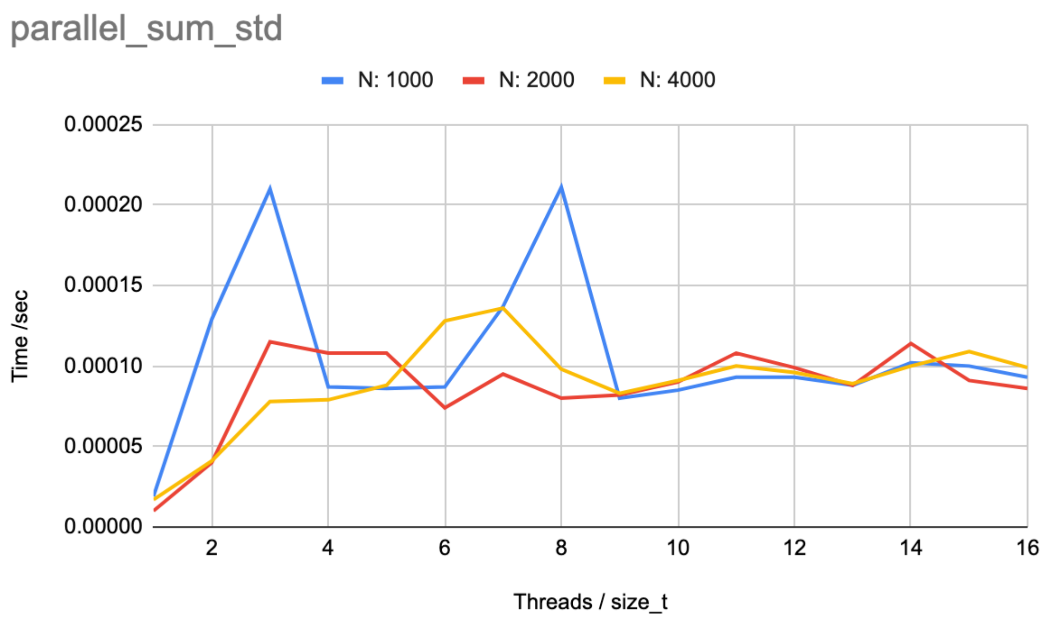
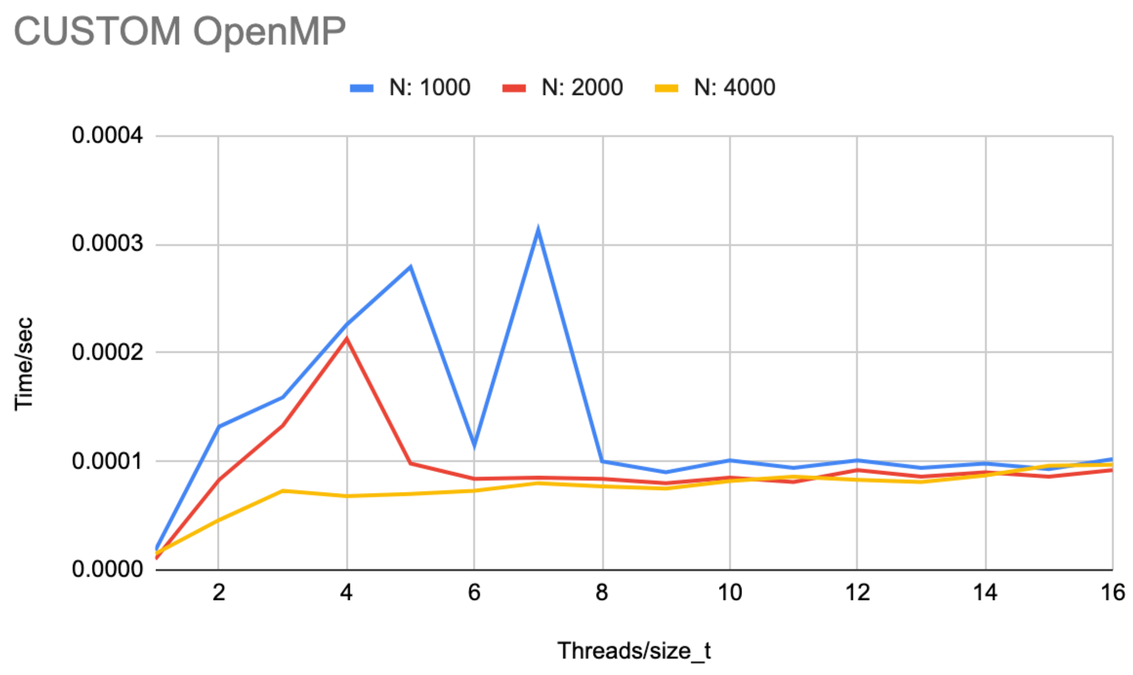
