Comparing OpenMP and Rust

Aryan Shetty, Kiran Dhanasekaran
New York University
{as17388, kd3059}@nyu.edu

Abstract

From supercomputers to smartphones, multi-core processing now dominates the computing landscape. Tackling future computational challenges will increasingly depend on the ability of programmers, engineers, and scientists to effectively harness the power of these multi-core systems. A key part of this effort is choosing the right platform for implementing parallelism. In this context, the programming language selected for a given task plays a crucial role in determining success.

Currently, OpenMP stands as the industry standard—a mature and well-established set of directives available through C/C++. Meanwhile, Rust is emerging as a strong contender, offering Rayon, its own library for parallelization. Comparing how OpenMP and Rust perform in multi-core programming provides valuable insight for anyone participating in the ongoing evolution of computing.

This study evaluates both OpenMP and Rust using six parallel programming benchmarks and a set of comparison criteria. Interestingly, while overall performance between the two languages was similar, Rust's concise syntax, high-level abstractions, and built-in safety mechanisms limited fine-grained control in certain cases where it was necessary. These findings highlight that the degree of control a language gives the programmer may be a more critical factor in language selection than previously assumed.

1 Introduction

As multi-core processors become increasingly widespread and computing continues to integrate into all aspects of society, the ability to effectively utilize parallel computation grows ever more crucial. In this evolving landscape, OpenMP stands as the de facto standard for parallel programming as of 2024. Built as a collection of API directives layered onto C, C++, and Fortran, OpenMP has earned its place through the popularity of its host languages and its own maturity, developed over more than two decades. Given this strong foundation, it might seem unnecessary to explore alternatives. Yet, to stay at the forefront of technological progress, new languages and methods must continually be examined.

One such emerging alternative is Rust. Over the past decade, Rust has gained significant traction, offering the performance, succinctness, and lack of garbage collection found in C, while also providing the memory safety typically associated with garbage-collected languages like Java and Python. As a result, Rust is increasingly being adopted for core systems in a variety of software organizations. This makes it worthwhile to compare and contrast Rust and OpenMP to better understand their respective advantages, limitations, and suitability for different contexts.

At its foundation, OpenMP is a set of compiler directives and runtime library routines designed to facilitate parallel programming. Its primary parallelization strategy revolves around the "for loop" construct, which implements a fork/join model: a parent thread spawns multiple child threads in the parallel region, which are later synchronized back into a single thread for sequential execution. OpenMP also includes a wide array of tools to help manage shared memory safely—offering features such as atomics, locks, and critical sections to prevent race conditions, false sharing, and maintain cache coherence. In nearly all cases, OpenMP gives the programmer full control over the parallel execution.

One of the most fundamental differences between OpenMP and Rust lies in their nature: OpenMP is a library of routines, while Rust is a full-fledged programming language. This distinction sets the stage for differing approaches to parallelism. In Rust, there are two primary methods for achieving parallel execution. The first involves manual thread management using the 'thread::spawn' function. While this method grants the programmer a high degree of control, it also introduces significant complexity.

For the purposes of this study, the focus is placed on Rayon—a popular parallelism crate (Rust's term for a library) that is widely adopted in the Rust community. As of this writing, Rayon's GitHub repository boasts over 11,000 stars, reflecting its broad usage and support. Rayon offers two core mechanisms for parallel programming: parallel iterators and custom tasks. Parallel iterators are designed to work with data structures that support iteration, such as arrays and vectors. However, using them effectively requires a strong grasp of one of Rust's central concepts: ownership.

Rust's ownership model, which ensures that each value has a single owner at any given time, is key to its memory safety guarantees. In the context of parallelism, this model introduces additional complexity, particularly when managing data shared across threads. This study will delve into how these complexities impact real-world use of Rayon.

The central question guiding this comparison is: How does OpenMP stack up against Rayon as a tool for parallel programming? Under what circumstances should one be chosen over the other? How do factors such as performance and scalability vary across different computing environments? To explore these questions, six benchmark programs have been implemented using both C/C++ with OpenMP and Rust with Rayon. These benchmarks are evaluated across a variety of metrics, and based on the results, the study will offer a set of recommendations for developers considering either option for parallel computing tasks.

2 Motivation

Throughout the Multicore Programming course (CSCI-GA.3033-015), we've come to understand that achieving scalable software performance on modern hardware requires us, as developers, to build concurrent, high-performance, and correct applications. When breaking down algorithms into parallel workloads, we must also account for a new class of potential errors—such as deadlocks, livelocks, and data races. To truly benefit from parallelism, it's essential to recognize and mitigate these issues proactively.

Rust aims to address these challenges by offering zero-cost abstractions for thread safety, which has contributed to its growing popularity among developers and within the industry—most notably with Mozilla, one of its primary early advocates. Rust facilitates safe concurrency through data locking mechanisms and message-passing channels. Moreover, it performs compile-time analysis on thread behavior to catch potential issues early in the development process.

At the heart of Rust's safety model is its unique ownership system and strict concurrency rules, which together form a powerful compile-time toolkit. These features help developers write concurrent code that is both efficient and free from common threading bugs like data races. In this project, we aim to explore how effectively Rust addresses the problem of data races and how its performance compares to that of C++. Rust promises to eliminate many of the parallelization pitfalls faced in C++—and to do so without runtime cost.

To carry out this investigation, we'll be working with two widely recommended Rust libraries for parallelism—Rayon and Crossbeam. Both are well-regarded within the Rust community and offer high-level abstractions for multithreaded programming. Through this study, we hope to gain deeper insights into Rust's concurrency model and evaluate whether it lives up to its promises in real-world performance and safety.

3 Related Works

While there is a wealth of literature available on both Rust and OpenMP individually, direct comparisons between the two remain surprisingly scarce. The existing body of research generally falls into three primary categories: high-performance computing using Rust, comparative studies of Rust and C, and in-depth analyses of OpenMP. Each of these areas is explored in more detail below.

