PARALLEL BENCHMARKING: A PERFORMANCE COMPARISON OF RUST RAYON AND C/C++ OPENMP

Safi Patel, Eugene Chang & Nathan Nagulapalli
New York University
CSCI-GA 3033 Multicore Processors & Architecture
{sp6559, ec4338, nsn4960}@nyu.edu

ABSTRACT

From super computers to smartphones, multi-core processing dominates the current computing landscape. Solving future computation problems will rely increasingly on the ability of the programmer, engineer, or scientist to properly leverage the immense power of these multi-core machines. Part of this challenge is to identify an appropriate medium through which parallelization can be accomplished. In this regard, the programming language used for any particular problem is perhaps the most important choice for ultimate success. The standard today is OpenMP, a widely respected and mature set of directives accessible through C/C++. A new rising star in the programming language world is Rust, which offers Rayon, it's own parallelization library. A comparison of multi-core programming between the two benefits all future participants of the multi-core revolution. In this study, six parallel program benchmarks are used to test OpenMP and Rust across a set of criteria. Surprisingly, while performance between languages was roughly comparable, Rust's conciseness, language abstraction and safety features significantly hampered granular control over the program, which certain contexts require. This suggests programmer control should be a larger consideration in the choice of language than initially anticipated.

1 INTRODUCTION

As multi-core processors proliferate throughout computing, and computing itself expands into every level of society, the ability to properly harness parallel computation becomes increasingly important. Within this paradigm, the de facto standard in 2024 is OpenMP, a set of API directives overlaid onto C, C++, and Fortran. With the popularity of C and C++, and the maturity that Open MP has achieved in it's now 20+ year existence, it's easy to wonder why one should consider any alternatives. However, new languages and approaches are always being developed and must be considered in order to maintain the state of the art. One such alternative to C and C++ is Rust, which has emerged over the last decade. Combining the speed, conciseness, and lack of garbage collection of C with the memory safety of garbage collected languages like Java and Python, Rust has increasingly made its way into the core systems of many software companies. Thus, it is worth exploring both approaches to parallel computation to understand their respective strengths, weaknesses, and level of applicability to situational circumstances.

At its core, OpenMP is a set of compiler directives and callable runtime library routines [1]. OpenMP's approach to parallelism is the "for loop" primitive; within this construct, OpenMP executes a fork/join model, in which the "parent" thread forks into multiple "children" threads in the parallel region before joining back into a single thread to re-enter the sequential region [1]. Of course, OpenMP provides a great many tools to avoid race conditions, false sharing, and maintain cache coherence. Atomics, Locks, Critical regions etc. are all used to avoid the problems one may encounter when dealing with shared memory amongst several threads. In almost all instances, control is left up to the programmer.

While OpenMP is a set of library routines, Rust itself is a language. This is of course, a first major difference. Within Rust, there are two main options to approach parallelism. The first is to manually manage threads via the 'thread::spawn' command. As with any manual management, this gives greater control to the programmer, but with a cost of increased complexity. The approach taken in this study is to focus on Rayon, the widely used parallelism "crate" (or library, in Rust nomenclature) used by many in the Rust community (the Rayon repository on Github has over 11,000 stars as of this writing). Rayon offers two main primitives to utilize parallelism: parallel iterators and custom tasks. Iterators require a data structure conducive to iteration (arrays, vectors etc), and also require understanding of a core feature within Rust: ownership. Core to the memory management of Rust, any value in Rust has exactly one owner at a time. In the scope of parallelism, this adds some additional complexity that will be explored throughout this paper.

Central Question: How does OpenMP compare with Rayon as a parallelism library? In which situations should one be used over the other? How are performance and scalability affected in different environments? Through a set of 6 benchmarks - each written in both C/C++ with OpenMP and Rust with Rayon - these questions will be explored in depth against a series of metrics. A set of recommendations will be presented at the end of the evaluation for programmers considering both as options for their parallel task.

