APROP - REPORT 2

Parallelism Benchmarks with Rust

Henrique Teixeira (1200883)
Alexander Paschoaletto (1222703)

Abstract

The current document presents the Rust implementation and execution results obtained for Group Evaluation #2. It consisted of calculating the area of a Mandelbrot set in three processing strategies: sequential execution, parallel execution with the Rayon library, and parallel execution with the Threadpool library. Tests were executed with different resolutions (N) and, in the case of the thread-based strategies, different block sizes (BS) and thread numbers (NUM_THREADS) as well. The analysis showed that both Rayon and Threadpool were consistently faster than the sequential alternative, and that the performance between them remained somewhat close throughout the different testing configurations.

Introduction

Rust

Rust is a compiled language officially released in May 2015 that features memory safety and performance as the main features. It is currently adopted by Mozilla, the primary project sponsor, as well as Discord, Tauri and the Linux Kernel. The memory safety is achieved through the concept of data ownership, which ensures that common programming errors such as null pointer dereferencing and race conditions are caught at compile time (also removing the need for a runtime garbage collector). Moreover, as a compiled language, Rust yields executable files that match the low-level performance of C, making it an adequate choice for modern software development.

Concurrency, Threadpool and Rayon

Rust also differs from C by having native support for concurrency with threads and message channels for exchanging information between them. Mutexes, conditional variables and barriers are also supported, but these features are still somewhat basic in terms of functionality. As a consequence, the Rust community has developed libraries with a more complete set of features for concurrency.

The two target libraries of this report are Threadpool and Rayon, which focus on the more flexible task-based concurrency: the parallel code generates X tasks to be executed by Y threads. Threadpool provides a fixed-size pool of worker threads to execute concurrent tasks, offering a straightforward approach to managing a limited number of threads. Rayon extends this approach with a work-stealing algorithm, which dynamically balances the load across threads and thus contributes to a more efficient CPU utilization on heterogeneous workloads.

Test Scope

This test aims at evaluating the performance of three strategies for calculating the area of a Mandelbrot set: a sequential execution, which runs the math on a basic for-loop; a Threadpool execution, which splits the for-loop iterations into tasks and handle those to a thread set; and a Rayon execution, which applies the same concept as the Threadpool but with its own inner implementation (which also features the aforementioned work-stealing algorithm). Each execution yielded computation results close to 1.510659, which is the expected outcome for the Mandelbrot set.

The execution considered variations in resolution (*N*), block size (*BS)* and thread number (*NUM_THREADS*), which ultimately rendered 24 combinations to be tested. Those can be seen in Table 1. It should be noted that, for sequential execution, only the resolution (N) is relevant for the analysis, as block size and thread numbers are only relevant for parallel execution.

N	BS	NUM_THREADS
1000	50	2
		4
		8
		16
	100	2
		4
		8
		16
	200	2
		4
		8
		16
2000	100	2
		4
		8
		16
	200	2
		4
		8
		16
	400	2
		4
		8
		16

Table 1: test combinations.

Each calculation of the Mandelbrot set had 1000 iterations of floating-point math operations, and each combination of Table 1 ran 50 times. Thus, the results presented are based on the average of these runs.

Results

We now proceed to analyze the results obtained with the tests. They were conducted in two Windows 11 machines whose main hardware features were:

1. Intel Core I5 10300H (x86, 4 cores, 8 threads, 8MB cache) and 16GB of RAM;
2. Intel Core I7 11370H (x86, 4 cores, 8 threads, 12MB cache) and 16GB of RAM.

As the following results will demonstrate, both parallel executions yielded significantly faster computation results than the sequential alternative, mostly due to the absence of shared data between threads or the need of precedence order between cycles. If present, those features could lead to race conditions and therefore undermine the benefits of parallelism.

Machine 1 (Core I5 10300H)

The average execution times for the sequential algorithm were 4.0333s for N=1000 and 12.0937s for N=2000, although, for this algorithm, the individual runs featured a considerable deviation: for example, with N=2000, the shortest sequential run was of 10.5773s, and the longest, 15.3768s, ultimately leading to a standard deviation of 1.1044 seconds. For the parallel implementations values were much more consistent, with standard deviations rarely surpassing the 0.2 second mark.

Figures 1 to 6 showcase the average performance observed for each algorithm according to the resolution, the number of threads employed and the block size for each thread to process. The numbers show that splitting the work into threads is very effective and yields great improvements for any number of threads selected. Interestingly enough, when the comparison is made between both parallel algorithms, the Threadpool approach offers better average performance than Rayon for any given combination. This is likely explained by the extra overhead Rayon offers for enabling the work stealing algorithm, and such is not very useful on a scene where the generated tasks are relatively equal in terms of computational complexity. The optimal value seems to be 8 threads (which is equal to the hardware threads of the I5 10300H processor) and a block size of 10% of N (100 for N=1000, 200 for N=2000), as those values gave rendered nearly optimal performance. More threads actually worsened the execution times on some cases (e.g. BS = 100 and N=1000).