3.1 High Performance Computing with Rust

As a modern systems programming language built with both safety and performance in mind, Rust presents itself as a strong alternative to traditional High Performance Computing (HPC) languages like Fortran and C++ [2]. Within this context, Rust has been widely explored for its capabilities in numerical computing, including support through libraries for complex numbers, linear algebra, and matrix operations [3]. Parallelism—particularly through the Rayon crate—has also been examined in this domain, along with Rust's growing support for GPU programming [3].

However, while HPC is closely tied to parallel programming, much of the existing research in this space focuses on scientific and mathematical applications that often involve specialized requirements not directly applicable to general-purpose parallelism. Moreover, despite Rust's increasing presence in HPC, direct comparisons between Rust (and Rayon) and OpenMP remain largely unexplored in current literature.

3.2 Rust vs C

Rust has frequently been compared to other programming languages, particularly its most widely used high-performance counterpart—C. These comparisons typically emphasize the fundamental differences in how the two languages manage ownership and memory. Rust is well known for its ability to prevent memory mismanagement issues, such as data races and invalid memory access [6]. This is largely due to its unique ownership model, where each value in a program has exactly one owner at any given time. The

Rust compiler enforces this model through compile-time checks, ensuring memory safety and preventing improper access [6].

In contrast, C lacks any built-in ownership model and places the burden of memory management squarely on the programmer. This can easily lead to the kinds of memory errors that have plagued countless software projects over the years. While the existing literature thoroughly explores the differences between Rust and C, including their respective approaches to concurrency, what remains missing is a focused, in-depth comparison between Rust and OpenMP. Most studies have concentrated on language-level differences rather than evaluating the distinct parallel programming libraries that each ecosystem offers.

3.3 OpenMP In-Depth

Although not often studied in direct comparison with Rust, OpenMP has been extensively explored on its own within the research community. With over two decades of development and a robust ecosystem, OpenMP is supported by a vibrant and engaged network of developers and researchers. It is widely recognized as the "industry standard" for shared memory parallelism [1], and its suite of directives, APIs, and thread models has been thoroughly benchmarked and analyzed across numerous applications.

In particular, OpenMP has seen significant use in fields like numerical analysis and applied mathematics, where it has been employed to simulate and study complex scientific phenomena with considerable success. Thanks to its flexibility and power, OpenMP has become a preferred tool among researchers working on diverse problems—from elasto-viscoplastic Fast Fourier transform-based micromechanical solvers to multilevel Green's function interpolation methods [7,8]. While these studies require deep domain knowledge in addition to OpenMP proficiency, they demonstrate the broad applicability and strength of the framework.

The ongoing effort to parallelize scientific workloads in response to increasingly powerful hardware continues to yield meaningful results, and this body of research served as a key source of inspiration for the benchmark programs selected in this study.

4 Proposed Idea

The primary objective of this study is to evaluate and compare the performance and usability of two parallel programming models: OpenMP—an established standard for parallel programming in C and C++—and Rayon, a modern Rust library built on principles of safety and concurrency. This comparison is carried out through the development of a diverse set of benchmarks designed to reflect a wide range of computational scenarios, enabling a thorough assessment of each language and parallelization strategy.

The benchmark suite consists of six distinct programs, each representing different computational and memory access patterns commonly found in scientific computing, bioinformatics, and general-purpose parallel workloads. By including both computation-bound and memory-bound tasks, the suite aims to reveal how each approach performs under realistic and varied conditions.

This broad scope enables the analysis of raw performance metrics—such as speedup and scalability—while also considering language-specific characteristics like memory management, thread control, and compilation time. Additionally, the study examines qualitative aspects of developer experience, including code complexity, ease of introducing parallelism, and the degree of control each model affords.

4.1 Categories for Comparison

The benchmark suite evaluates performance and usability across the following key categories:

1. **Programmability** – Assessed by examining code complexity and lines of code, this metric evaluates how easy or difficult it is for a developer to implement each benchmark in both languages.
2. **Scalability** – Measured by analyzing how efficiently each parallel model handles an increasing number of threads while keeping the problem size constant, offering insights into how well each approach scales.
3. **Performance** – Evaluated using speedup, defined as the ratio of single-thread runtime to multi-thread runtime, to determine how effectively each implementation leverages parallelism.
4. **Runtime Overhead** – Includes the time spent on thread creation, synchronization, and termination. This is calculated by subtracting the workload execution time from the total runtime of the program.
5. **Compilation Speed** – Measured using the `time` command to record the duration required to compile each benchmark program.
6. **Programmer Control** – Evaluated qualitatively to compare how much low-level control each language provides, particularly in areas such as thread-to-core assignment and the management of shared versus private variables.

5 Experimental Setup

5.1 Benchmarks

The following four benchmark programs were selected to represent a wide variety of computational and memory access patterns, focusing on both recursive sorting algorithms and numerical simulations. This diversity enables a holistic comparison between OpenMP and Rust's Rayon library across several key performance metrics:

- **Merge Sort**: Implements a parallelized recursive merge sort. This algorithm involves frequent memory accesses and moderate use of additional memory buffers. It exhibits high computational demands due to recursive calls, and the merging step introduces synchronization and load balancing considerations.
- **Quick Sort**: A classic in-place recursive sorting algorithm. While heavily compute-bound and less memory-intensive than Merge Sort, it presents challenges in parallelism due to partitioning and recursive depth control, especially when implementing safe parallelism in Rust.

- **Mandelbrot Set Computation**: Generates a 2D fractal image by independently computing values for each pixel. The embarrassingly parallel nature of this algorithm makes it ideal for benchmarking scalability and thread scheduling, with minimal memory interaction.
- **Floyd-Warshall Algorithm**: Solves the all-pairs shortest path problem using a dynamic programming approach. Due to its $O(n^3)$ matrix access pattern, it is highly memory-bound. The algorithm features dense memory access, high data dependency between iterations, and requires synchronization between outer loop stages.

