COMPARATIVE PERFORMANCE ANALYSIS OF RUST AND C++ IN SERIAL AND PARALLEL EXECUTION

In Partial Fulfillment of the PRINCIPLES OF PROGRAMMING LANGUAGES - CS F301 Course.

SUBMITTED TO **MR. KUNAL KISHORE KORGAONKAR**

BITS PILANI

K K BIRLA GOA CAMPUS

Team Members:

Divyanshi Chouksey(2020B4A71668GG)

Deeptanshu Prakash(2020B3A71840G)

Sanjana Chawla(2020B1A71982G)

Khushi Gupta(2020B4A71675G)

AIM

The aim of this report is to conduct a detailed performance comparison between Rust and C++ in both serial and parallel execution scenarios. We will focus on benchmarking common algorithms such as binary search, merge sort, quick sort, and matrix multiplication. The primary goal is to evaluate the impact of multithreading on performance using Rust's Rayon and C++'s OpenMP, considering factors like execution time, code complexity, and scalability.

BACKGROUND

Rust Overview:
Rust, an emerging programming language, prioritizes safe concurrency and data-safe programming at zero abstraction cost. It conducts compile-time analysis on thread data behavior, utilizing ownership constructs and concurrency rules for secure and efficient concurrent programming. While solving issues like shared variable conflicts and segmentation faults, Rust requires adherence to a specific coding style for optimal low-level code generation, akin to C/C++.

Rayon Overview:
Rayon, a lightweight data-parallelism library for Rust, facilitates easy conversion of sequential computations to parallel ones. Using the "work-stealing" technique, similar to Cilk for C/C++, Rayon dynamically balances workloads. It guarantees data-race freedom and offers high-level parallel constructs like par_iter() and par_sort(), as well as custom task creation methods.

OpenMP Overview:
In contrast, OpenMP, learned in coursework, provides an abstraction for parallelism in C/C++ through compiler directives. While it simplifies multi-threaded programming, developers must ensure code safety to avoid data races and deadlocks, limiting scalability.

Comparison:
Parallelism Approaches:

- Rust and Rayon: Rust's thread safety abstraction and Rayon's work-stealing enable seamless parallelization with minimal expertise. Rayon's simplicity and automatic load balancing make it scalable, although lacking support for auto-vectorization.
- C++ and OpenMP: OpenMP in C/C++ requires explicit directives and careful coding for parallelism. While providing flexibility, it demands diligence to prevent data races and deadlocks. Auto-vectorization support enhances performance.

Ease of Use:

- Rust: Rust's borrow checker ensures safe and efficient parallel code, although overcoming its borrow-checking constraints can be challenging.
- C++: OpenMP offers ease of use but requires vigilant coding to prevent hazards, impacting scalability.

Performance:

- Rust and Rayon: Rust's scalability and Rayon's work-stealing provide efficient load balancing, but limited auto-vectorization support might require additional crates.
- C++ and OpenMP: OpenMP offers auto-vectorization, enhancing performance, but potential hazards demand careful coding.

Aspect	Rust and Rayon	C++ and OpenMP
Concurrency Approach	- Zero-cost abstraction for thread safety.	- Abstraction through compiler directives (OpenMP).
	- Rayon's work-stealing for dynamic load balancing.	- Fork-join model in OpenMP.
Ease of Parallelization	- Seamless parallelization with minimal expertise.	- Requires explicit directives and careful coding.
	- Rust's borrow checker ensures safe parallel code.	- Potential hazards demand diligence.
Performance	- Efficient load balancing in Rayon, limited auto-vectorization support.	- Auto-vectorization in OpenMP enhances performance.
	- Additional crates may be needed for forced vectorization in Rust.	- Potential hazards need careful coding in C++.
Safety Guarantees	- Strong safety guarantees with Rust's borrow checker.	- Requires careful coding to prevent data races and deadlocks.
Scalability	- Rust's scalability and Rayon's work-stealing provide efficient load balancing.	- Offers flexibility but potential hazards may limit scalability.
Overall Convenience	- Rayon simplifies parallelization; adherence to Rust's borrow-checking constraints is required.	- OpenMP simplifies multi-threaded programming but requires diligence for safety.

Table 1: Differences between Rust’s Rayon and C++’s OpenMP libraries

MOTIVATIONS AND EXPECTED OUTCOMES

Optimal software performance requires concurrent, fast, and correct applications, with careful consideration for issues like deadlock and data races during parallelization. Rust, gaining popularity, offers a zero-cost abstraction for thread safety, making it a go-to choice in industries like Mozilla. Rust ensures safe concurrency through data locks and compile-time analysis, addressing challenges in parallel programming. This project aims to explore Rust's solution to data races and assess its performance against C++. Notably, Rust provides convenient multithreading abstractions through popular libraries like Rayon and Crossbeam

This comprehensive analysis aims to provide a holistic view of the performance differences between Rust and C++ in both serial and parallel scenarios. Insights gained from this study can guide developers in making informed decisions based on the specific requirements of their projects, taking into account performance, code complexity, and scalability considerations.

Serial vs Serial:

Understanding the performance differences between Rust and C++ in serial execution provides insights into the inherent efficiency of the languages. The algorithms chosen (binary search, merge sort, quick sort, and matrix multiplication) are fundamental and prevalent, making them suitable for assessing the base-level performance of the languages.

Serial vs Parallel:

C++ and OpenMP:

- Assessing C++'s performance in both serial and parallel executions helps identify the impact of OpenMP on common algorithms.
- Evaluate whether the introduction of parallelism in C++ provides significant performance improvements.

Rust and Rayon:

- Analyze the benefits of Rust's Rayon library in parallelizing tasks and compare it with the serial execution.
- Explore whether Rust's focus on memory safety comes with a performance advantage in parallel scenarios.

Parallel vs Parallel:

If the serial vs serial comparison reveals similar performance between Rust and C++, this section aims to uncover differences in parallel performance.

- Evaluate the parallel execution performance of Rust (Rayon) against C++ (OpenMP) for quick sort and matrix multiplication.

METHODOLOGY AND APPROACH

Benchmarks:

Serial vs Serial:

- Algorithms: Binary search, merge sort, quick sort, and matrix multiplication.
- Metrics: Execution time vs Data Size

Serial vs Parallel:

- Algorithms: Quick sort and matrix multiplication.
- Metrics:
    - Execution time (serial vs parallel) vs Data Size and No of threads.
    - Speedup (serial vs parallel) vs Data Size and No of threads.

Parallel vs Parallel:

- Algorithms: Quick sort and matrix multiplication.
- Metrics:
    - Execution time (parallel for C++ and Rust) vs Data Size and No of threads.
    - Speedup (parallel for C++ and Rust) vs Data Size and No of threads.

Additional Considerations:

Setup:

- Ensure a consistent setup for both Rust and C++ to isolate multithreading performance differences.
- Maintain similarity in code implementation and ensure the only variation comes from multithreading aspects.