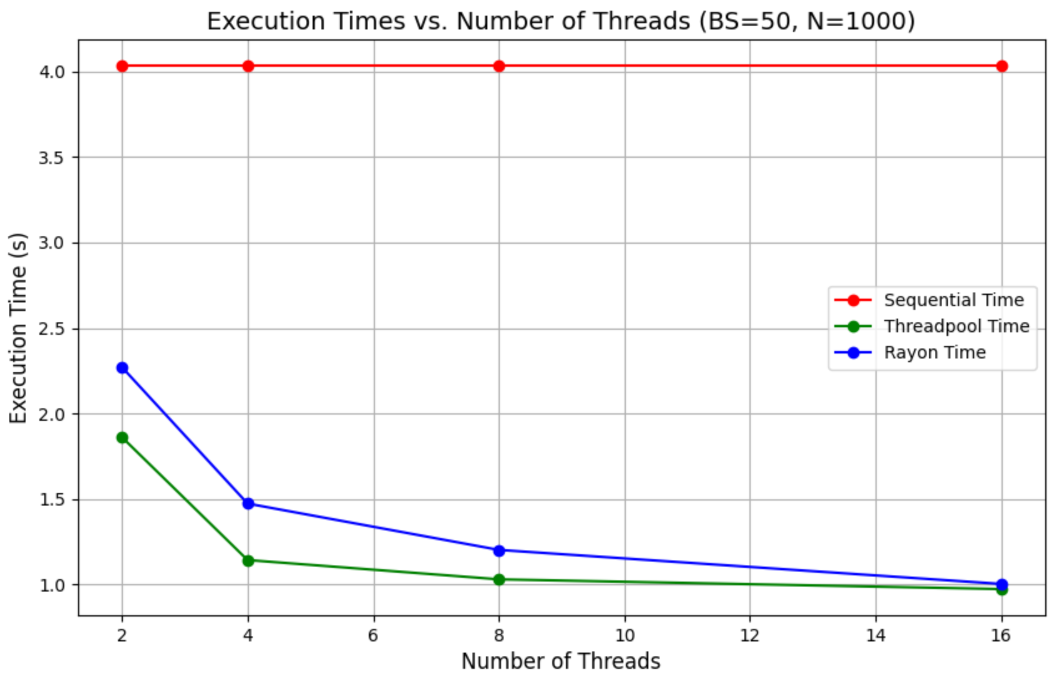

Figure 1: Execution times with BS=50 and N=1000.

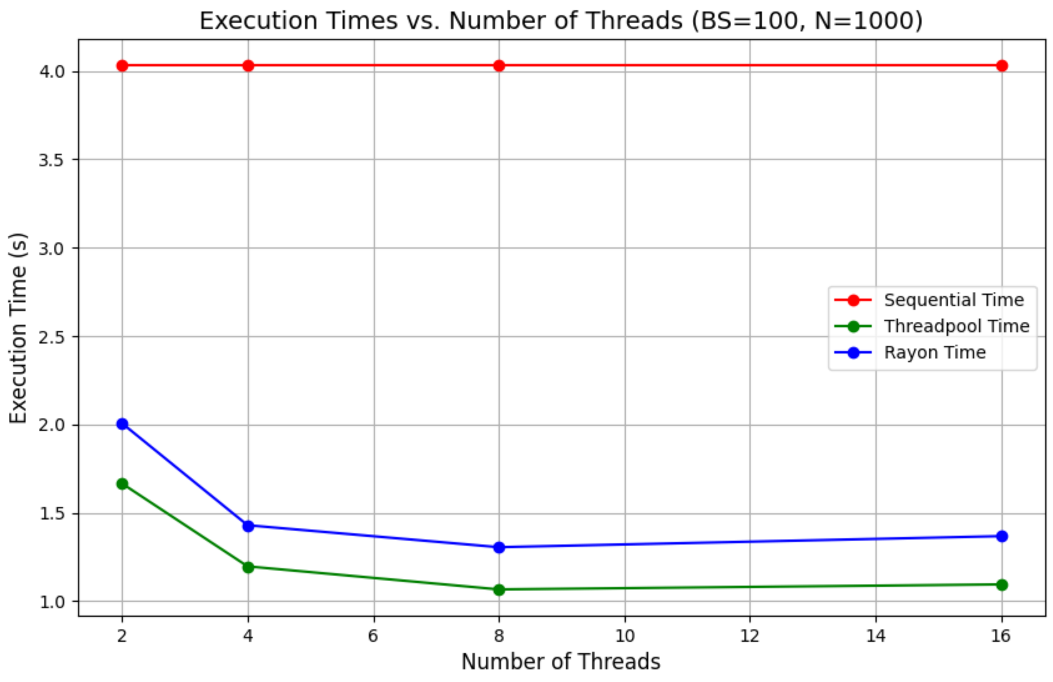

Figure 2: Execution times with BS=100 and N=1000.