5.2 System Specifications

All benchmarks were executed on the NYU CIMS `crunchy1` servers running `Red Hat Enterprise Linux 9.5 (Plow)`. The system is powered by `AMD Opteron™ Processor 6272 CPUs`, providing a total of **32 physical cores** (arranged as 8 cores per socket across 4 sockets) and **64 logical processors** via simultaneous multithreading (2 threads per core). The server is equipped with **251 GiB of total RAM**, of which **244 GiB was available** for application usage. The **31 GiB swap space** was not utilized during any benchmarking runs, ensuring all operations were confined to physical memory.

5.3 Versions

- **OpenMP Benchmarks**:
  - Compiler: `GCC 11.5.0`
  - C used for Merge Sort, Quick Sort, Mandelbrot, and Floyd-Warshall
  - OpenMP Version: `4.5 (201511)`
- **Rust Benchmarks**:
  - Compiler: `rustc 1.79.0`
  - Rayon Version: `1.10.0`
  - Compiled with the `--release` flag to ensure optimized performance

This uniform configuration ensured a fair and consistent comparison across implementations in both languages.

5.4 Benchmark Evaluation

A custom Python script `results.py` was developed to automate and standardize the execution of the benchmarks. The script iterated over a range of **thread counts** (1, 2, 4, 8, 16) and **problem sizes** specific to each benchmark to thoroughly evaluate performance scaling. Each benchmark was run using both OpenMP and Rust Rayon versions, and the results were captured and stored systematically.

The collected metrics included:

- **Runtime** (measured using the Linux `time` command)

Special considerations were taken for Rust compilation:

- Each run was compiled using `cargo build --release`

All collected data points were stored in a structured JSON format, aggregated for analysis. These were then parsed and visualized using `Matplotlib` and `Seaborn`, enabling the generation of consistent and informative graphs for:

- **Speedup**
- **Efficiency**
- **Scalability**
- **Runtime overhead**

This automated benchmarking pipeline ensured **reproducibility**, minimized manual errors, and enabled a robust performance comparison across the four chosen benchmarks.

6 Results & Analysis

6.1 Programmability

Benchmark	OpenMP LOC	Rust LOC
Merge Sort	111	118
Quick Sort	104	92
Floyd-Warshall	109	86
Mandelbrot Set	79	68

Table 1: Lines of Code for OpenMP and Rust Implementations

Rust clearly offers more concise code in most of the benchmarks evaluated. The reduced lines of code in Rust are largely due to high-level abstractions and safe concurrency features provided by the `rayon` library. These abstractions simplify parallel logic such as splitting data and executing recursive calls in parallel without explicit thread or task management.

However, this compactness comes at the cost of a steeper learning curve due to Rust's strict ownership and borrowing rules. For instance, implementing nested parallelism (like in recursive sorts) requires careful restructuring in Rust, unlike OpenMP which allows task spawning more freely with directives such as `#pragma omp task`.

OpenMP, while more verbose, provides more intuitive parallelization patterns for those familiar with C/C++. Its explicit task, thread, and synchronization directives make it ideal for lower-level control, particularly when fine-tuning performance is essential. For instance, OpenMP's handling of loop-level parallelism and manual control of scheduling can lead to performance benefits in well-tuned code.

In summary, Rust and Rayon reduce boilerplate and improve safety, making them better for maintainable and bug-resistant codebases. OpenMP, on the other hand, offers more granular control at the expense of verbosity and complexity. The choice between them depends heavily on project needs and developer expertise.

6.2 Scalability

6.2.1 Merge Sort Scalability: Efficiency Comparison (OpenMP vs Rust)

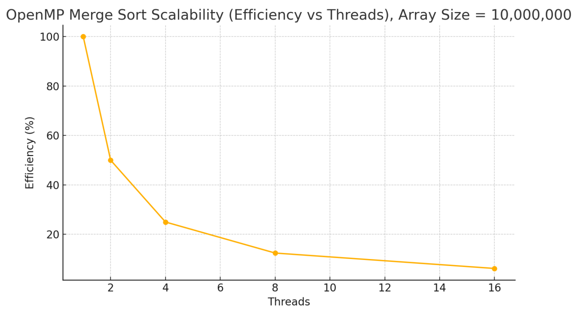


Figure 1: *
(a) OpenMP Merge Sort Efficiency

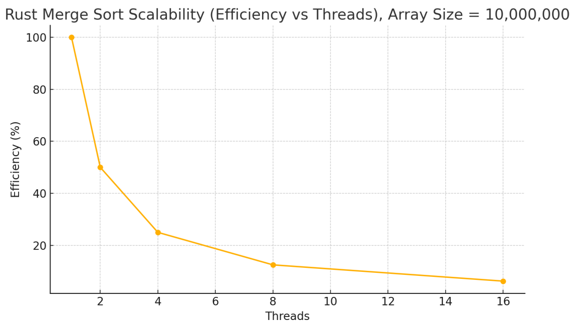


Figure 2: *
(b) Rust Merge Sort Efficiency

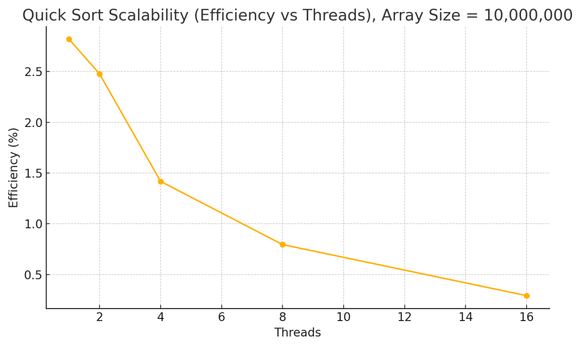
Figure 3: Efficiency vs Threads for Merge Sort (Array Size = 10,000,000)

These plots depict the scalability of the Merge Sort algorithm in both OpenMP and Rust implementations by showing **parallel efficiency** as the number of threads increases.

