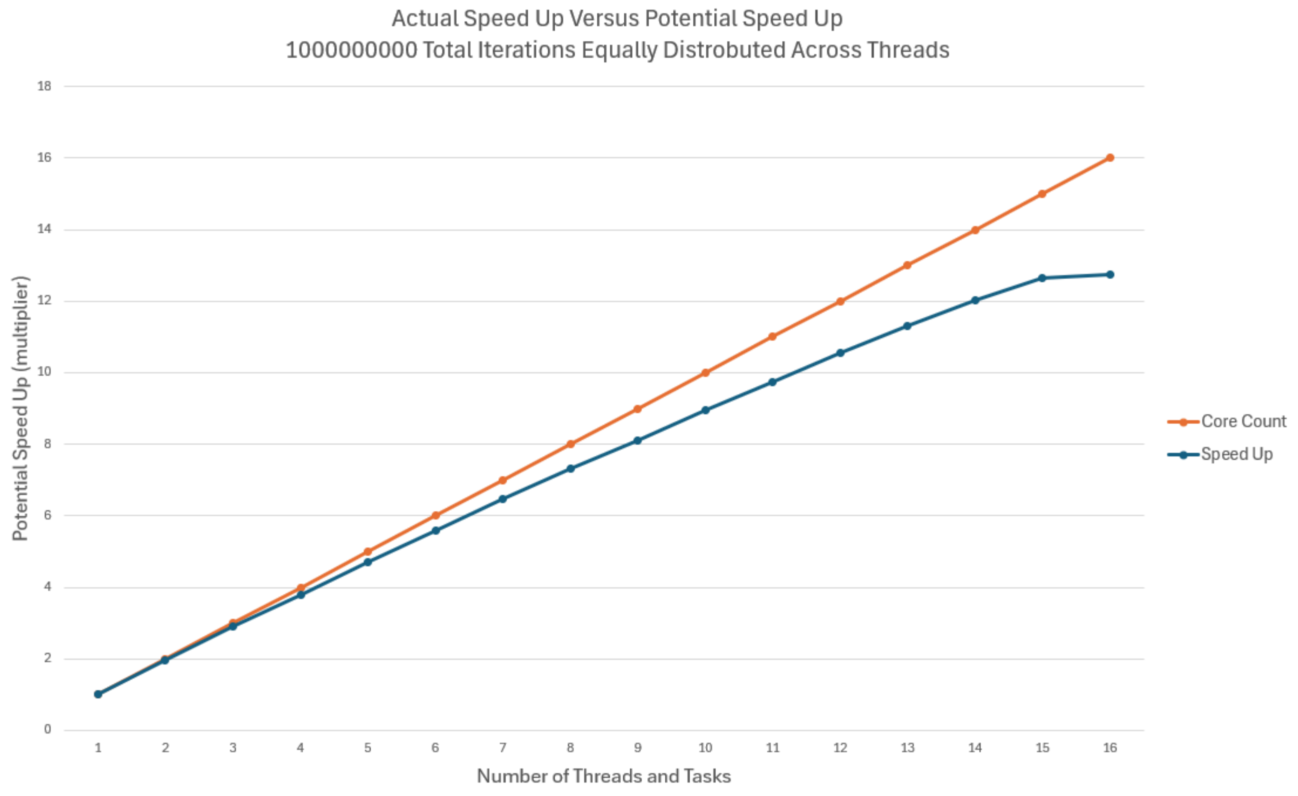
Total iterations split to individual task for each thread.

When dividing iterations to thread count, any remainder was added in a additional task (number of tasks exceeding thread count)



	1	2	3	4	5	6	7	8
Theoretical Speedup	2.529842377	1.264921188	0.843280792	0.632460594	0.505968475	0.421640396	0.361406054	0.316230297
Median Time	2.529842377	1.282925606	0.867230892	0.668115854	0.539237499	0.451951981	0.391728878	0.346086979

	9	10	11	12	13	14	15	16
Theoretical Speedup	0.281093597	0.252984238	0.229985671	0.210820198	0.19460326	0.180703027	0.168656158	0.158115149
Median Time	0.31160593	0.282890558	0.259418726	0.239688396	0.223857164	0.210232496	0.199848413	0.198738575