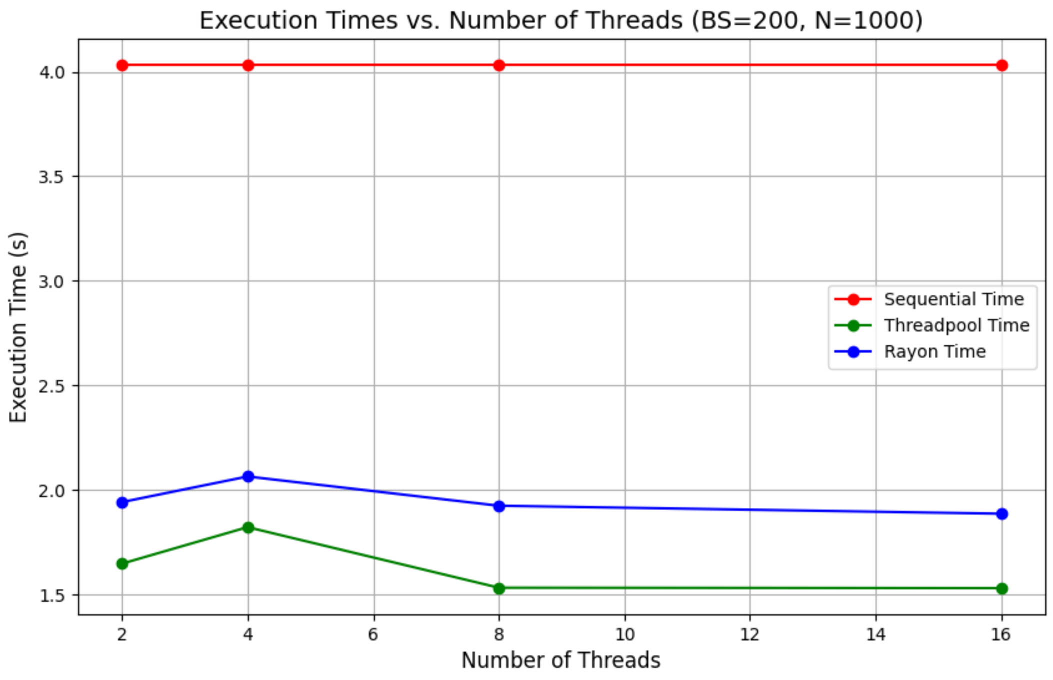

Figure 3: Execution times with BS=200 and N=1000.

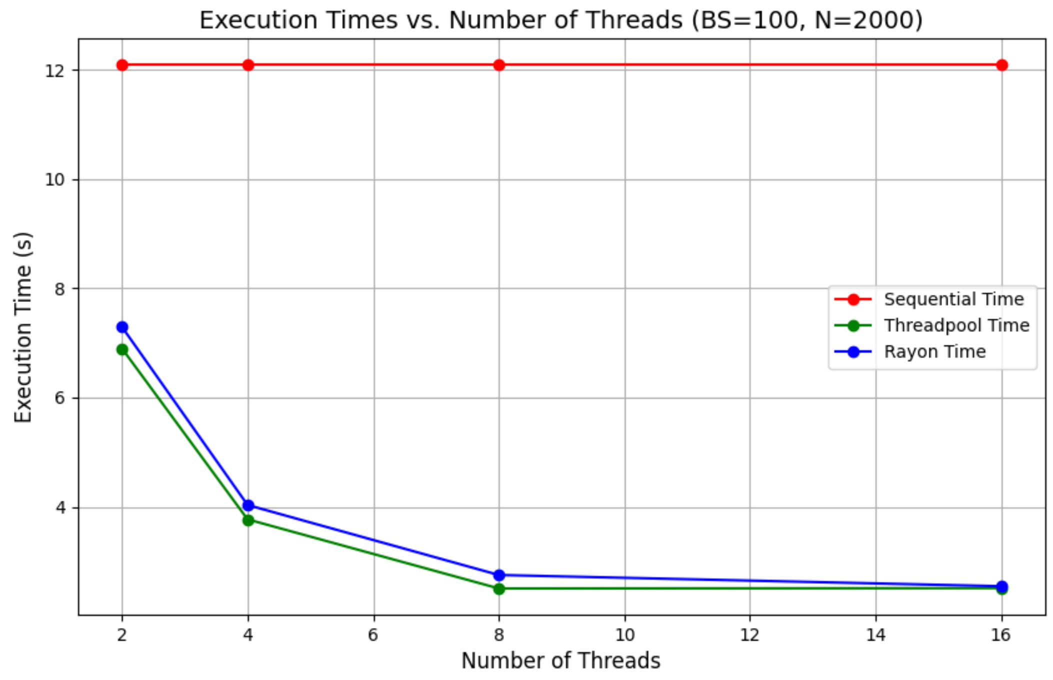

Figure 4: Execution times with BS=100 and N=2000.