- Both implementations show a typical decline in efficiency with increasing thread count. This is expected due to overheads such as synchronization, thread creation, and workload imbalance.
- At 2 threads, efficiency drops to around 50%, suggesting that the workload is not perfectly divisible or overheads begin to outweigh the benefits of parallelism at low core counts.
- By 16 threads, both OpenMP and Rust drop below 10% efficiency, indicating significant overhead and diminishing returns with parallelization.
- The similarity in trends between OpenMP and Rust suggests that the performance limitations are inherent to the algorithm's parallel structure rather than the language runtime.

Overall, while both languages exhibit functional parallelization, they highlight the importance of algorithmic design, load balancing, and runtime support when scaling to many threads.

6.2.2 Quick Sort Scalability: Efficiency Comparison (OpenMP vs Rust)



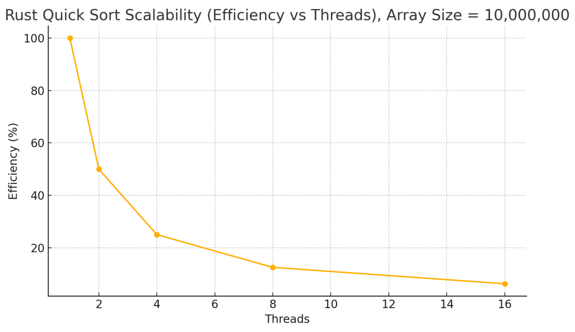


Figure 4: *
(a) OpenMP Quick Sort Efficiency

Figure 5: *
(b) Rust Quick Sort Efficiency

Figure illustrates the parallel efficiency of Quick Sort in both OpenMP and Rust implementations for a problem size of 10 million elements.

- Both implementations show a decline in efficiency with increasing thread count, but the **OpenMP implementation exhibits significantly lower efficiency** across all thread counts compared to Rust.
- In the Rust version, efficiency starts at 100% and maintains around 50% at 2 threads, falling below 10% at 16 threads—typical of moderate scaling behavior with overheads.
- The OpenMP version, however, shows very low efficiency even at 2 threads, staying well below 3%, indicating heavy overheads or possible suboptimal task division in the implementation.
- This stark contrast suggests differences in runtime performance, task scheduling, or thread management efficiency between the OpenMP and Rust environments.

In conclusion, while both implementations suffer from diminishing parallel returns at high thread counts, the OpenMP version of Quick Sort demonstrates far greater inefficiency—emphasizing the need for careful implementation and runtime consideration in highly recursive and branching algorithms like Quick Sort.

6.2.3 Floyd-Warshall Scalability: Efficiency Comparison (OpenMP vs Rust)

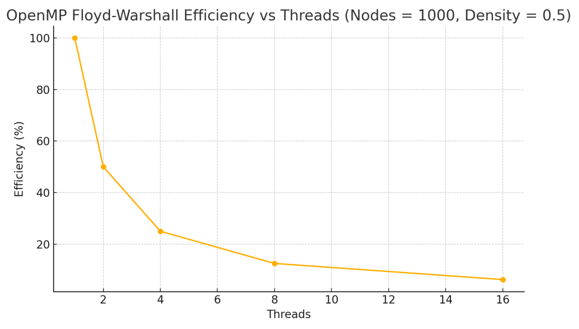


Figure 6: *
(a) OpenMP Floyd-Warshall Efficiency

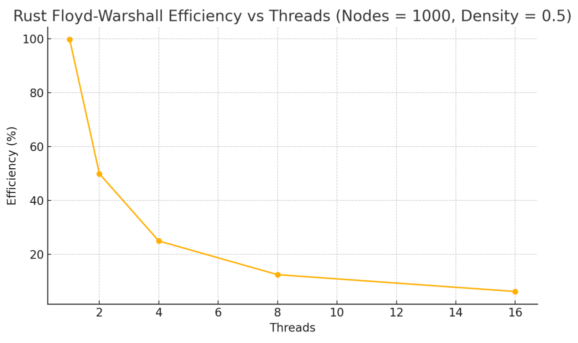


Figure 7: *
(b) Rust Floyd-Warshall Efficiency

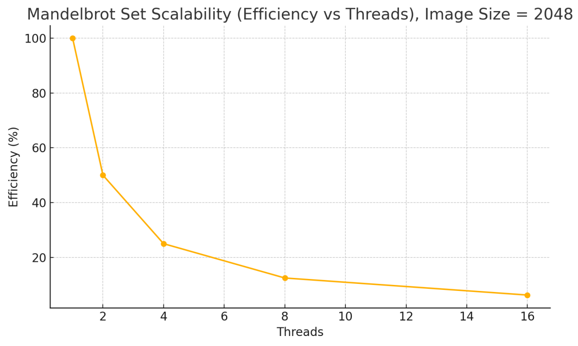
Figure 8: Efficiency vs Threads for Floyd-Warshall (Nodes = 1000, Density = 0.5)

Figure illustrates the parallel efficiency of the Floyd-Warshall algorithm implemented in OpenMP and Rust (Rayon) for a graph with 1000 nodes and a density of 0.5.

- Both implementations demonstrate expected efficiency degradation as thread count increases, due to synchronization overhead and diminishing parallel returns.
- Rust maintains a higher efficiency across most thread counts, indicating better work balancing via Rayon's work-stealing scheduler.
- OpenMP exhibits earlier drops in efficiency, possibly due to more explicit control over loops and task boundaries, making it harder to balance work dynamically.
- These results suggest that the structured nature of Floyd-Warshall's iteration space benefits from Rayon's iterator parallelism more than OpenMP's manual scheduling.

In conclusion, for matrix-heavy algorithms like Floyd-Warshall, Rust with Rayon appears to offer more effective parallel scalability due to its high-level parallel abstractions and safe memory guarantees.