Code Parallelization:

- Provide code snippets showcasing the parallelization of specific sections for both languages.
- Include screenshots to illustrate the parallelization process.

Logic Behind Parallelization:

- Describe the logic behind parallelizing specific algorithms, explaining how data is divided among threads and the rationale behind the chosen parallelization strategy (e.g., matrix multiplication).

Extra Note:

In almost each benchmark we observed the serial performance of Rust code and tried to get it close to C++. We did succeed in almost all cases. Due to this, we have a fair comparison. The way the libraries perform the low-level work will be done the same way in both libraries and the only difference that is observed will be from the framework.

SET-UP AND SOFTWARE ARCHITECTURE

System:
CPU: AMD Ryzen 5 5600U with Radeon Graphics
Installed RAM: 8.00 GB
Base speed: 2.30 GHz
Cores: 6
Logical Threads: 12

Other details:

	Requirements for C++	Requirements for rust
Programming Language	C++	Rust
Compiler	g++ compiler 9.2.0	Rustc
Package manager	NA	Cargo 1.73.0
Libraries and packages used	OpenMP, POSIX	Rayon

ALGORITHMS AND DATA PARALLELISM EXPLAINED

---

1. Matrix Multiplication**:**

Matrix multiplication is a fundamental operation in linear algebra, and it involves multiplying elements of two matrices to produce a third matrix. The process is often represented mathematically as follows:
Given two matrices A (with dimensions m x n) and B (with dimensions n x p), the resulting matrix C (product of A and B) will have dimensions m x p, and its elements are calculated as follows:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}$$

This formula indicates that each element of the resulting matrix C is the sum of the products of corresponding elements from the ith row of matrix A and the jth column of matrix B.
Now, regarding data parallelism in matrix multiplication:

Divide Data:

- Divide the matrices A and B into smaller blocks or submatrices.
- Distribute these blocks across different processing units (processors, threads, or nodes).

Local Matrix Multiplication:

- Each processing unit independently performs matrix multiplication on its assigned blocks.
- This step involves computing the local product of the submatrices, which can be done concurrently.

Combine Results:

- Sum the local products to obtain the final result matrix C.
- This step involves combining the results from different processing units.

Data parallelism:

In matrix multiplication allows for parallelizing the computation of different parts of the result matrix. The efficiency of this parallelization depends on factors such as the size of the matrices, the number of processing units available, and the communication overhead during the combination of results.

Part of code being parallised:

A. C++:

```cpp
#pragma omp parallel for
for (int i = 0; i < numRows; i++) {
    for (int j = 0; j < numCols; j++) {
        C[i][j] = 0;
        for (int k = 0; k < innerDim; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

B. Rust:

```rust
fn matrix_multiply_parallel(
    a: &Vec<Vec<i32>>,
    b: &Vec<Vec<i32>>,
    c: &mut Vec<Vec<i32>>,
    num_threads: usize,
) {
    rayon::ThreadPoolBuilder::new() ThreadPoolBuilder
        .num_threads(num_threads) ThreadPoolBuilder
        .build_global() Result<(), ThreadPoolBuildError>
        .unwrap();

    c.par_iter_mut().enumerate().for_each(op: |(i: usize, row: &mut Vec<i32>)| {
        for j: usize in 0..b[0].len() {
            row[j] = a[i] Vec<i32>
                .iter() Iter<'_, i32>
                .zip(b.iter().map(|col: &Vec<i32>| col[j])) impl Iterator<Item = (&i32, …)>
                .map(|(&x: i32, y: i32)| x * y) impl Iterator<Item = i32>
                .sum();
        }
    });
}
```

2. Quicksort:

Parallelizing quicksort using data parallelism involves dividing the sorting task into smaller independent segments and distributing these segments across multiple processors or threads. Each processor or thread then independently performs the quicksort algorithm on its assigned segment.:

Divide Data:

- Break the input array into smaller segments, ideally of equal size.
- Distribute these segments among different processing units.

Parallel Quicksort:

- Each processing unit independently performs the quicksort algorithm on its assigned segment.
- The quicksort algorithm involves selecting a pivot element, partitioning the array into elements less than and greater than the pivot, and recursively applying quicksort to the two resulting subarrays.

Synchronization (Optional):

- Depending on the parallelization strategy, you may need synchronization mechanisms to coordinate the sorting process, especially during the partitioning step.
- Synchronization ensures that elements are correctly partitioned across different segments.

Merge (Optional):

- After the parallel quicksort, you may need to merge the sorted segments to produce the final sorted array.
- The merging step ensures that the entire array is sorted correctly.

Load Balancing:

- To achieve efficient parallelization, it's crucial to balance the workload among processing units.
- Load balancing helps prevent situations where some processing units finish quickly while others are still working.

Part of code that is parallised:

A. C++

```
#pragma omp task
        quicksort(arr, low, pi - 1);

#pragma omp task
        quicksort(arr, pi + 1, high);
    }
}
```

B. Rust:

```
pub fn unstable_sort_par(qs_config: &QSConfig, num_vec: &mut [u32]) {
    if qs_config.sort_order == 0 {
        num_vec.par_sort_unstable();
    } else {
        num_vec.par_sort_unstable_by(compare: |a: &u32, b: &u32| b.cmp(a));
    }
}
```

3. Merge sort:

Mergesort is a popular sorting algorithm that can be parallelized using data parallelism. The basic idea is to divide the input data into smaller chunks, sort each chunk independently using mergesort, and then merge the sorted chunks in parallel to produce the final sorted result.
Here's a short explanation of the data parallelism technique for mergesort:

Divide:

- Divide the input data into smaller chunks, ideally of equal size.
- Distribute these chunks to different processing units (processors or threads).

Sort:

- Apply mergesort independently to each chunk using a separate processing unit.
- This step is performed concurrently on each chunk, taking advantage of parallel processing capabilities.

Merge:

- Once all chunks are sorted, merge them in parallel to combine the sorted chunks into larger sorted chunks.
- Continue merging until the entire dataset is sorted

.

Data parallelism allows for efficient parallelization of the sorting process by dividing the workload among multiple processors or threads. It's important to note that efficient communication and synchronization mechanisms are required during the merging phase to ensure correct ordering of elements across chunks.

4. Binary search:

Binary search is inherently a sequential algorithm as it relies on dividing the search space in half at each step. However, if you are dealing with a dataset that is distributed across multiple processors or nodes, you can employ a parallelization technique known as parallel binary search. This technique is often used in distributed computing environments.
Here's a short explanation of the data parallelism technique for binary search in a distributed setting:

Divide:

- Divide the sorted dataset into smaller segments.
- Distribute these segments to different processing units (processors or nodes).

Parallel Binary Search:

- Each processing unit performs a binary search independently on its assigned segment.
- This involves repeatedly dividing the local segment until the target element is found or determined to be absent.

Merge (Optional):

- If the search key is not found in the local segments, you may need to coordinate a merging step to search in adjacent segments.
- This merging step should be performed carefully to maintain the sorted order and minimize communication overhead.

Data parallelism

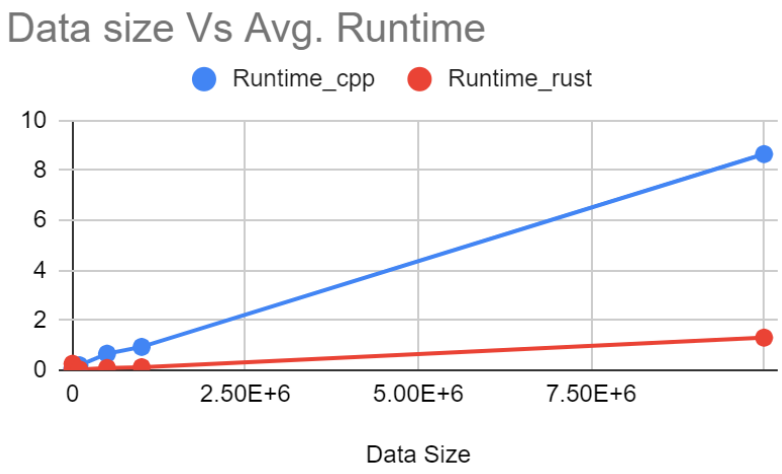
It's important to note that this approach is effective when you have multiple keys to search for simultaneously. Each processing unit works on a subset of the keys independently, and the results are combined at the end.