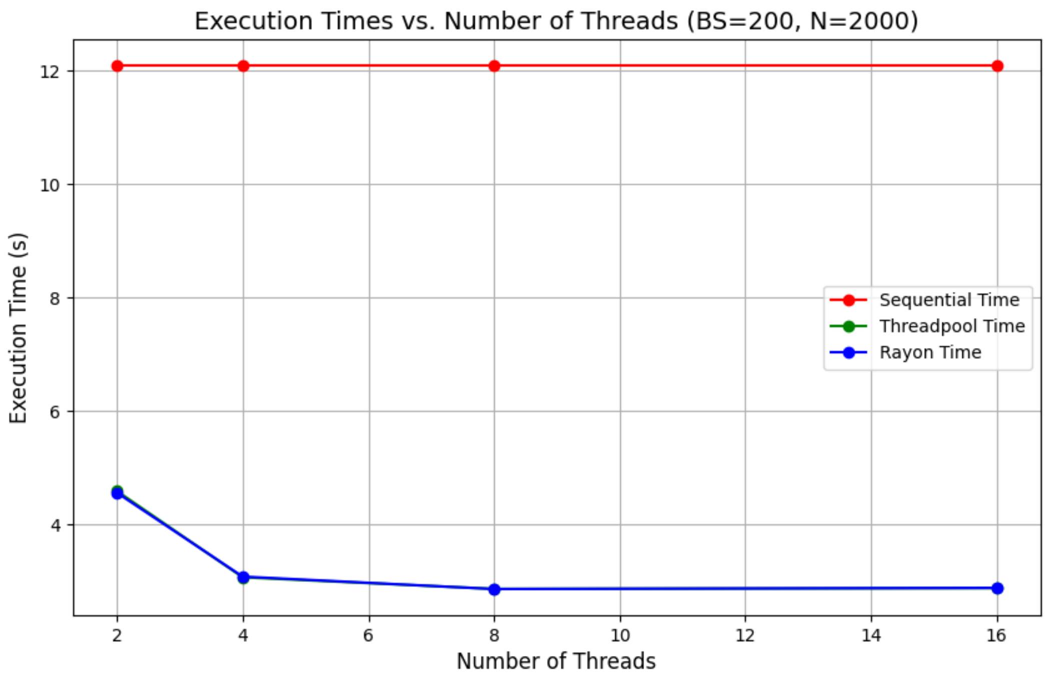

Figure 5: Execution times with BS=200 and N=2000.

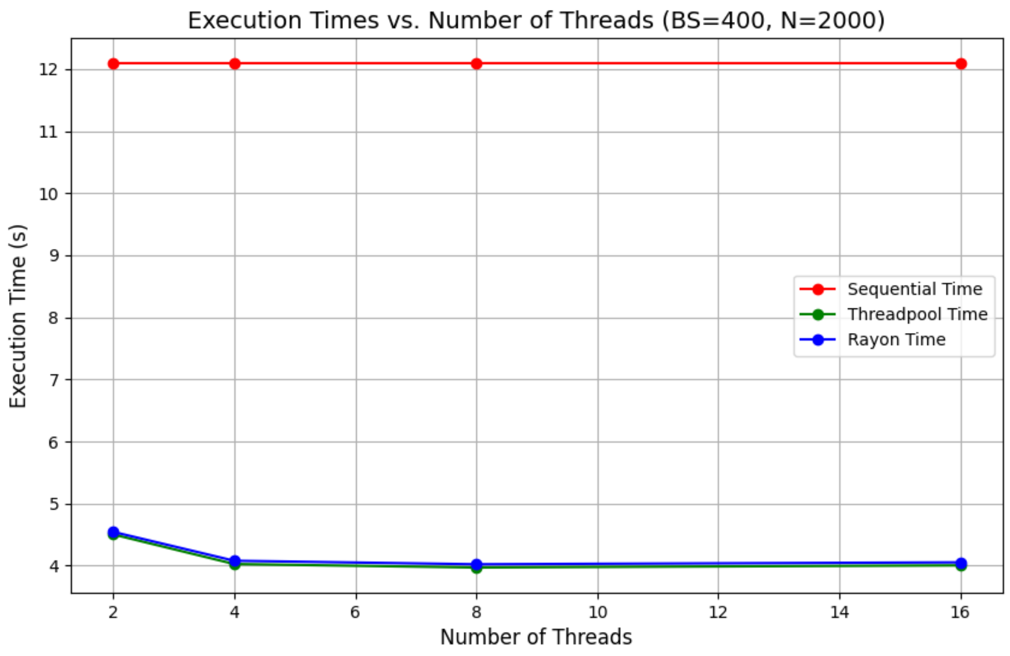

Figure 6: Execution times with BS=400 and N=2000.