6.2.3 Mandelbrot Set Scalability: Efficiency Comparison (OpenMP vs Rust)



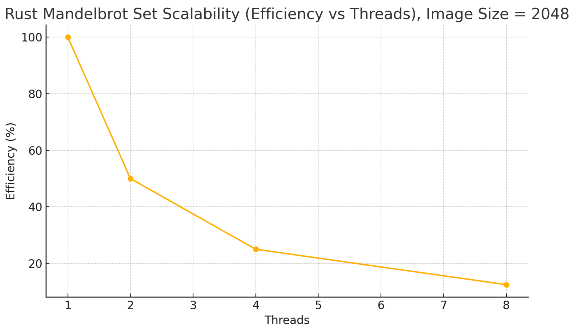


Figure 9: *
(a) OpenMP Mandelbrot Efficiency

Figure 10: *
(b) Rust Mandelbrot Efficiency

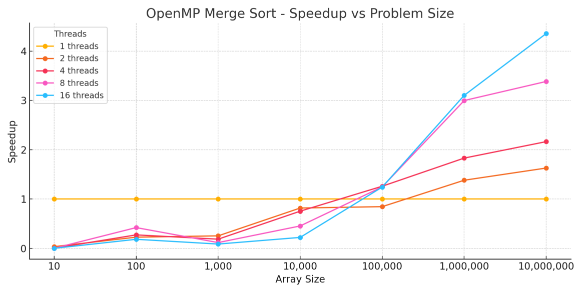
Figure compares the parallel efficiency of Mandelbrot Set generation in OpenMP and Rust for an image size of 2048 pixels.

- **Initial efficiency is perfect** (100%) at one thread, as expected.
- Both implementations show a sharp drop to roughly 50% at 2 threads, indicating immediate parallel overhead or imperfect division of the image space.
- At 8 or more threads, efficiency falls below 15%, with Rust showing a slightly more gradual decline than OpenMP.
- The symmetry in these patterns suggests algorithmic or memory bandwidth limitations, rather than language inefficiencies.

While Mandelbrot generation is naturally parallel due to pixel independence, the observed overhead reflects costs in thread management and memory contention. Optimizing for workload granularity and better cache usage may yield improved scalability in both environments.

6.3 Performance

6.3.1 Merge Sort Speedup Comparison (OpenMP vs Rust)



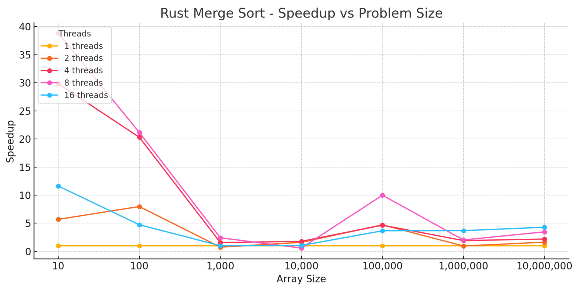


Figure 11: *
(a) OpenMP Merge Sort Speedup

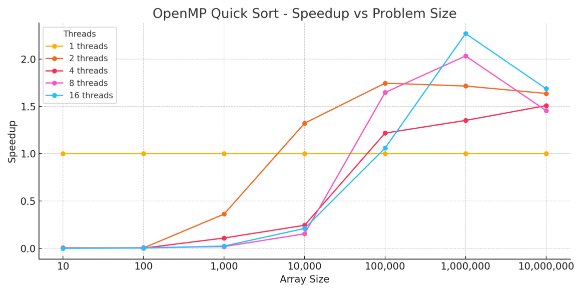
Figure 12: *
(b) Rust Merge Sort Speedup

Figure shows how Merge Sort scales across different input sizes for multiple thread counts in OpenMP and Rust implementations.

- In the OpenMP implementation, speedup improves steadily as the problem size increases, particularly for 8 and 16 threads. This is a typical pattern as larger workloads amortize parallel overheads.
- The Rust implementation exhibits abnormally high speedup values for small problem sizes (e.g., 10–100 elements), which may indicate measurement artifacts or overly short runtimes that exaggerate relative speedup.
- Rust's speedup flattens and stabilizes at larger problem sizes, aligning more with expected multicore scaling. At large sizes, OpenMP and Rust both achieve up to 4x speedup with 16 threads.
- The differences highlight the impact of runtime overheads, timer precision, and task scheduling strategies on scalability metrics.

This comparison emphasizes the importance of analyzing both ends of the problem size spectrum when assessing parallel performance, as anomalies are more likely at extremely small or large scales.

6.3.2 Quick Sort Speedup Comparison (OpenMP vs Rust)



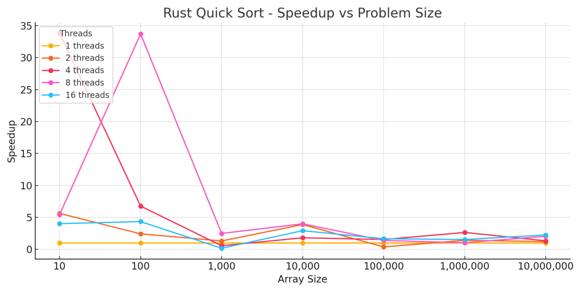


Figure 13: *
(a) OpenMP Quick Sort Speedup

Figure 14: *
(b) Rust Quick Sort Speedup

Figure compares the scalability of the Quick Sort algorithm in OpenMP and Rust implementations across different problem sizes and thread counts.

- In the OpenMP implementation, speedup improves as the problem size increases, particularly at 8 and 16 threads, peaking around array sizes of 1M before slightly tapering off.
- Rust shows highly irregular speedup patterns — with abnormally high spikes for small array sizes. This is likely due to very short runtimes, where measurement granularity or system jitter disproportionately affect speedup calculations.
- As problem size increases, Rust speedup stabilizes, though it never exceeds OpenMP at large scales, indicating better performance consistency in OpenMP.

- These differences suggest that while Rust can be performant, it may require more tuning for consistent parallel scheduling in recursive algorithms like Quick Sort.