2 Related Works

There is an abundance of literature on both Rust and OpenMP, but interestingly there is a dearth of study regarding the comparison of both. There are three main categories of literature that exist today: High Performance Computing with Rust, Rust vs C, and various studies of OpenMP. Each of these areas are detailed below:

2.1 High Performance Computing with Rust

As an emergent systems programming language designed with both safety and performance in mind, it stands to reason that Rust seems to be a worthy alternative to the standard High Performance Computing languages of Fortran and C++ [2]. Within this context, the realm of numerical computation has been investigated extensively with Rust, including it's libraries for complex numbers, linear algebra operations, and matrix operations [3]. Of course, parallelism and the Rayon crate specifically have been investigated within this context, as well as it's support for GPU programming [3]. In general, while High Performance Computing is obviously highly relevant to parallel programming, there are many other elements (especially within the scientific and mathematical domains being investigated) that are not necessary when thinking about more general purpose parallelism. Additionally, Rust as it compares to OpenMP has not been thoroughly investigated in this area of research.

2.2 Rust vs C

Of course, Rust has been compared to other languages, specifically it's most commonly used high performant alternative, C. Typically, this involves the major differences between the languages, which often boil down to how these two languages handle ownership and memory. Rust has become notable for its avoidance of memory mismanagement such as data races and errors in accesses [6]. This is accomplished through the concept of "ownership", in which every values has exactly one variable owner at a given time. The Rust compiler will perform security checks and prevent improper ownership from taking place [6]. This level of memory control is very different from the approach in the C programming language, which has no such concept of ownership and which grants much of the responsibility of memory management to the programmer. This of course can lead to the memory errors mentioned earlier that have produced many software bugs for many unfortunate software developers. The literature does indeed discuss the differences in Rust and C, and even the different approaches to concurrency, but an in-depth comparison of Rust and OpenMP is still lacking. Most researchers have focused on the differences in languages, without investigating the differences in libraries.

2.3 OpenMP in-depth

While not necessarily studied in comparison with Rust, OpenMP has of course been investigated on its own by the research community. With over 20 years of life and a vibrant ecosystem, the OpenMP community is a thriving and curious group of developers and researchers. Widely considered the "Industry Standard" for shared memory parallelism [1], the various directives, APIs, and thread models of OpenMP have been benchmarked and tested with various levels of intensity. Within the fields of numerical analysis and applied mathematics, OpenMP has been used to investigate and simulate various scientific phenomena to great success. Given it's flexible nature, it has become a great choice for researchers from fields as diverse as elasto-viscoplastic Fast Fourier transform-based micromechanical solvers to multilevel Green's function interpolation methods [7,8]. These works, while obviously requiring an in-depth level of OpenMP, also require scientific expertise of the given field. Parallelizing various implementations and benchmarks as computers become faster and more capable can obviously be a fruitful endeavor, and this body of work served as inspiration for the benchmarks chosen later in this work.

3 Proposed Idea

The primary goal of this study is to compare the performance and usability of two parallel programming paradigms: OpenMP, widely used for parallel programming in C and C++, and Rayon, a modern parallelization library built on Rust's safety-first principles. To achieve this, the project aims to create a diverse suite of benchmarks that cover a broad range of computational scenarios, allowing us to comprehensively evaluate the strengths and weaknesses of each language and parallelization approach.

The benchmark suite includes six programs designed to represent distinct computational and memory access patterns commonly encountered in scientific computing, bio-informatics, and general-purpose parallel workloads. By incorporating benchmarks that are computation-bound, memory-bound, or both, the suite provides insights into how each language and library performs under various real-world conditions. This diversity allows for the evaluation of raw performance metrics, such as speedup and scalability, as well as language-specific features like memory management efficiency, thread control, and compilation speed. Furthermore, qualitative aspects of programmability and usability for developers are analyzed, including code complexity, ease of implementing parallelization, and the level of control offered by each paradigm.

3.1 Categories For Comparison

The benchmark suite evaluates the following key categories:

1. Programmability: Assessed based on code complexity and lines of code, analyzing how easy or difficult it is for a programmer to implement the benchmarks in both languages.
2. Scalability: Measured by evaluating efficiency as thread count increases, while keeping the problem size fixed, to understand how well each language's parallel model scales.
3. Performance: Measured by speedup which is calculated as the ratio of 1-thread runtime to n-thread runtime for each benchmark.
4. Memory Management: Evaluated based on memory allocation and access patterns to measure efficiency.
5. Runtime Overhead: This included the time spent on thread creation, synchronization, and termination, calculated by subtracting workload runtime from the total program runtime.
6. Compilation Speed: Measured using the time command to determine how long it took to compile each program.
7. Programmer Control: Qualitatively analyzed to compare the degree of control programmers had over aspects like thread-to-core mapping and shared/private variable management.