Machine 2 (Core I7 11370H)

The system efficiently leverages parallel computing to significantly outperform the sequential algorithm across all configurations. For sequential runs, the average execution time for N=1000 was around 1.75s, while the parallel implementations consistently achieved times under 1 second, regardless of the number of threads. The Threadpool approach exhibited better performance compared to Rayon in most cases, likely due to reduced overhead and a more straightforward task scheduling mechanism.

The data in both files highlights that Threadpool's execution times are not only faster but also more consistent, with deviations rarely exceeding a few milliseconds. For example, in results.csv, Threadpool execution times for N=1000N=1000, Block Size=50, and 2 threads varied between 907.894ms and 1.06s, with minor fluctuations across runs. Rayon, while still faster than sequential execution, showed slightly higher deviations in the same configuration, ranging from 902.509ms to 1.008s. These results suggest that Rayon's work-stealing algorithm introduces some overhead that is less effective for computationally uniform tasks.

From the averages presented in averages.csv, the optimal configuration for the system aligns with using 8 threads (equal to the hardware threads of the I5 10300H processor) and a block size of approximately 10% of NN (BS = 100 for N = 1000 and BS = 200 for N = 2000). This balance ensures that each thread processes a manageable workload without incurring significant overhead. For instance, with N=2000, the optimal block size of 200 yielded near-minimal execution times while maintaining a standard deviation under 0.2 seconds across runs.

Interestingly, increasing the thread count beyond the hardware's physical capabilities (e.g., 16 threads) often resulted in diminishing returns or slightly worse performance. This effect was more pronounced for smaller block sizes, where thread contention and scheduling overhead outweighed the benefits of additional parallelism.

Figures 7 to 12, accompanying the analysis would show that the system achieves optimal performance by carefully balancing the number of threads and block size. The parallel implementations consistently outperform the sequential approach, but the performance gains are highly dependent on the workload distribution and hardware constraints.

The system demonstrates impressive performance improvements through parallelization, particularly with the Threadpool implementation. By aligning the configuration with the hardware's capabilities and selecting appropriate block sizes, the system achieves near-optimal execution times with minimal variability, making it a robust solution for tasks requiring efficient and consistent performance.

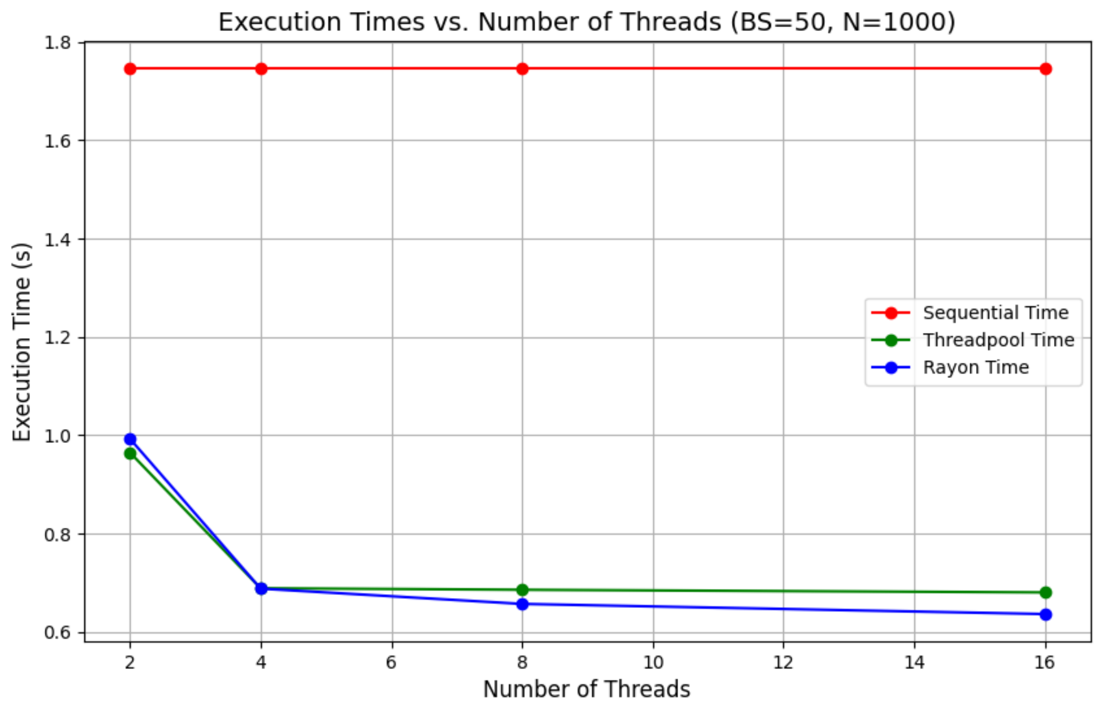

Figure 7 - Execution times with BS=50 and N=1000.