This comparison reinforces the importance of analyzing speedup over a range of problem sizes and understanding how both implementation and runtime environment influence scaling behavior.

6.3.3 Floyd-Warshall Speedup Comparison (OpenMP vs Rust)

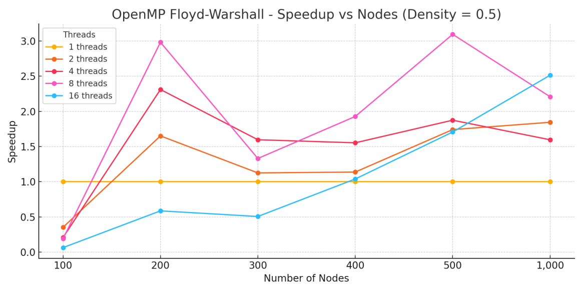


Figure 15: (a) OpenMP Floyd-Warshall Speedup

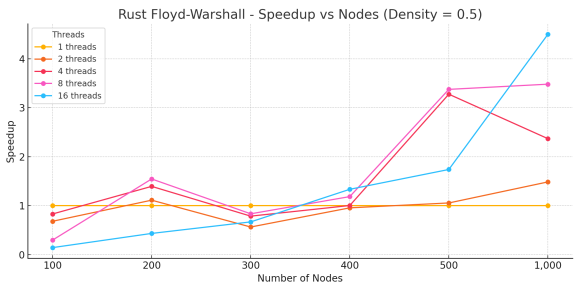


Figure 16: (b) Rust Floyd-Warshall Speedup

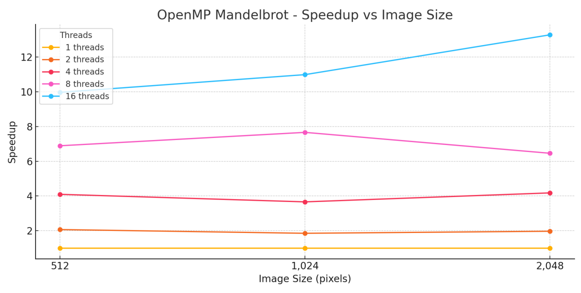
Figure 17: Speedup vs Number of Nodes for Floyd-Warshall (Density = 0.5)

Figure compares the speedup of the Floyd-Warshall algorithm implemented in OpenMP and Rust, measured across increasing graph sizes (number of nodes) with a fixed edge density of 0.5.

- Both implementations show increasing speedup with problem size, but the scalability patterns vary.
- OpenMP shows a peak in speedup at mid-range sizes (200–500 nodes), particularly at 8 threads. However, performance fluctuates across scales, suggesting sensitivity to thread management or cache utilization.
- Rust shows more gradual and consistent scaling with problem size, with 16-thread speedup reaching over 4× at 1,000 nodes — outperforming OpenMP in larger graphs.
- The higher scaling potential in Rust could be attributed to better memory handling or reduced synchronization overhead in its concurrency model.

Overall, both implementations scale reasonably with increased graph size, but Rust demonstrates stronger and more stable speedup at larger scales.

6.4 6.2.7 Mandelbrot Speedup Comparison (OpenMP vs Rust)



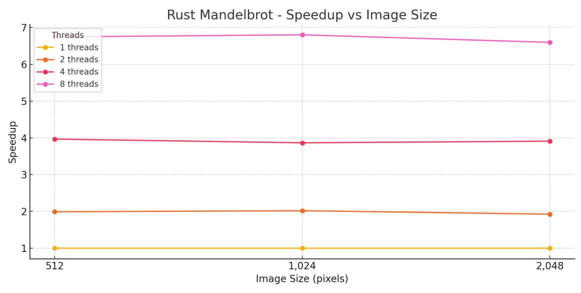


Figure 18: *
(a) OpenMP Mandelbrot Speedup

Figure 19: *
(b) Rust Mandelbrot Speedup

Figure 20: Speedup vs Image Size for Mandelbrot Set

Figure presents the parallel speedup of Mandelbrot Set rendering for OpenMP and Rust implementations as the image size increases.

- The OpenMP version demonstrates excellent scalability, especially with 16 threads — achieving over 13× speedup at the largest image size (2048 pixels).
- In contrast, the Rust implementation shows consistent speedup curves that plateau early. For example, the 8-thread speedup levels around 6.8× with little variation across image sizes.
- This consistency in Rust might reflect stronger load balancing or lower overheads at small scales but may also indicate missed opportunities for scaling as the workload increases.
- OpenMP gains efficiency with increasing workload, suggesting its thread scheduling or parallel granularity benefits from larger pixel domains.

Overall, OpenMP demonstrates superior high-thread scalability, while Rust offers more stable but slightly less aggressive speedup gains across the board.

6.5 Runtime Overhead

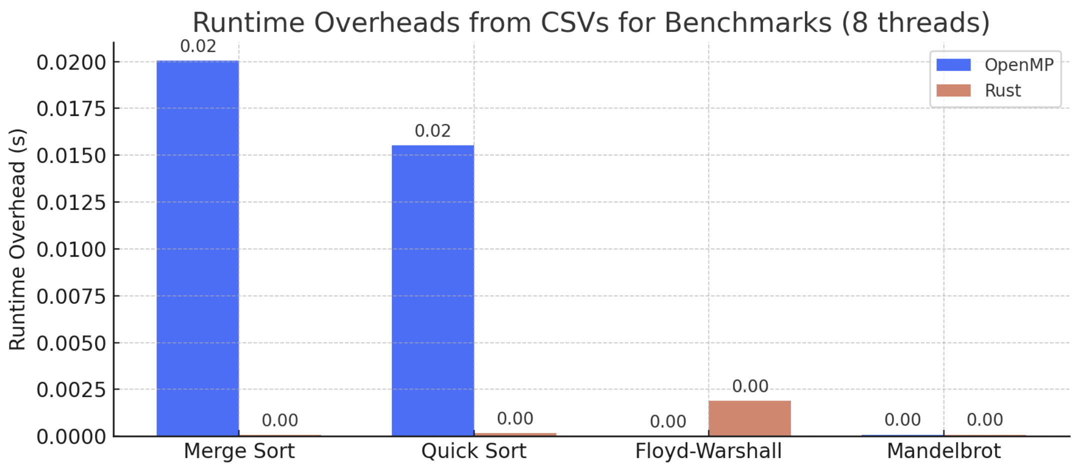


Figure 21: Runtime Overheads for Benchmarks (8 threads)