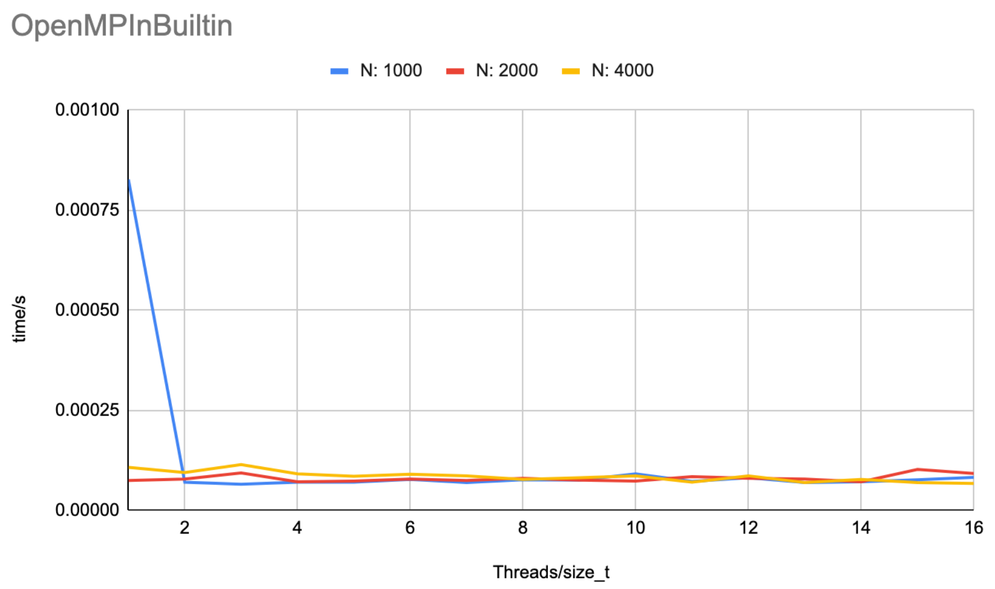
OpenMP analysis File.