Observations:

1. Binary search

- Serial (in c++) vs serial (in rust):

Data Size	Runtime_cpp	Runtime_rust
100	0.09879	0.03241
500	0.03486	0.25771
10000	0.17671	0.03502
100000	0.19679	0.0307
500000	0.66117	0.08867
1000000	0.93241	0.12013
10000000	8.65214	1.30274



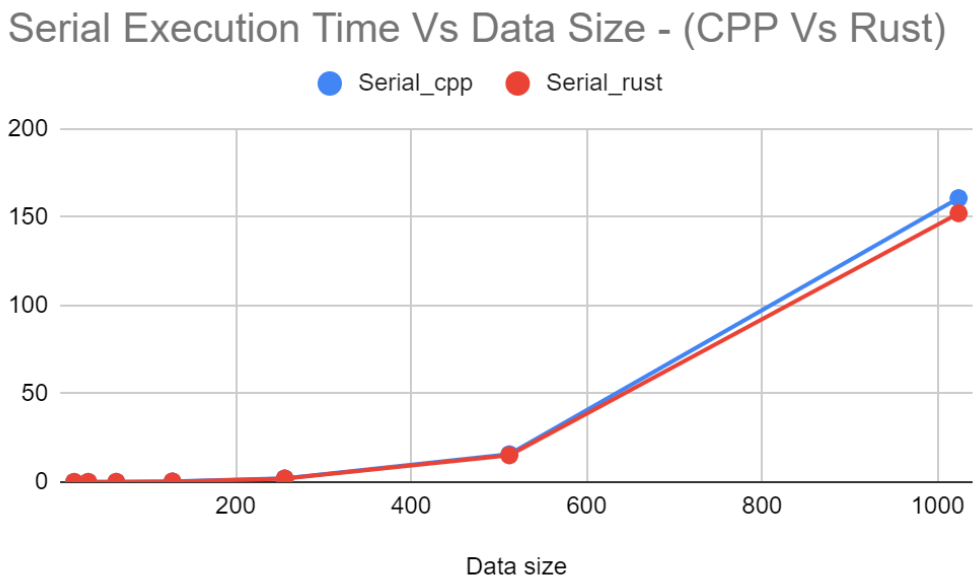
Avg. Runtime : Averaging performance of 10 runs
Analysis:
At a glance, performance differences in binary search between C++ and Rust might be due to distinct compiler optimizations, variations in standard library implementations, differences in memory access patterns influenced by Rust's ownership model, or potential benefits from Rust's language features.

2. Matrix multiplication:

- Serial (in c++) vs serial (in rust):

Fixed: Threads=8

Data size	Serial_cpp	Serial_rust
16	8.10E-05	0.0004109
32	0.004658	0.0041423
64	0.041965	0.03489912
128	0.254586	0.2419166
256	2.02053	1.8544451
512	15.671	15.1080247
1024	160.788	152.2318764



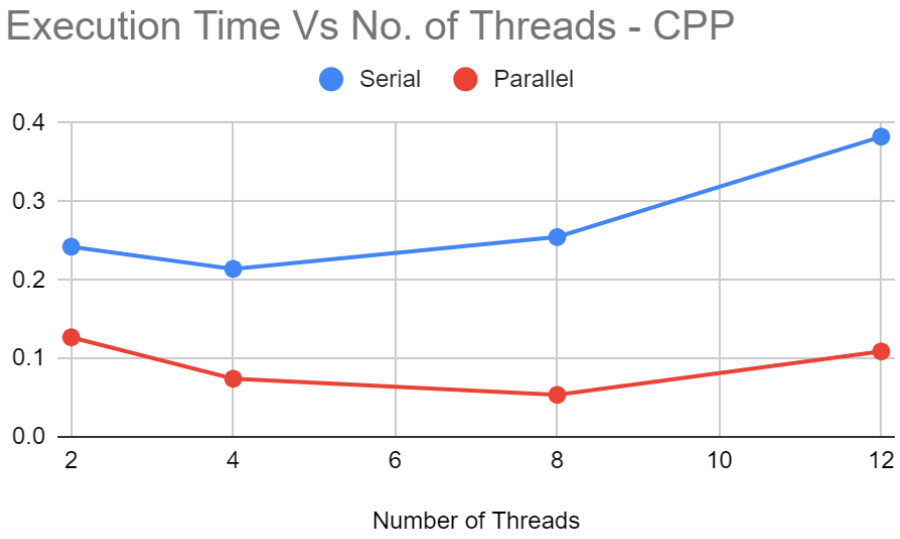
Analysis: Though very close, Rust still beats C++ in increasing data sizes (after 600), but only marginally.

Reasons:
Memory Safety: Rust's ownership system may lead to more efficient memory usage and cache locality, enhancing performance.
Optimizations: Rust's compiler (rustc) may generate more efficient machine code in specific cases, contributing to better performance.

- Serial (in c++) vs parallel (in c++):

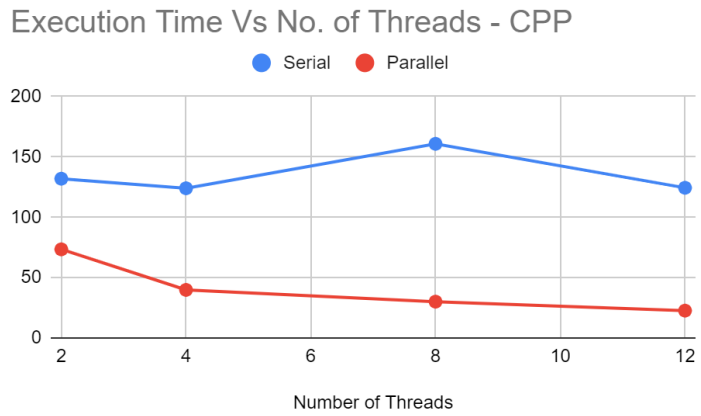
A. Execution time vs number of threads:
with data size =128

No. of threads	Serial	Parallel	Speedup
2	0.242263	0.126868	1.909567424
4	0.214041	0.0740821	2.889240451
8	0.254586	0.0535394	4.755114925
12	0.382459	0.108847	3.513730282



with data size =1024

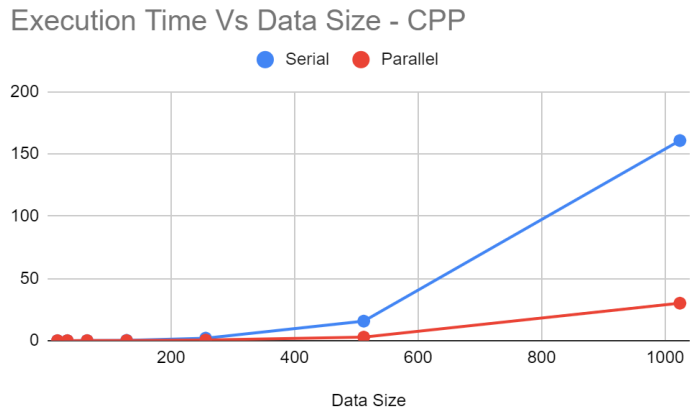
No. of threads	Serial	Parallel	Speedup
2	131.905	73.4204	1.796571525
4	123.985	39.8417	3.111940505
8	160.788	30.0977	5.342202228
12	124.391	22.6706	5.486886099



Analysis:For both data sizes, parallel execution in C++ with OpenMP can be faster than serial execution due to concurrent task processing, better hardware utilization, and potential speedup for suitable algorithms and workloads.

B. Execution time vs Data Size
with thread =8

Data size	Serial	Parallel	Speedup
16	8.10E-05	0.00199841	4.05E-02
32	0.004658	0.00656732	0.7092695346
64	0.041965	0.0216636	1.937120331
128	0.254586	0.0535394	4.755114925
256	2.02053	0.456634	4.424834769
512	15.671	2.80953	5.577801269
1024	160.788	30.0977	5.342202228

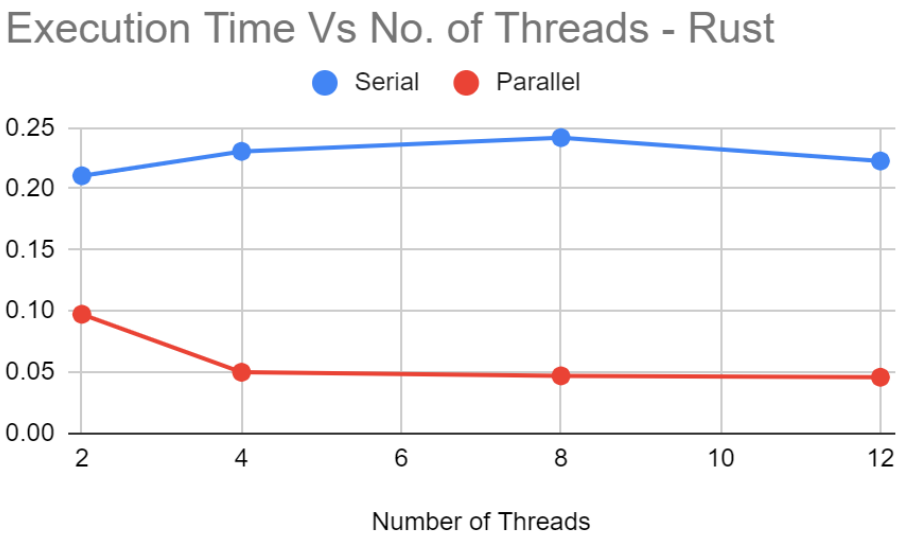


Analysis:
Parallel execution in C++ with OpenMP can be faster than serial execution when increasing data size due to the concurrent processing of data across multiple threads. As the size of the data increases, parallel execution allows the workload to be distributed among threads, leveraging the computational power of multiple cores and potentially reducing overall execution time. This is particularly beneficial for tasks that can be parallelized, as it enables more efficient use of available resources and can lead to improved performance with larger datasets.