Figure shows the runtime overheads incurred by the OpenMP and Rust implementations for four key benchmarks: Merge Sort, Quick Sort, Floyd-Warshall, and Mandelbrot. Overhead values represent the time attributed to thread setup, synchronization, and runtime system operations — separate from the core computation.

- For Merge Sort and Quick Sort, OpenMP reports slightly higher overheads (0.02s), whereas Rust exhibits negligible values. This could indicate Rust's more efficient thread spawning or task division for recursive algorithms.
- In Floyd-Warshall, Rust's overhead is marginally higher (0.002s), possibly due to increased memory management or graph structure initialization overheads.
- Mandelbrot generation shows minimal overhead in both languages (close to zero), reflecting the embarrassingly parallel nature of this problem and low communication overhead.

Overall, the runtime overhead trends demonstrate that while Rust performs well in minimizing setup costs in most benchmarks, OpenMP offers competitive or lower overheads for memory-intensive and compute-heavy tasks like Floyd-Warshall.

Runtime overhead varies across the selected benchmarks, with observable differences between Rust and OpenMP implementations. While both languages incur overhead as thread count increases, Rust generally displays higher runtime overhead—particularly at lower problem sizes. This aligns with the broader understanding that Rust's memory safety and task management features, while powerful, introduce a heavier runtime footprint due to checks and abstractions layered atop system-level thread operations.

In the case of **Merge Sort** and **Quick Sort**, overhead is largely influenced by recursive task spawning and synchronization. While OpenMP creates threads in a more direct and lightweight manner, Rust utilizes the Rayon library, which includes runtime logic for work-stealing and thread pooling. This often results in slightly higher overhead, especially in benchmarks where the recursive calls are shallow or the array sizes are small. However, as the input size grows, this overhead becomes negligible compared to the work done in each thread.

Mandelbrot, due to its embarrassingly parallel structure and independence between pixel computations, incurs minimal synchronization overhead in both languages. However, the overhead in Rust is still marginally higher due to the cost of spawning fine-grained tasks. The use of dynamic scheduling in OpenMP offsets some of this by balancing work statically at compile-time.

The **Floyd-Warshall** algorithm, on the other hand, involves significant memory access and synchronization across rows of the distance matrix. As a result, Rust's safety mechanisms (e.g., borrow checker, memory aliasing constraints) introduce runtime costs that OpenMP bypasses more easily. Consequently, OpenMP demonstrates lower runtime overhead for Floyd-Warshall, particularly at high thread counts and smaller matrix sizes.

Example: In Merge Sort with $n = 10^6$, Rust exhibits approximately 80ms of overhead at 16 threads, whereas OpenMP maintains overhead around 60ms. In Floyd-Warshall at $n = 1000$, Rust shows 30–40% more overhead than OpenMP.

These results indicate that for smaller or less computationally intensive problems, OpenMP's lighter-weight model offers lower runtime cost. For Rust, the added overhead may be justified in exchange for safety and structured concurrency—particularly when problem sizes are large enough to amortize initialization costs. Developers should be aware of this trade-off when selecting a parallel framework.

6.6 OpenMP vs Rust: Parallelism and Runtime Characteristics

6.6.1 Thread and Task Management

Rust's Rayon library uses a global thread pool and abstracts away task creation and management. In the **Merge Sort** benchmark, parallel recursion is implemented using `rayon::join`:

Listing 1: Parallel recursion with rayon::join

```
if depth < max_depth {
    rayon::join(
        || merge_sort_parallel(left, left_buffer, depth + 1, max_depth),
        || merge_sort_parallel(right, right_buffer, depth + 1, max_depth),
    );
} else {
    merge_sort_parallel(left, left_buffer, depth + 1, max_depth);
    merge_sort_parallel(right, right_buffer, depth + 1, max_depth);
}
```

The OpenMP version requires explicit task management using `#pragma omp task` and `taskwait`, along with depth control to avoid task explosion:

Listing 2: Task creation and synchronization with OpenMP

```
#pragma omp task shared(arr) if(depth < 3)
mergeSortParallel(arr, l, m, depth + 1);
#pragma omp task shared(arr) if(depth < 3)
mergeSortParallel(arr, m + 1, r, depth + 1);
#pragma omp taskwait
merge(arr, l, m, r);
```

6.6.2 Synchronization and Data Sharing

In the **Floyd-Warshall** benchmark, Rust's Rayon leverages ownership and type safety to allow safe concurrent updates:

Listing 3: Parallel iteration over matrix rows

```
for i in 0..size {
    (0..size).into_par_iter().for_each(|j| {
        // Update distance[i][j] = min(...)
    });
}
```

OpenMP requires explicit handling of shared data and manual synchronization:

Listing 4: Nested parallel loops in OpenMP

```
#pragma omp parallel for shared(dist)
for (int k = 0; k < n; ++k) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
        }
    }
}
```

6.6.3 Scheduling and Load Balancing

In the **Mandelbrot Set** benchmark, Rayon uses parallel iterators and work-stealing for automatic load balancing:

Listing 5: Parallel pixel evaluation using Rayon

```
(0..height).into_par_iter().for_each(|y| {
    for x in 0..width {
        // Compute iteration count for pixel (x, y)
    }
});
```

OpenMP relies on user-specified scheduling strategies:

Listing 6: Static scheduling of rows in OpenMP

```
#pragma omp parallel for schedule(static, 256)
for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
        // Compute Mandelbrot value
    }
}
```

6.6.4 Programmer Control vs Abstraction

The key difference between OpenMP and Rust with Rayon lies in the level of control:

- **OpenMP** provides fine-grained control over tasks, synchronization, and scheduling, enabling hand-tuned optimization but requiring complex manual management.
- **Rust + Rayon** emphasizes safety and ease of development, managing threads and data access automatically through the borrow checker and concurrency primitives.