Strong Scaling

1. The graphs were plotted and the following results were observed.





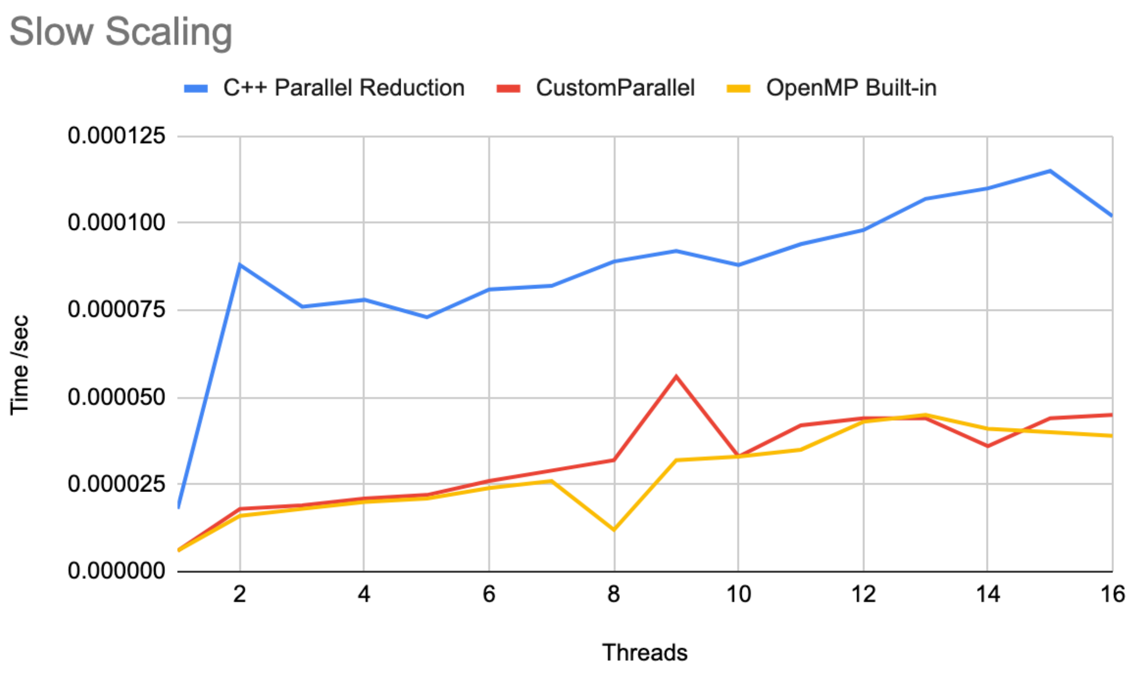


Ideally, an efficient algorithm would demonstrate a decreasing trend in time taken as the number of threads increases. However, our findings deviated from this expectation.
For the C++ parallel reduction algorithm, the initial timing was lower with fewer threads. As the thread count increased, the cost of adding more threads surpassed the benefits of parallelism, leading to high spikes in the

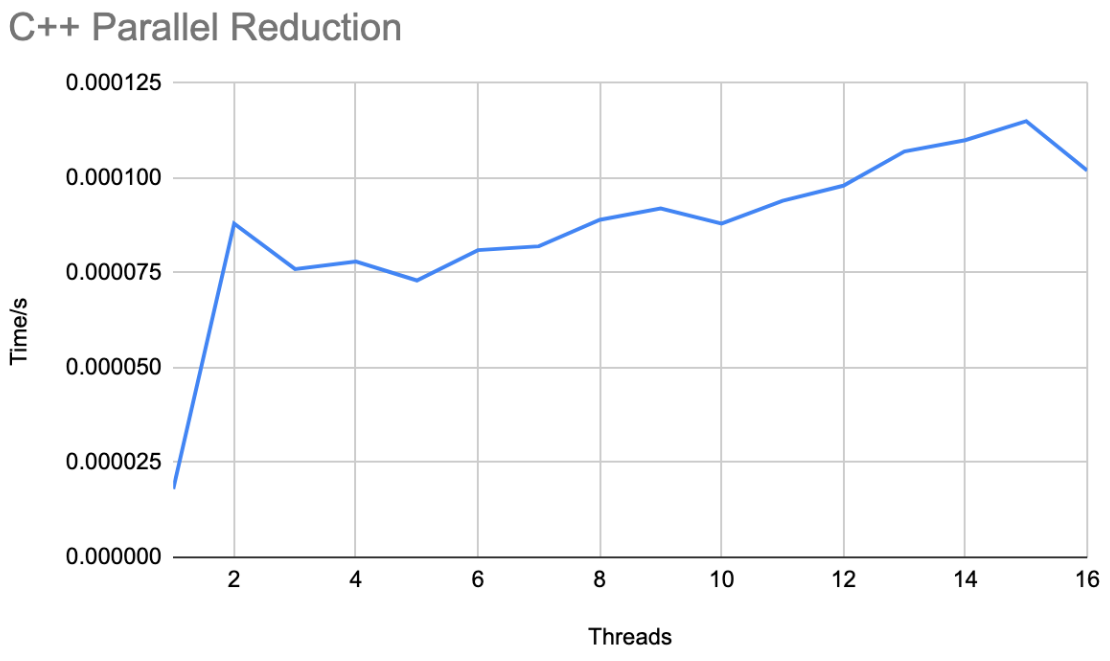
timings. Nevertheless, the time taken decreased as the thread number continued to increase. Possible explanations for this include CPU scheduling, where multiple threads are scheduled at different times, and the use of locks to safeguard critical sections. Locks are known to consume time when locked and unlocked, which can affect the execution time of algorithms.
In contrast, the custom OpenMP algorithm demonstrated faster timings with increasing thread counts. This can be attributed to the introduction of OpenMP parallelization, which handled thread parallelism, and the use of #pragma omp critical to protect the critical section and prevent race conditions. This ensured that the guarded code block was executed by one thread at a time, preventing simultaneous execution by multiple threads. The observed spikes were in line with our expectations, given that CPU scheduling handles numerous programs running on the machine.
Finally, the built-in method was the fastest among the thread methods. This can be attributed to its efficiency in running everything in the background. However, at times, it was slow with many threads, possibly due to the creation of unnecessary threads or the time required to set up the method.