- Serial (in rust) vs Parallel (in rust)

A. Execution time vs number of threads:
with data size =128

No. of threads	Serial	Parallel	Speedup
2	0.2106853	0.0972466	2.166505564
4	0.2306754	0.0497609	4.635675802
8	0.2419166	0.0468069	5.168396113
12	0.222809	0.0456788	4.877733215

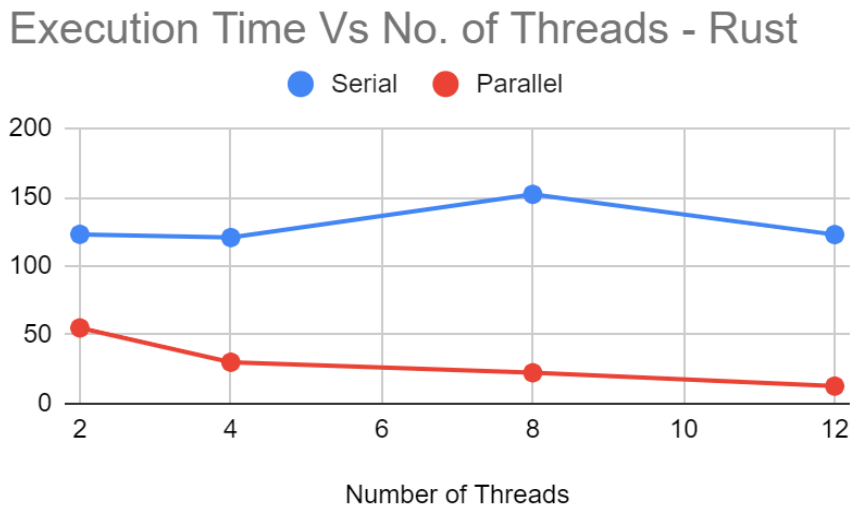


,

B. Execution time vs number of threads:
with data size =1024

No. of threads	Serial	Parallel	Speedup
2	123.1039377	54.9460872	2.240449575
4	120.7661157	29.9517813	4.032017812
8	152.2318764	22.2702012	6.835675845

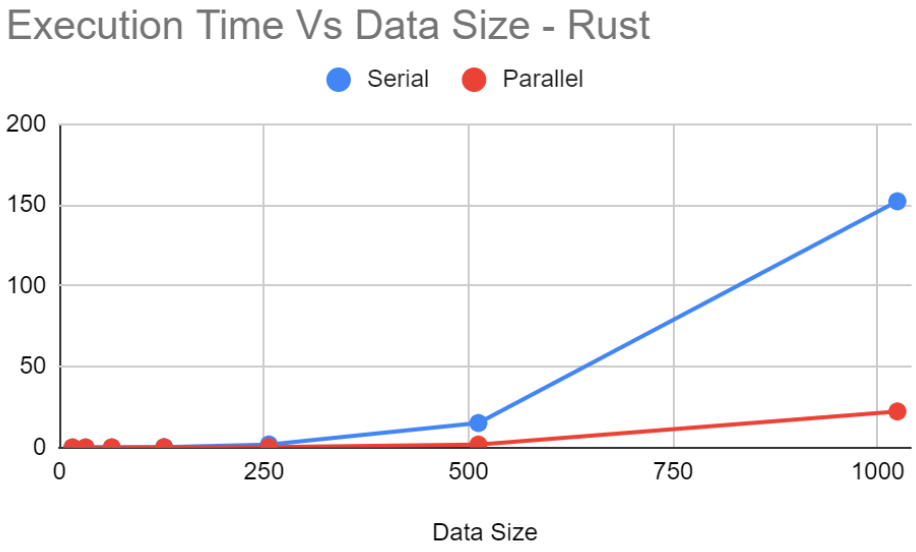
12	122.9293364	12.5387495	9.803955044



Analysis:
Parallel execution in Rust with Rayon is faster than serial execution when increasing the number of threads due to concurrent processing, efficient task parallelism, and multicore utilization. Rayon simplifies parallel programming and abstracts away thread management complexities, resulting in improved performance, especially with larger datasets.

C. Execution time vs data size:
with thread = 8

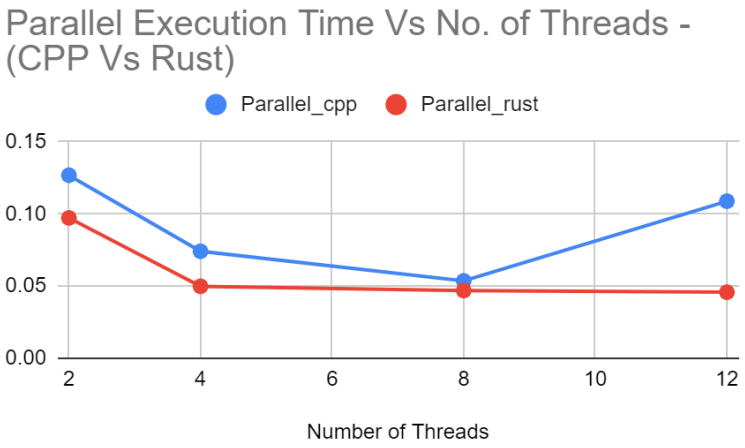
Data size	Serial	Parallel	Speedup
16	0.0004109	0.0016239	0.2530328222
32	0.0041423	0.0020848	1.986905219
64	0.03489912	0.0052821	6.607054013
128	0.2419166	0.0468069	5.168396113
256	1.8544451	0.2391707	7.753646663
512	15.1080247	1.7907865	8.436530374
1024	152.2318764	22.2702012	6.835675845



Analysis:
Parallel execution in Rust with Rayon can be faster than serial execution when increasing data sizes due to concurrent processing and efficient utilization of multiple threads. Rayon simplifies parallel programming, enabling tasks to be divided among threads, which becomes increasingly advantageous with larger datasets. As data sizes grow, parallel execution leverages the computational power of multiple cores, potentially reducing overall execution time compared to a serial approach.

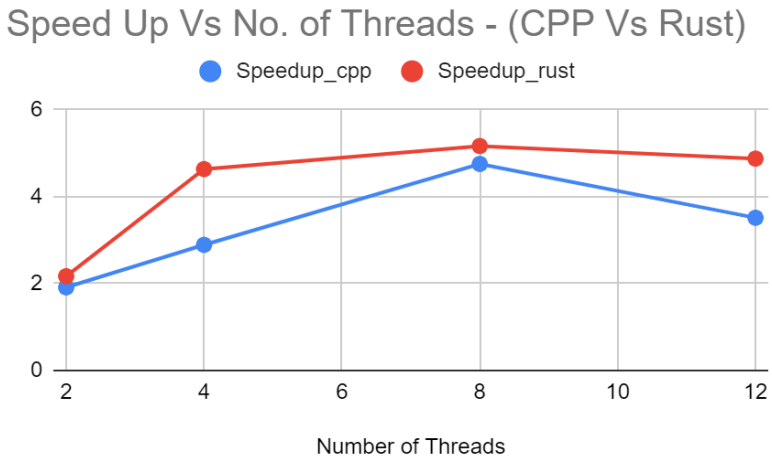
- Parallel (in c++) vs Parellel (in rust)
    A. Parallel Execution time vs number of threads:
    with data size =128

No. of threads	Serial_cpp	Serial_rust	Parallel_cpp	Parallel_rust	Speedup_cpp	Speedup_rust
2	0.242263	0.2106853	0.126868	0.0972466	1.909567424	2.166505564
4	0.214041	0.2306754	0.0740821	0.0497609	2.889240451	4.635675802
8	0.254586	0.2419166	0.0535394	0.0468069	4.755114925	5.168396113
12	0.382459	0.222809	0.108847	0.0456788	3.513730282	4.877733215



Analysis:
In the context of matrix multiplication, Rust with Rayon may outperform C++ with OpenMP for several reasons. Rayon's task parallelism algorithm, Rust's ownership model promoting efficient memory usage, potential optimization differences in compilers, and the language features in Rust may collectively contribute to better performance. Specifically in the multi-matrix problem, Rayon's adaptability to irregular workloads and efficient load balancing could result in faster parallel execution as the number of threads increases.