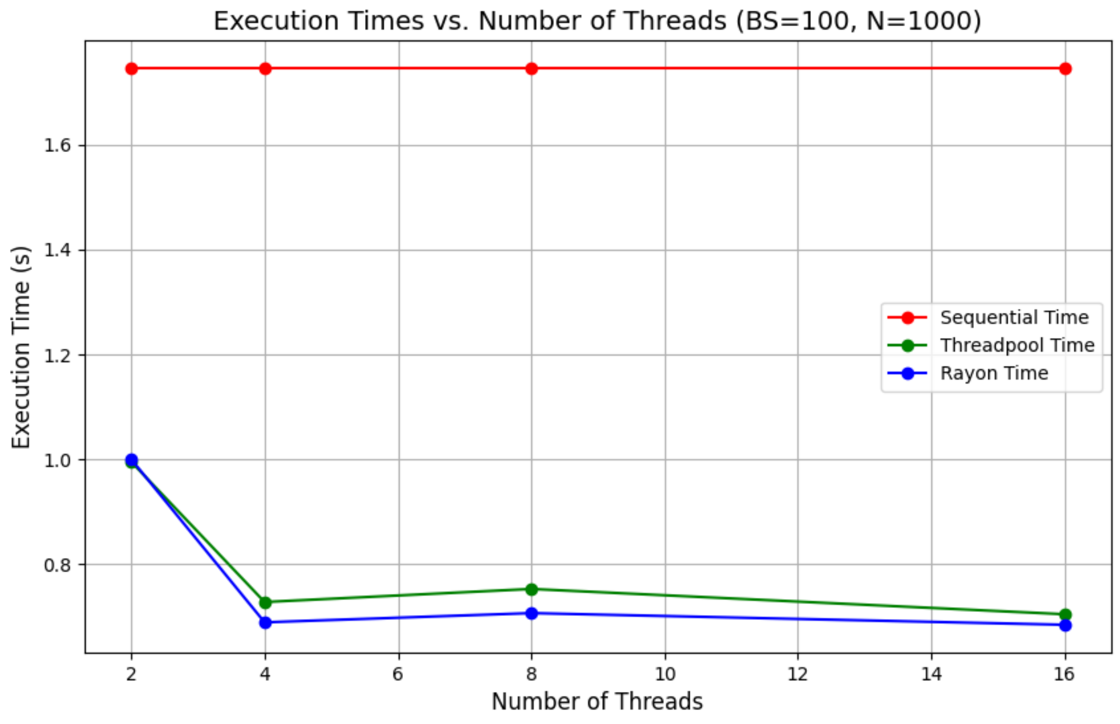

Figure 8 - Execution times with BS=100 and N=1000.