4 Experimental Setup

4.1 Benchmarks

The following six benchmark programs were chosen to represent diverse workloads in both computation and memory access:

- LU Decomposition: Performs LU factorization, widely used in linear algebra and scientific computing, and while computation-heavy, exhibits a high degree of memory access.
- Quadrature: Uses a numerical method based on trapezoidal quadrature to approximate definite integrals and, given the number of function evaluations required to be stored and read, is memory bounded.
- Histogram: Calculates a frequency histogram for a large dataset, involving significant memory writes but few reads.
- Merge Sort: Implements a parallelized version of merge sort, involving recursive computations with frequent memory access and high degree of computation required as with most sorting algorithms.
- Needleman-Wunsch Algorithm: A dynamic programming algorithm used for sequence alignment in bioinformatics, involving matrix computations and significant memory access.
- Monte Carlo Pi Estimation: Uses a Monte Carlo method to approximate the value of pi, involving random number generation and independent calculations, making it an Embarrassingly parallelizable problem.

4.2 System Specifications

The benchmarks were executed on the NYU CIMS crunchy2 servers running Red Hat Enterprise Linux 9.5 (Plow). The system is equipped with AMD Opteron(TM) Processor 6272 CPUs, providing 32 physical cores (8 cores per socket across 4 sockets) and 64 logical processors with 2 threads per core. The server has 251 GiB of total RAM, with 244 GiB available for applications, and a 31 GiB swap space, which was not utilized during the benchmarking runs.

4.3 Versions

The LU Decomposition and Quadrature benchmarks were written in C++ OpenMP, while the others were implemented in C OpenMP. Each benchmark was also implemented in Rust using the Rayon parallelization library.

For OpenMP benchmarks, GCC 11.5.0 was used, while the C++ programs were compiled with G++ 11.5.0, both from the Red Hat 11.5.0-2 distribution, which supports OpenMP 4.5 (201511). Rust benchmarks were implemented using the Rayon parallelization library, version 1.10.0, alongside rustc 1.79.0, also provided by Red Hat. This configuration ensures consistency as well as sufficient computational and memory resources to evaluate the scalability and performance of all benchmarks.

4.4 Benchmark Evaluation

A Python script "results.py" to automate running the programs with various problem sizes and thread counts was written. The script iterates over predefined configurations - specifically different thread counts (e.g., 1, 2, 4, 8, 16, 32) and various configured problem sizes per benchmark - and executes each benchmark in both Rust and OpenMP. This automation ensures reproducibility and minimizes human error during the testing process, as well as making running the entire suite much easier. The results, including runtime, memory usage, and other performance metrics, are collected. We use the "time" linux command, and also the perf command. From it, we can get the information to understand the cache loads and cache misses.
One important thing to note is the use of "–release" when running the Rust cargo commands as compiling for the release version has a significant performance difference. Also, this Python script runs "cargo clean –release" before the "cargo build –release" compile command. This is because

the build command will normally only run if files are changed, otherwise it uses the cached builds. Cleaning gives us the true compile time for Rust. Every suite run adds more data points to the history by aggregating them into a JSON file. Then, to analyze the data, the script parses these results and uses the Matplotlib library (with the Seaborn module) to plot graphs for comparison. These visualizations highlight key trends such as speedup, efficiency, scalability, memory usage, and runtime overhead, providing a clear basis for evaluating the performance and efficiency of Rust and OpenMP implementations across diverse workloads. This automated pipeline streamlines the benchmarking process and ensures comprehensive data collection for analysis.

4.4.1 GITHUB REPOSITORY

All experimental setup code and benchmarks:
`https://github.com/egnechng/Rust-OpenMP` To run the benchmarks, follow the instructions in the ReadMe on the specified computing machine.