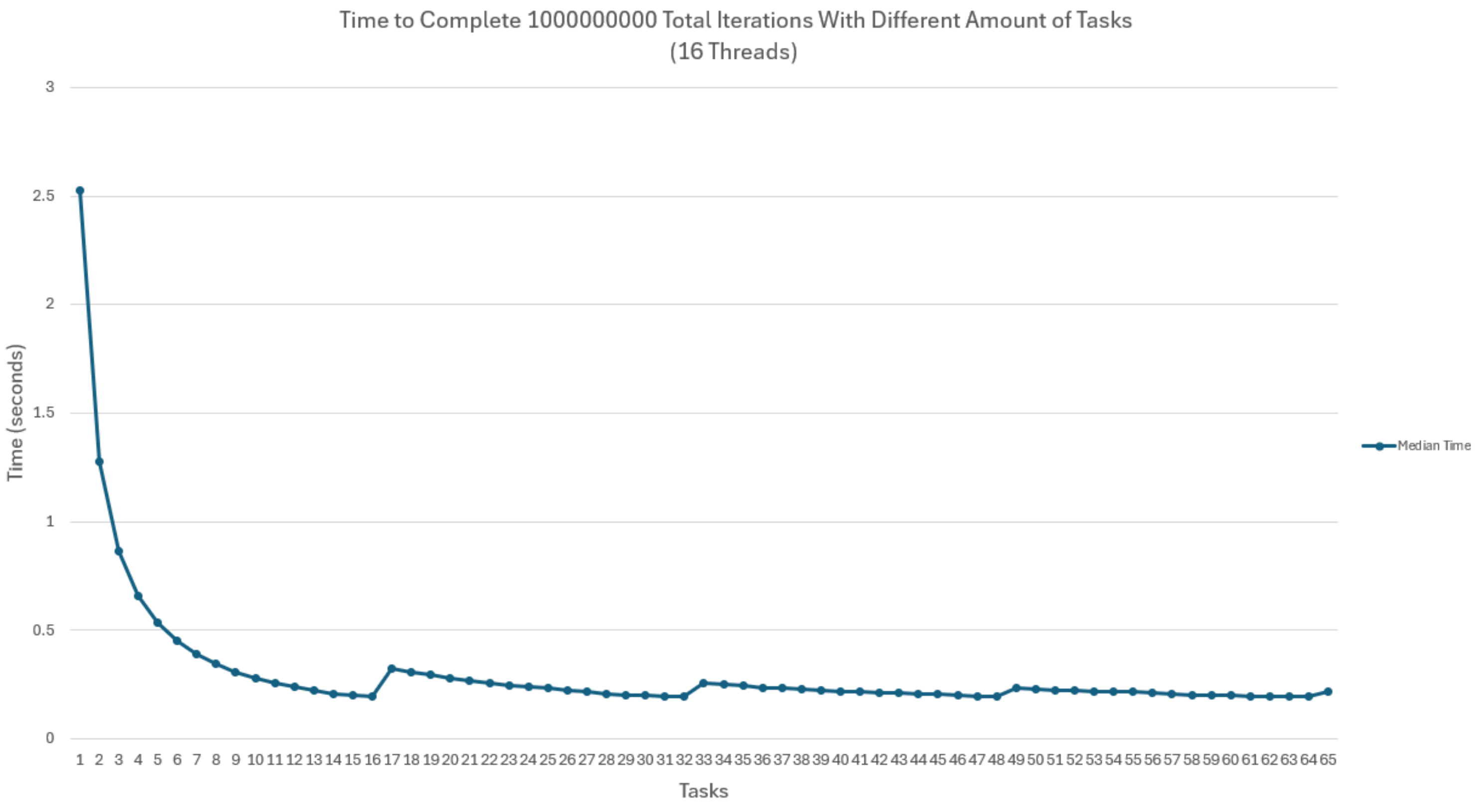
Case 3: Average of results 0.5627125

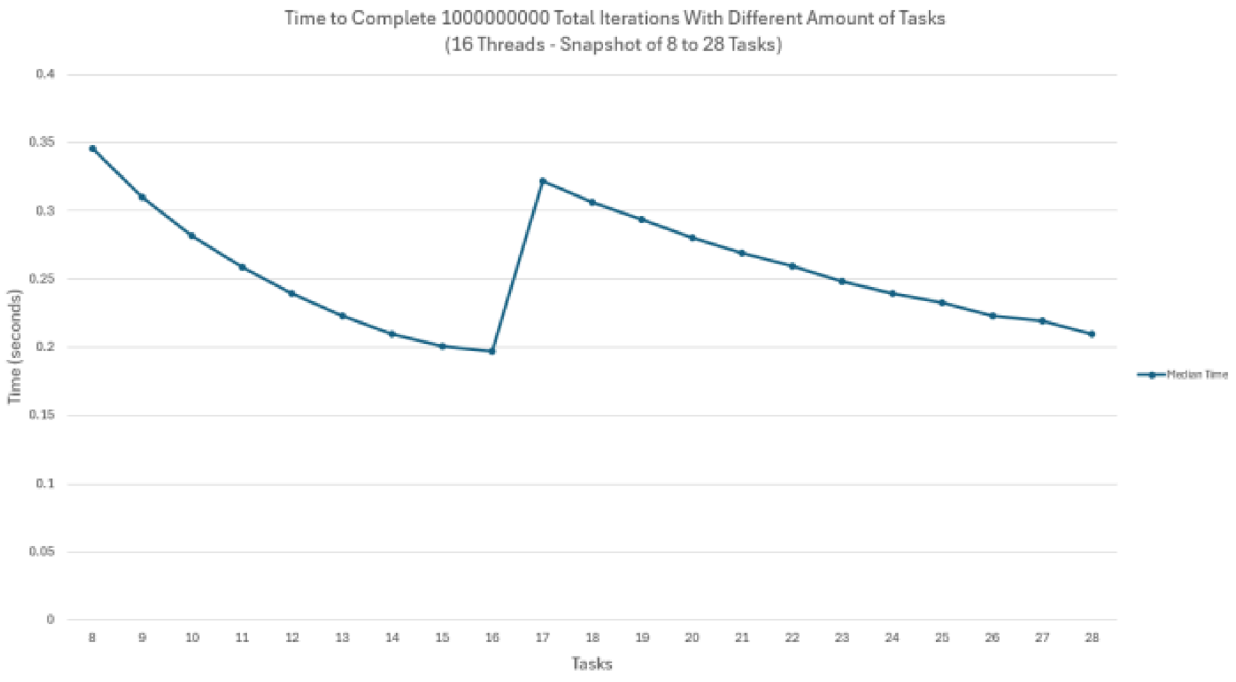
Case 3 Analysis

This test case was developed to analyze how task sizes that match the number of threads affects performance for a given workload. Once again, iteration size was set to 1,000,000,000 which was evenly distributed amongst the task sizes for each case. The results in this experiment follow a close resemblance to case 1, with the major outlier being the plateau in observed speedup in having 16 threads. Although 100 repeats were used for each thread pool, there are still a lot of factors that could have resulted in this trend. Maybe windows launched an update in the background, or another application was working on something. In the end, this case highlights how tasks pools equal to and greater than thread size follow similar performance uplift trends for the same workload.

The average for the results for case 1 were 0.5540974, while case 3 had an average time of 0.5627125. Not accounting for the outlier, the results from these two cases demonstrate how increasing task sizes past thread pool size can result in similar performance, all to a reasonable extent. These effects will likely veer off as task sizes are magnitudes larger than thread pool for a given workload, which proposes the need for another experiment. The effects of larger tasks sizes for a given thread pool are analyzed in Case 4.

Analyzing Performance in Increasing Task Count
Case 4: 1 000,000,000 Iterations, 16 Thread, Tasks 1-100
Tasks = 1,2,3,4..100 (Iterations evenly distributed to tasks)
Thread Count = 16
Repeats = 100, Plot median results
Same work, split into more and more tasks
When splitting iterations into respective task sizes, remainder is added to the final task.





Case 4 Analysis

This experiment was used to determine if a varying number of tasks would have a performance impact on a multithreaded application. It was done by having a constant thread pool of size 16, and 1,000,000,000 iterations split into each respective number of tasks, achieving the same amount of work. The results demonstrated that the number of tasks do affect performance, performing at best when evenly divisible by number of worker threads. This is most evidence from task numbers 8-28, where there is a major jump in execution times between tasks 16 and 17. This increase in time results from the program hanging on a single thread of execution, as the program has to wait for a single thread to complete its calculation whilst all other threads sleep. Whilst a more even distribution of workload, seen in executing a program with 16 tasks, can increase performance as all threads complete execution around the same time. This remaining work gets distributed more evenly as task numbers approach a divisor of thread count, leading to faster performance. This is evident in the tasks 17 - 32, as the tasks begin to reach a divisor of thread count, the load becomes more evenly distributed; resulting in a better runtime.