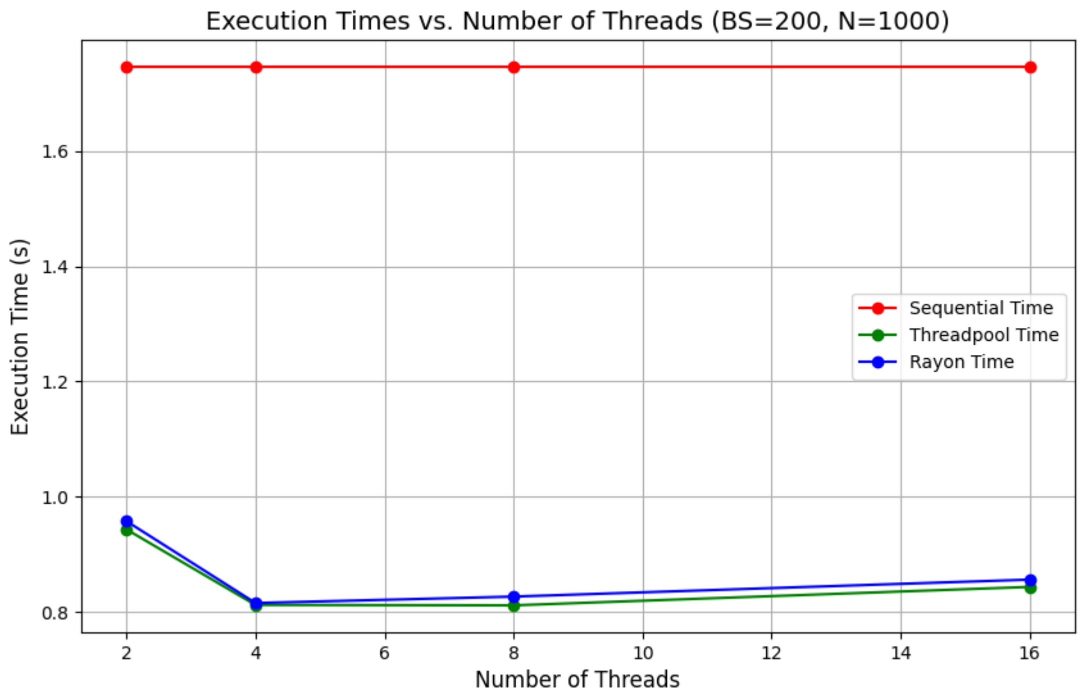

Figure 9 - Execution times with BS=200 and N=1000.

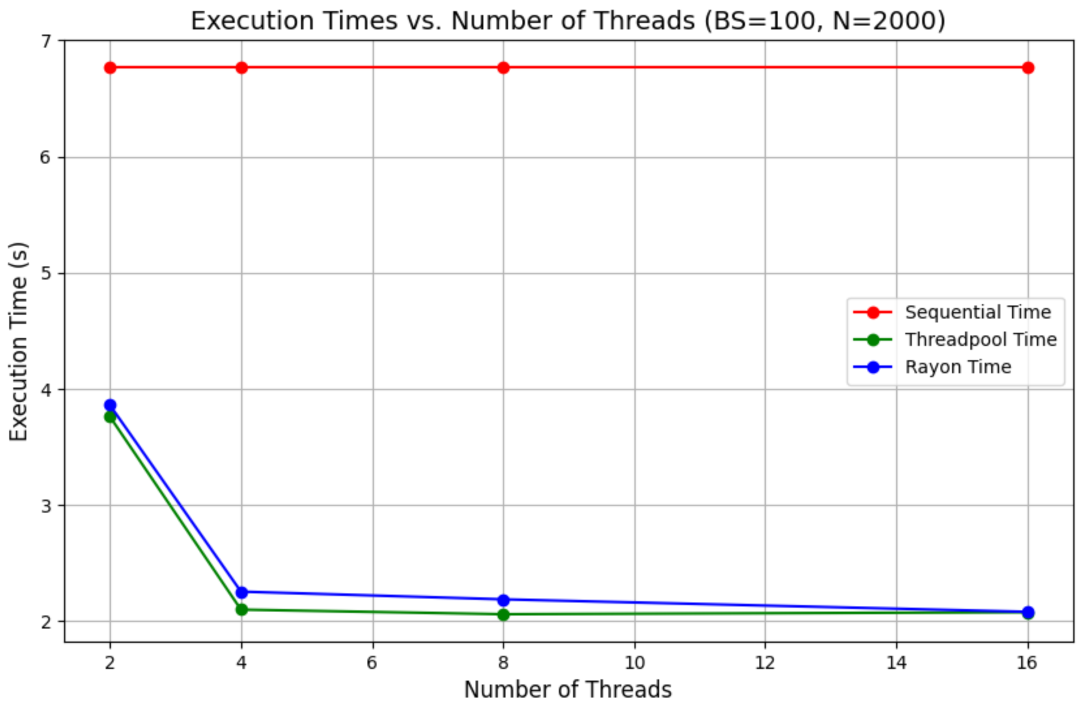

Figure 10 - Execution times with BS=100 and N=2000.