5 RESULTS & ANALYSIS

5.1 PROGRAMMABILITY

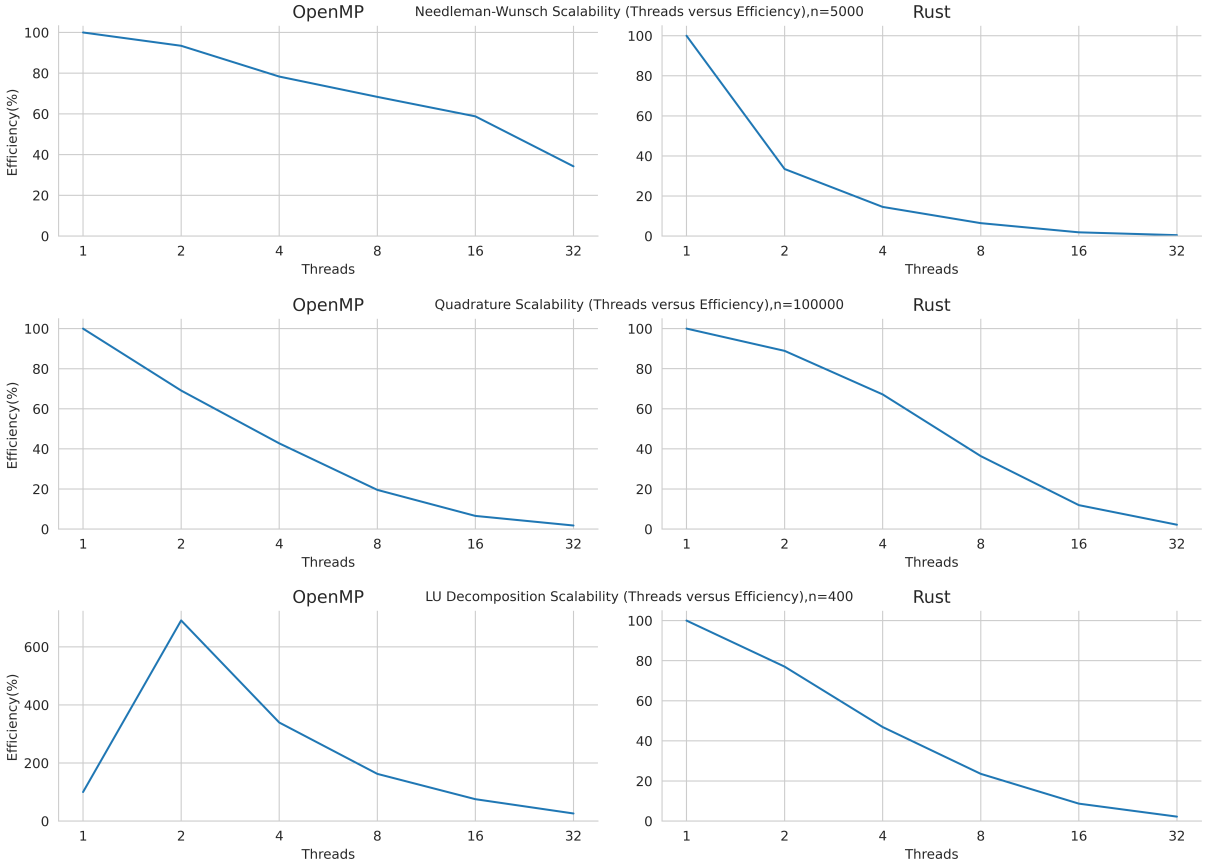
Benchmark	OpenMP	Rust
LU Decomposition	167	111
Quadrature	120	68
Merge Sort	111	118
Histogram	91	79
Monte Carlo	58	44
Needleman-Wunsch	135	124

Table 1: Lines of Code

Rust clearly is a more compact and concise language than C and OpenMP. However, Rust comes with downsides that also must be considered. While the language itself is compact relative to C, the parallel library chosen (Rayon) in combination with key Rust principles such as ownership made for a less intuitive understanding of parallelism. While OpenMP is fundamentally grounded in for loops, Rayon is meant most for iterators of data structures. This consideration, along with the lack of mutability across for loops made nested iteration quite difficult and hard to fully implement efficiently. In this way, OpenMP was a more intuitive and programmable set of directives and principles. That said, if parallelism is required for an iterable data structure (without nesting) then Rust and Rayon seem to be solid choices, as long as the concept of ownership is properly understood.