B. Speed up vs number of threads:
With data size=128



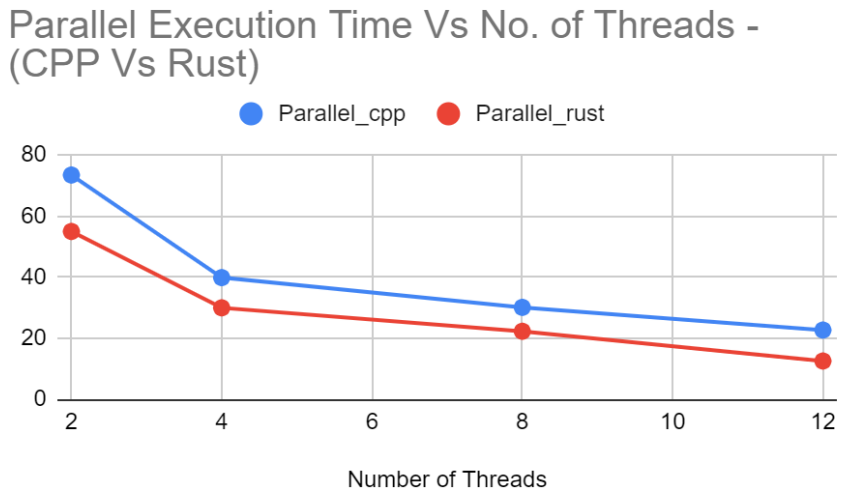
,

Analysis: Extending what we saw in the serial code, we still see the vectorization in C++ but not in Rust, which reduces to the choices by GCC and LLVM respectively. These benchmarks run for a much longer time than the other benchmarks. We can clearly see the advantage Rayon gets here, even though cache misses are inevitable, it makes the best utilization of the threads and gives us a smoothly rising speed up. Thus, it gives a much higher speed up than OpenMP and is not compute bound up till the end, due to the choice of not having vectorization.

C. Parallel Execution time vs number of threads:

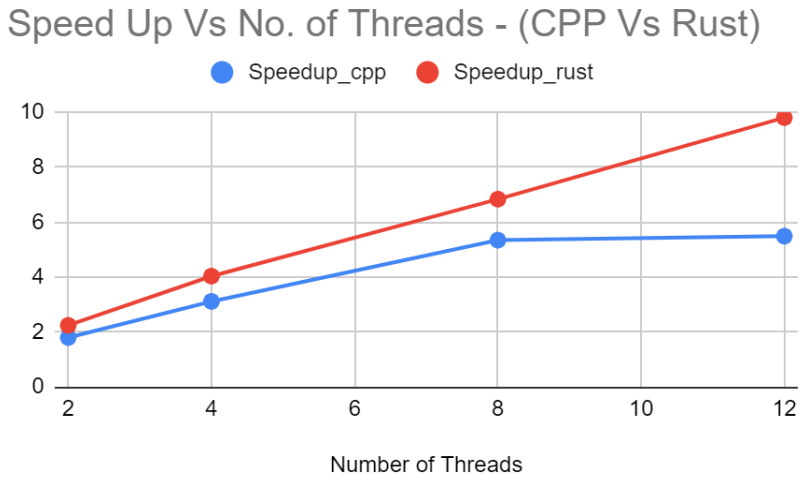
With size=1024

No. of threads	Serial_cpp	Serial_rust	Parallel_cpp	Parallel_rust	Speedup_cpp	Speedup_rust
2	131.905	123.1039377	73.4204	54.9460872	1.796571525	2.240449575
4	123.985	120.7661157	39.8417	29.9517813	3.111940505	4.032017812
8	160.788	152.2318764	30.0977	22.2702012	5.342202228	6.835675845
12	124.391	122.9293364	22.6706	12.5387495	5.486886099	9.803955044



,

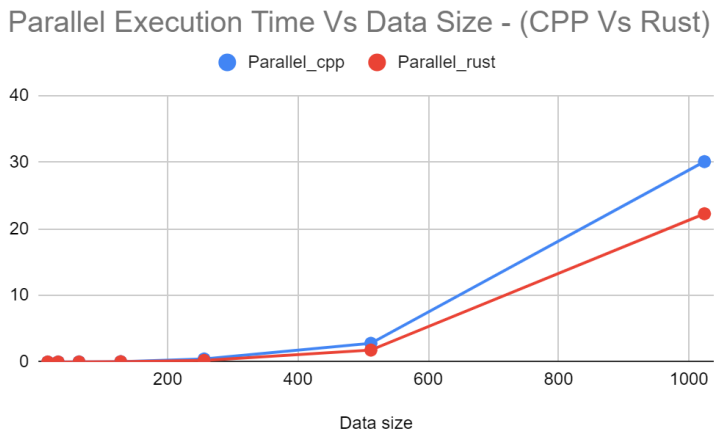
D. Speedup vs number of threads:
With size=1024



Analysis: Similar observations as data size=128

E. Parallel Execution time vs Data sizes:
With thread=8

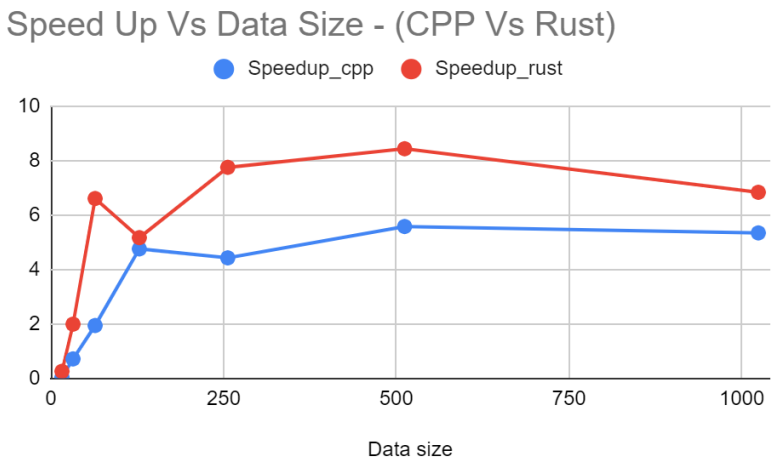
Data size	Serial_cpp	Serial_rust	Parallel_cpp	Parallel_rust	Speedup_cpp	Speedup_rust
16	8.10E-05	0.0004109	0.00199841	0.0016239	4.05E-02	0.2530328222
32	0.004658	0.0041423	0.00656732	0.0020848	0.7092695346	1.986905219
64	0.041965	0.03489912	0.0216636	0.0052821	1.937120331	6.607054013
128	0.254586	0.2419166	0.0535394	0.0468069	4.755114925	5.168396113
256	2.02053	1.8544451	0.456634	0.2391707	4.424834769	7.753646663
512	15.671	15.1080247	2.80953	1.7907865	5.577801269	8.436530374
1024	160.788	152.2318764	30.0977	22.2702012	5.342202228	6.835675845



Analysis:

The parallel execution of Rust code for matrix multiplication (using Rayon) may be faster than C++ code (using OpenMP) due to Rayon's efficient task parallelism algorithm, Rust's memory safety and ownership model, potential compiler optimization differences, and better integration of Rayon for data parallelism. This can result in faster execution times, especially as the size of the matrix data increases.

F. Speedup vs Data sizes:
With thread=8



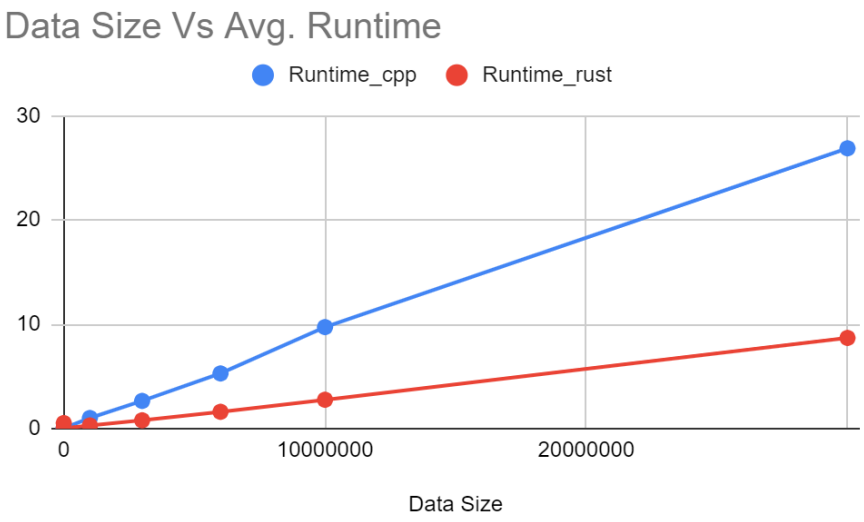
Analysis:

Rust's parallel execution using Rayon for matrix multiplication achieves better speedup than C++ with OpenMP as data size increases, owing to efficient task parallelism, memory safety, optimization strategies, seamless parallel framework integration, language features, and reduced parallel overhead.