Analyzing Performance in Increasing Iteration Count

Case 5: Increasing Iterations, 8 Threads, 8 Tasks

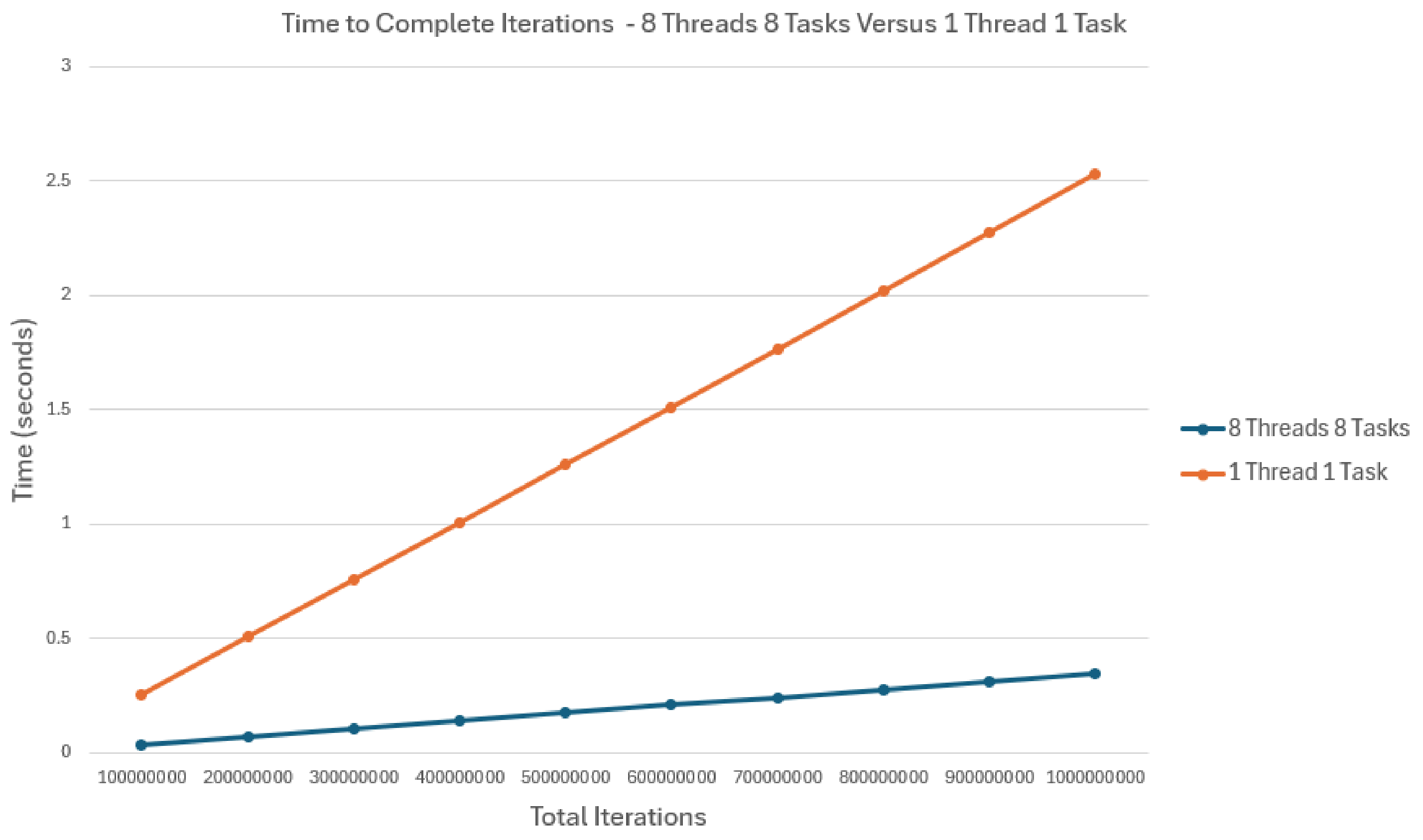
Iterations = **100,000,000, 200,000,000 … 1 000,000,000**

Tasks = 8

Thread Count = 8

Repeats = 100, Plot median results

When splitting iterations into respective task sizes, remainder is added to the final task.



Case 5 Analysis

In this test, we compared the performance of a single-threaded program with 1 task to a multithreaded program with 8 threads handling 8 tasks. The results indicate that as the number of iterations increases, the single-threaded version takes longer to complete the full set of iterations, with execution time increasing steadily. On the other hand, the multithreaded version maintains a much lower and relatively consistent execution time, even as the number of iterations grows. This is due to the fact that the number of iterations may be split between more threads. This suggests that using multiple threads allows the workload to be divided more efficiently, while the single-threaded approach struggles as the number of iterations increases.

Performance Analysis Conclusions

Overall, the results from this report demonstrate a best case scenario for performance uplifts as a result from parallel programming. This is because there is no combining step required with the data produced from each thread, allowing workload to be easily split amongst each worker thread. This allowed program speedup to closely follow the theoretical limits, only being limited by scheduling and thread safe overheads (Case1). Thread based programming is always limited by the CPU, which was evident in Case 2. As threads count continues past the constraints of the CPU, performance slowly begins to decrease. The amount of tasks also can impact performance, as it may leave threads strangling as the rest of the pool waits. This is shown as a worst case when there is a remainder one task, leaving the entire thread pool waiting for the singular thread to complete an equal share of the work. Finally, when comparing the results of threaded versus multi threaded programs for increasing workload, threaded applications have a major performance advantage. As seen in case 5, serial programs follow a linear trend, whilst threaded programs have a logarithmic one. In the end, the results from this analysis showcase the sheer complexity of the world of parallel programming. These experiments demonstrate the importance of statistical analysis, as programmers chase the bleeding edge of performance that can be achieved through parallel programming.