5.2 Scalability

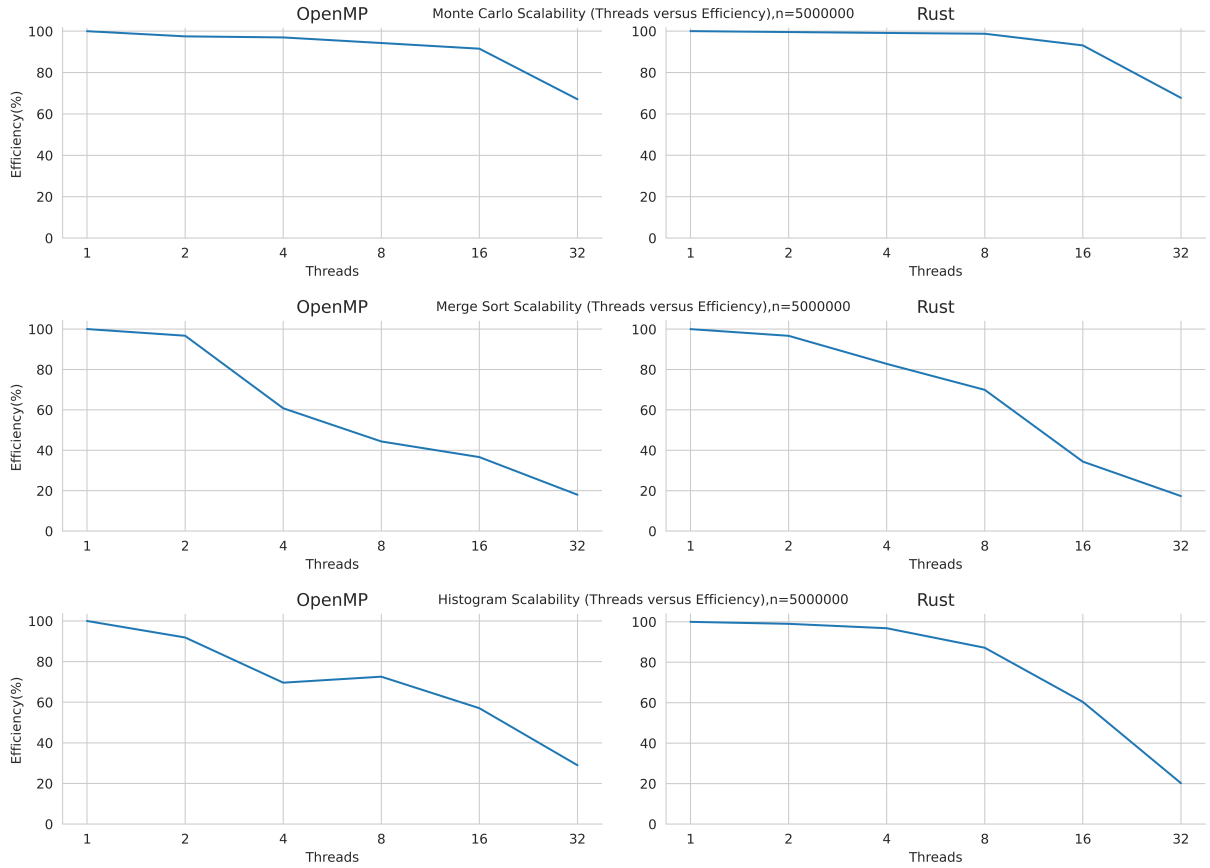
5.2.1 Memory Bound



In the memory bounded benchmarks, efficiency suffers as the number of threads is increased. Whether due to dynamic programming constraints (Needleman-Wunsch), computing and storing many functional values (quadrature), or updating many individual matrix values (LU Decomposition), a high degree of memory access impedes a large number of threads from operating together as the communication and synchronization overhead increases. Comparing Rust with OpenMP, the efficiency performs slightly better as thread count goes up. This is likely due to the Rust compiler's ability to perform memory checks pre-run time. The sole exception to this is the Needleman-Wunsch algorithm. In this case, the Rust implementation required work-arounds in order to parallelize the dynamic programming implementation, with threads having to mutate on the diagonals of the matrix but still having to access the cells of the previous diagonals; this meant having to fight against the memory protection mechanisms since the diagonals of matrix don't neatly fit into partitions. These work-arounds impeded performance, due to the intrinsic constraints of the language itself.

One area of note here is the LU-Decomposition graph, which seemingly has an extraordinary level of efficiency at the lower thread count. This could be due to the gains made by hyper-threaded cores and the high degree of nested for-loops in the algorithm, or due to the initial gains made by the compiler optimization. The Rust implementation, with it's limited mutability of iterators, is far less able to take advantage of these optimizations since a buffer is used to update the values.