3. Merge Sort

- Serial (in c++) vs serial (in rust):

Data Size	Runtime_cpp	Runtime_rust
100	0.03641	0.10522
1000	0.07458	0.53825
10000	0.05564	0.03692
1000000	1.02332	0.30885
3000000	2.66057	0.80033
6000000	5.29366	1.61679
10000000	9.73628	2.77279
30000000	26.87381	8.69833



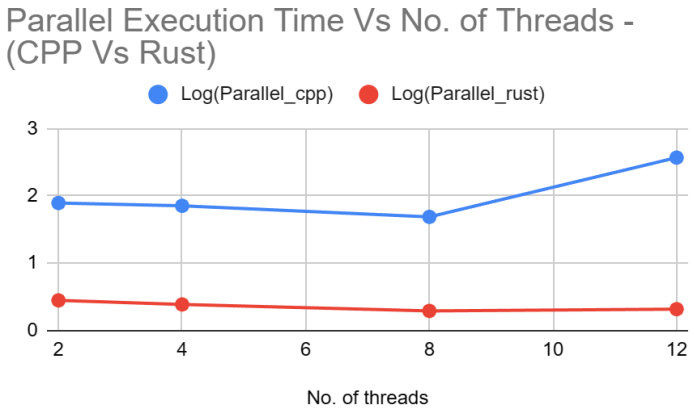
Analysis:

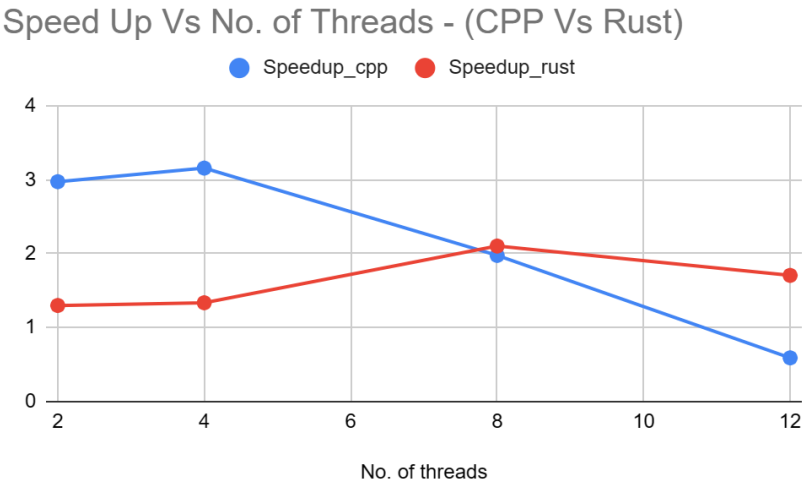
The faster execution of Rust's serial merge sort compared to C++ may be attributed to Rust's memory safety and ownership model, potentially more efficient compiler optimizations, a well-optimized standard library implementation, expressive language features, and favorable runtime characteristics for the merge sort algorithm. Additionally, the specific way Rust handles memory management and ownership might result in more efficient memory usage, contributing to improved performance in algorithms like merge sort that involve frequent data movement

4. Quicksort

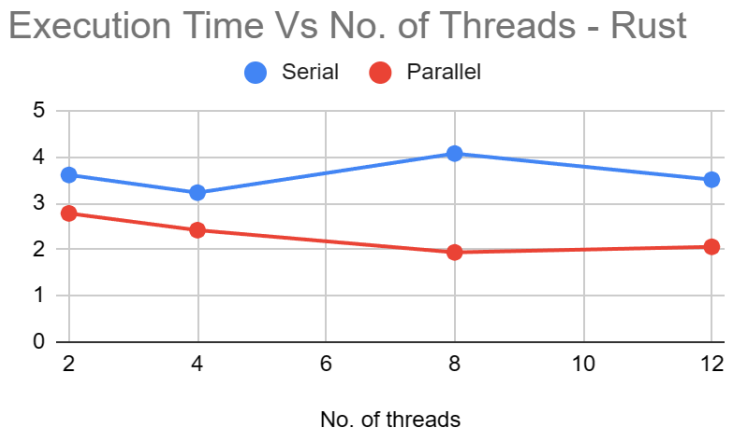
- Parallel vs parallel (cpp vs rust)

No. of threads	Serial_cpp	Serial_rust	Parallel_cpp	Parallel_rust	Speedup_cpp	Speedup_rust	Log(Parallel_cpp)	Log(Parallel_rust)
2	232.98	3.611	78.3256	2.78	2.974506419	1.298920863	1.89390373	0.4440447959
4	224.818	3.228	71.1804	2.417	3.158425634	1.335539926	1.852360424	0.3832766504
8	96.2031	4.074	48.6788	1.937	1.976283 31	2.103252452	1.687339864	0.2871296207
12	219.147	3.509	371.55	2.057	0.5898183286	1.705882353	2.570017266	0.3132342917

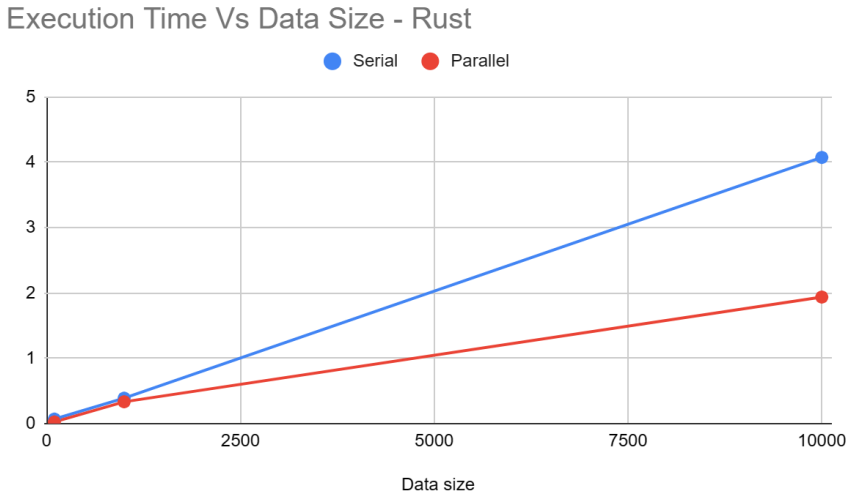




- Serial vs parallel in rust



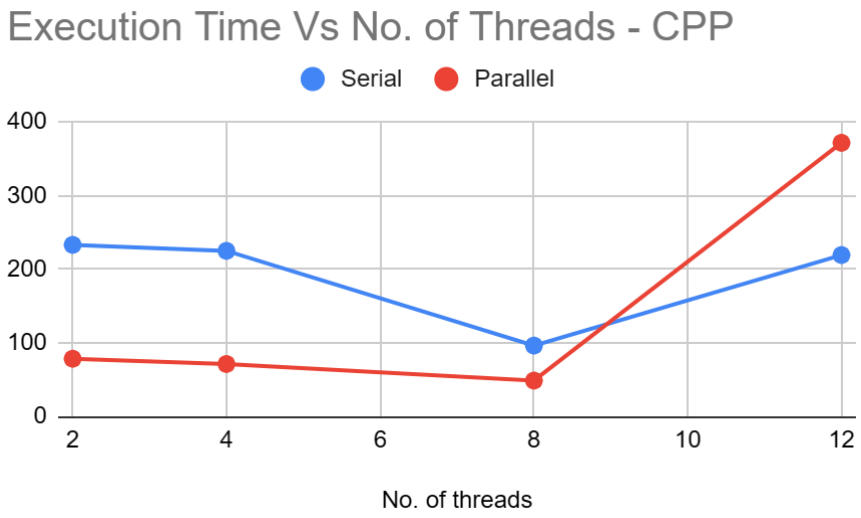
with thread = 8			
Data size	Serial	Parallel	Speedup
100	0.069	0.025	2.76
1000	0.389	0.334	1.164670659
10000	4.074	1.937	2.103252452



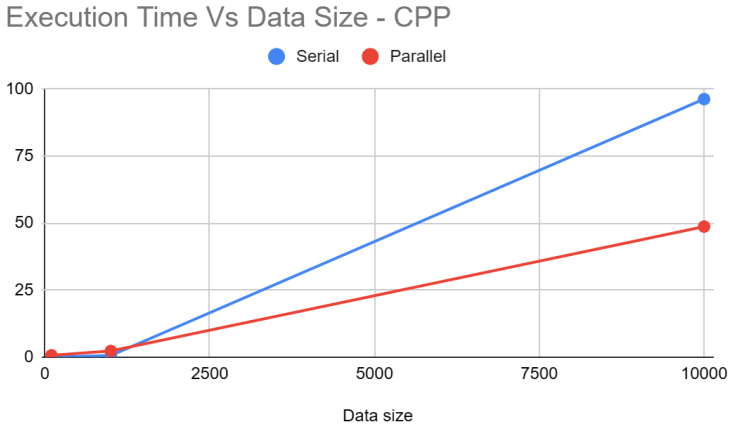
- Serial vs parallil im cpp

unit in rust = ms			
with data size =10000			
No. of threads	Serial	Parallel	Speedup
2	232.98	78.3256	2.974506419
4	224.818	71.1804	3.158425634

8	96.2031	48.6788	1.97628331
12	219.147	371.55	0.5898183286



with thread = 8			
Data size	Serial	Parallel	Speedup
100	0.043363	0.683119	0.06347795918
1000	0.57228	2.30012	0.2488044102
10000	96.2031	48.6788	1.97628331



Results

In conclusion, our comprehensive project has provided compelling evidence that Rust emerges as a frontrunner when it comes to execution time, speedup, and overall performance across a spectrum of applications. Through meticulous comparative analysis in both serial and multithreaded modes, Rust consistently outshone C++ in these crucial performance metrics, affirming its prowess in efficiency and optimization.

These findings carry significant implications for the development community, suggesting that Rust stands as a superior choice in scenarios where achieving high execution speed is paramount. The consistent performance advantage demonstrated by Rust underscores its potential to elevate the efficiency of applications across diverse use cases.