In Weak scaling

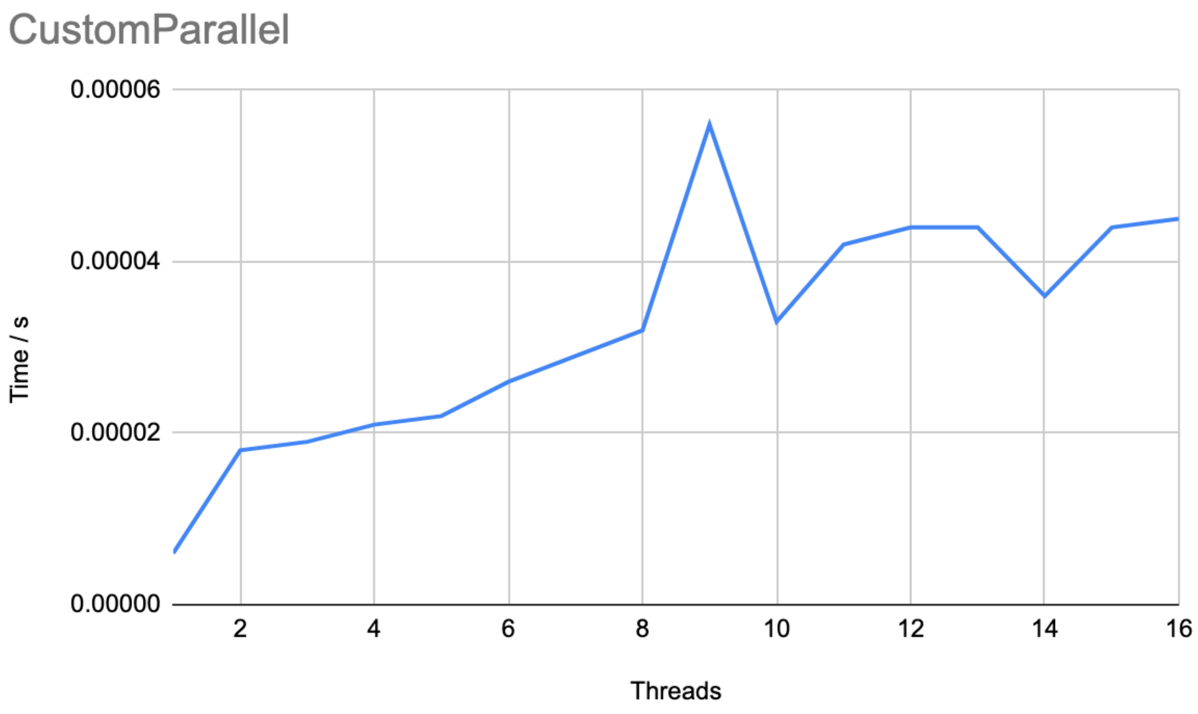
we increased the value of N and the number of threads simultaneously, and expected the time taken to remain roughly constant. The following graph illustrates our observations.
Overall, the timings remained consistent with a few variations. The C++ parallel reduction algorithm experienced a significant spike at the beginning, which is consistent with the results of Strong Scaling. The in-built OpenMP algorithm demonstrated the fastest execution time across all variations. Additionally, both the in-built and custom OpenMP algorithms maintained a roughly constant execution time, which is in line with our expectations.