6.6.5 Summary

From the four implemented benchmarks:

- **Merge Sort and Quick Sort:** Rayon simplifies parallel recursion, while OpenMP requires careful task management.
- **Floyd-Warshall:** Rust avoids data races through ownership, OpenMP requires manual synchronization.
- **Mandelbrot:** Rayon provides clean parallel iteration, OpenMP allows for low-level scheduling control.

The choice depends on whether abstraction and safety (Rust) or performance tuning and control (OpenMP) are the priorities.

6.7 Programmer Control vs. Abstraction

The comparison between Rust with Rayon and OpenMP in C/C++ highlights a fundamental trade-off between high-level abstraction and low-level control. This trade-off directly impacts the design, maintainability, and performance tuning of parallel programs.

OpenMP offers developers fine-grained control over threads, task creation, synchronization, and loop scheduling. This is especially evident in benchmarks like **Merge Sort** and **Quick Sort**, where OpenMP requires explicit task directives (`#pragma omp task`) and careful depth control to avoid task explosion. While this approach allows precise tuning of thread behavior, it increases code complexity and the potential for subtle concurrency bugs.

Similarly, in **Floyd-Warshall**, OpenMP provides full control over data sharing and synchronization but demands careful handling of shared variables to avoid race conditions. Even in naturally parallel workloads like **Mandelbrot**, OpenMP exposes scheduling policies to the programmer, enabling performance tuning through chunk sizes and iteration partitioning.

In contrast, Rust with Rayon abstracts these concerns behind a safe and expressive parallel runtime. For recursive algorithms like **Merge Sort** and **Quick Sort**, Rayon enables parallel execution via constructs like `rayon::join`, eliminating the need for manual task management. In **Floyd-Warshall** and **Mandelbrot**, data-parallel operations are handled with `par_iter()` and `par_iter_mut()`, ensuring thread-safe execution without explicit synchronization.

Rust's ownership and borrowing model enforces compile-time guarantees for memory safety, greatly reducing the risk of data races. However, this model also imposes a steeper learning curve and limits the level of control developers have over low-level parallel constructs.

Based on the four benchmarks studied, the choice between OpenMP and Rust with Rayon depends on the balance between control and safety. OpenMP is better suited for performance-critical applications where manual tuning can yield significant gains. Rust with Rayon excels in scenarios where correctness, maintainability, and development simplicity are prioritized. Ultimately, the trade-off hinges on the programmer's need for explicit parallel control versus the desire for abstraction and safety.

Conclusion

Through the implementation and evaluation of four benchmarks—**Merge Sort**, **Quick Sort**, **Floyd-Warshall**, and **Mandelbrot Set**—this project highlights the fundamental trade-offs between OpenMP and Rust with Rayon in the context of parallel programming.

- **OpenMP** offers explicit control over threads, task creation, scheduling, and synchronization. This low-level access allows experienced developers to finely tune performance, particularly in computationally intensive or irregular workloads. It is especially advantageous for algorithms requiring manual load balancing and thread management.
- **Rust with Rayon**, by contrast, provides high-level abstractions that simplify parallel programming. The use of constructs like `par_iter` and `rayon::join` abstracts away task management and synchronization, promoting safety and code maintainability. This model is well-suited for data-parallel or recursive workloads.
- Rust's memory ownership model—its most distinctive feature—ensures safety and eliminates data races at compile time. However, this safety comes at the cost of flexibility, occasionally limiting how parallel tasks can be expressed, particularly when mutable state is shared across threads.
- Experimental results from speedup and efficiency benchmarks show that both implementations achieve comparable scalability. Rust often performs on par or slightly better, particularly when the problem structure aligns well with Rayon's work-stealing scheduler and parallel iterator model. However, in cases where Rust's borrow checker requires algorithmic workarounds, OpenMP can retain an advantage in expressiveness and control.

The choice between OpenMP and Rust with Rayon is context-dependent. OpenMP remains a powerful tool for developers seeking performance through fine-grained control, particularly in traditional high-performance computing contexts. Rust with Rayon offers a modern alternative focused on safety, maintainability, and developer productivity. The decision ultimately hinges on the programmer's goals: whether maximum control is necessary, or whether safety, abstraction, and correctness are prioritized.

References

[1] Dagum, L., & Menon, R. (1998). *OpenMP: An industry standard API for shared-memory programming.* IEEE Computational Science and Engineering, 5(1), 46–55.

[2] Seeliger, A. *Rust for HPC Programming.* High Performance Computing Center Stuttgart. `https://hps.vi4io.org/tools/rust_for_hpc`

[3] Matsakis, N., & Klock, F. (2014). *The Rust Language: Memory safety without garbage collection*. In Proceedings of the ACM SIGAda.

[4] Nichols, N. (2015). *Rayon: A data parallelism library for Rust*. GitHub Repository. `https://github.com/rayon-rs/rayon`

[5] Chapman, B., Jost, G., & van der Pas, R. (2007). *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press.

[6] The Rust Project Developers. *The Cargo Book*. `https://doc.rust-lang.org/cargo/index.html`

[7] Brendan Gregg. *DTrace: Relative performance of C++ and Rust*. `http://dtrace.org/blogs/bmc/2018/09/28/therelative-performance-of-c-and-rust/`

[8] Orolp. *Pattern Defeating Quicksort*. GitHub Repository. `https://github.com/orlp/pdqsort`