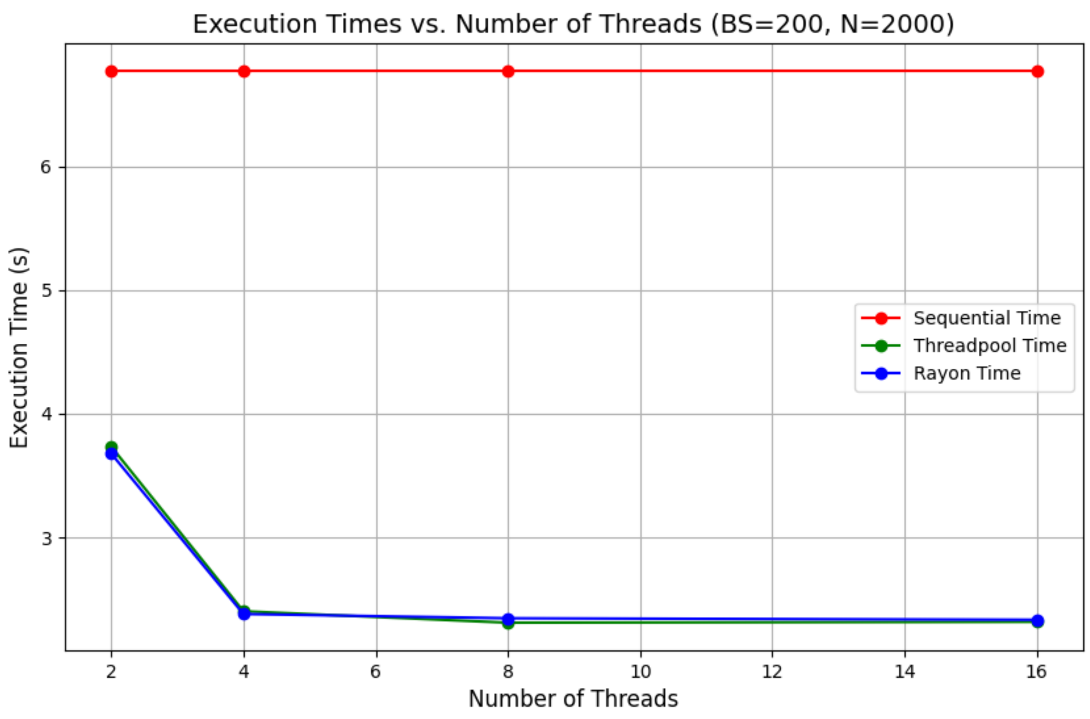

Figure 11 - Execution times with BS=200 and N=2000.

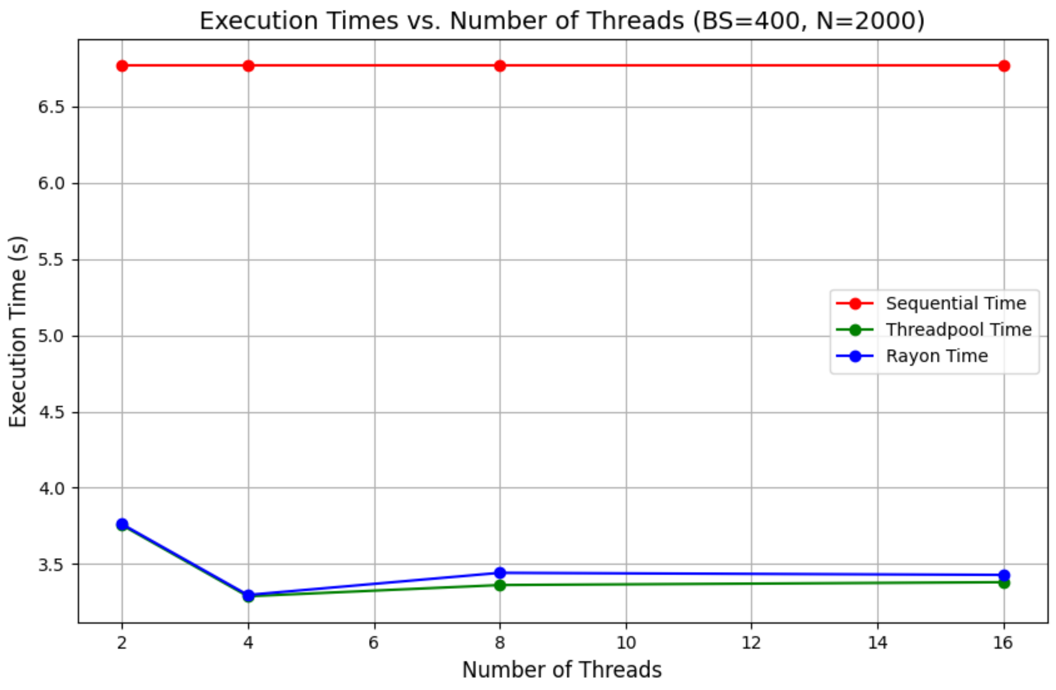

Figure 12 - Execution times with BS=400 and N=2000.

Conclusion

The performance analysis of the sequential, Threadpool, and Rayon implementations on two systems—an Intel Core I5 10300H (4 cores, 8 threads, 8MB cache) and an Intel Core I7 11370H (4 cores, 8 threads, 12MB cache)—reveals consistent trends in execution times and scalability. The sequential implementation exhibited the slowest performance on both systems due to its inability to leverage parallelism, with execution times increasing significantly for higher values of NN.

Threadpool consistently outperformed Rayon on both processors, achieving faster execution times and lower variability. This advantage can be attributed to Threadpool's lightweight scheduling, which is particularly effective for workloads with evenly distributed computational complexity. On the I7 11370H, the larger cache provided slightly better performance compared to the I5 10300H, particularly for larger block sizes, as the additional cache capacity reduced memory bottlenecks.

Overall, both processors achieved optimal performance using 8 threads and a block size of approximately 10% of NN. However, the I7 11370H showcased marginally better results across all configurations due to its superior cache size, demonstrating the importance of hardware capabilities in optimizing parallel performance.