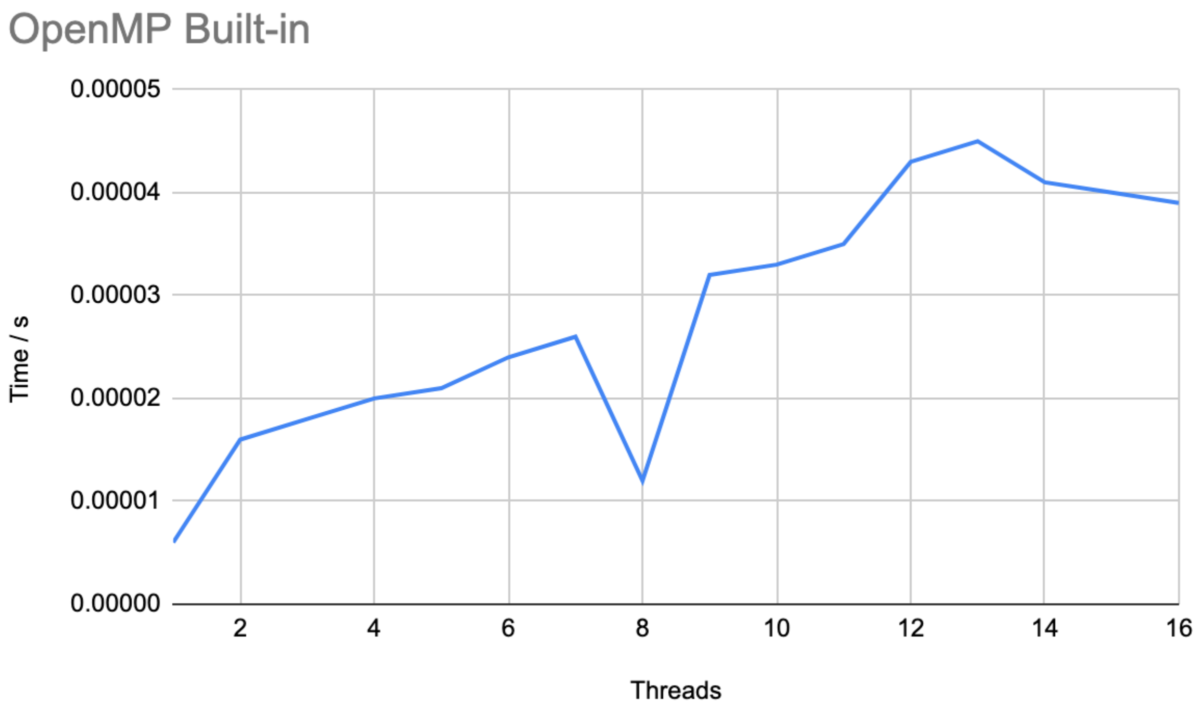


Slow Scaling
C++ Parallel Reduction
CustomParallel
OpenMP Built-in
Time /sec
0.000125
0.000100
0.000075
0.000050
0.000025
0.000000
2
4
6
8
10
12
14
16
Threads



C++ Parallel Reduction
Time/s
0.000125
0.000100
0.000075
0.000050
0.000025
0.000000
2
4
6
8
10
12
14
16
Threads


CustomParallel
0.00006
0.00004
0.00002
0.00000
Time / s
2
4
6
8
10
12
14
16
Threads


OpenMP Built-in
0.00005
0.00004
0.00003
0.00002
0.00001
0.00000
Time / s
2
4
6
8
10
12
14
16
Threads

Overall, the OpenMP Built in is the fastest one compared to the other two methods.