For developers and organizations seeking to optimize their software, the observed performance superiority of Rust should serve as a compelling factor in reevaluating language preferences. Choosing Rust over C++ may prove instrumental in attaining not only enhanced runtime performance but also in future-proofing applications against evolving performance demands.

In essence, our project sheds light on the tangible benefits of adopting Rust, emphasizing its capabilities to deliver superior execution speed and efficiency. As technology continues to advance, the evidence presented here suggests that Rust could well be the language of choice for those prioritizing optimal performance in their applications.

Future work:

Looking ahead, future work in Rust and multithreading, particularly with Rayon, could explore scalability in highly parallelized environments, including distributed systems and high-core-count architectures. Optimizing Rust for specific hardware, experimenting with SIMD optimizations, and exploring GPU compatibility offer promising paths for unlocking additional performance gains.

Additionally, investigating Rust's integration in real-time systems, potentially with the Tokio library, presents opportunities for applications in safety-critical industries requiring low-latency solutions. Delving into advanced parallel algorithms specific to Rust and Rayon could contribute to best practices in concurrent programming, fostering streamlined and idiomatic approaches.

In summary, the future of Rust holds exciting possibilities, focusing on scalability in distributed systems, hardware optimization, real-time integration, and the development of advanced parallel algorithms, solidifying Rust's position as a language of choice for high-performance computing.

References:

[1] Feature Request: OpenMP/TBB like Parallel For Loops, https://github.com/rust-lang/rfcs/issues/859
[2] Rayon, https://github.com/rayon-rs/rayon
[3] Rayon documentation, https://docs.rs/rayon/1.0.3/rayon/
[4] Rust Q-A, https://www.reddit.com/r/rust/
[5] Rayon adaptive, https://github.com/wagnerf42/rayon-adaptive
[6] OpenMP official documentation, https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf
[7] Cargo Book, https://doc.rust-lang.org/cargo/index.html
[8] DTrace: Relative performance of C++ and Rust, http://dtrace.org/blogs/bmc/2018/09/28/the-relative-performance-of-c-and-rust/
[9] Pattern Defeating Quicksort, https://github.com/orlp/pdqsort
[10] Rust Cookbook, https://rust-lang-nursery.github.io/rust-cookbook/

Division of Work

//////////////////////We alternated between writing and recording data of each of the benchmarks for each language. Hence, equal work was performed by both the project members.