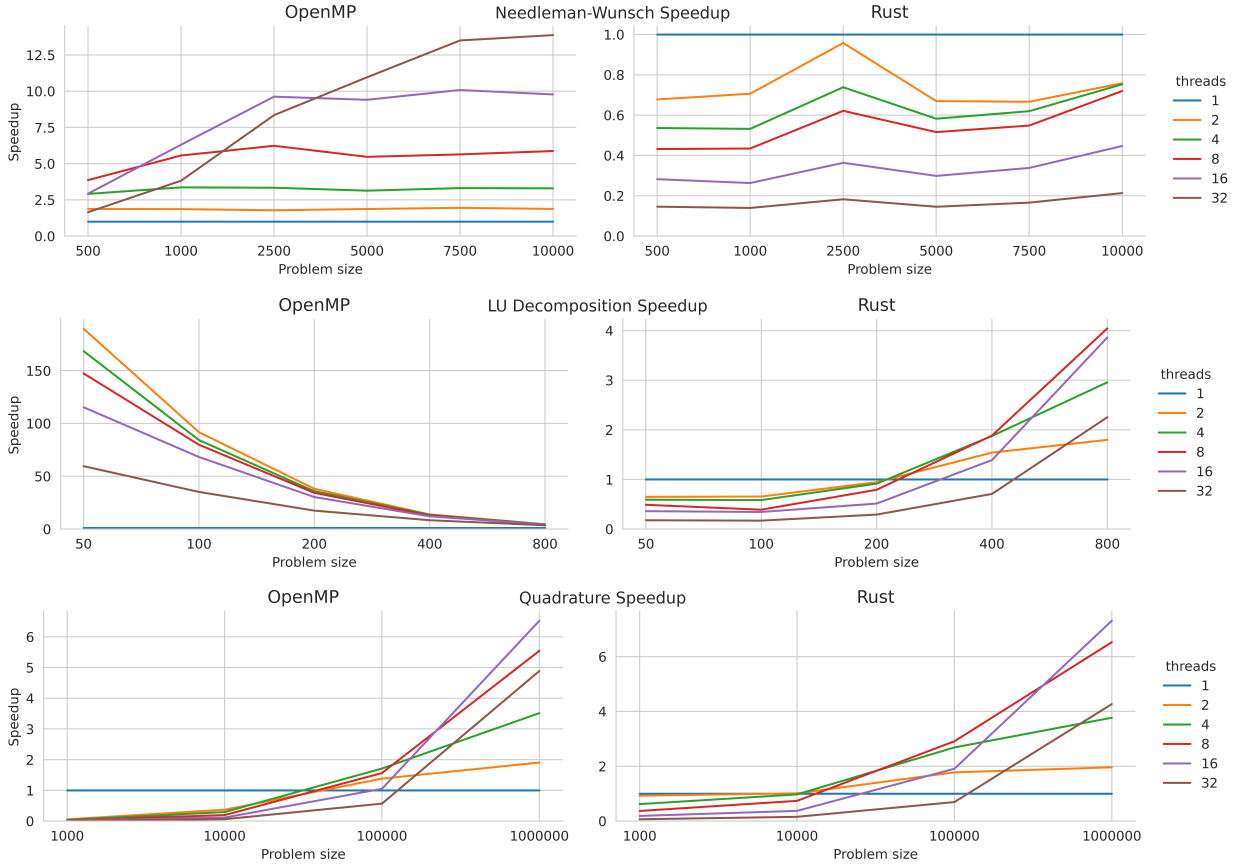
5.2.2 CPU BOUND



Here, the CPU Bounded benchmarks do indeed maintain a better degree of linear efficiency as the number of threads increase. This is as expected as each benchmark, being CPU bounded, does not require as much communication and synchronization between threads. Monte Carlo, as the most embarrassingly parallel benchmark, is the most strongly scalable benchmark. Comparing both languages, there is again a slight advantage to Rust over OpenMP at each thread count, in keeping with the high performant nature of Rust and the optimizations allowed by the Rust-specific Rayon library. OpenMP, with its applicability to C, C++, and Fortran, may forfeit some degree of performance in order to gain a more diverse set of language adoption. Ultimately, Rust seems to possess an edge in scalability.

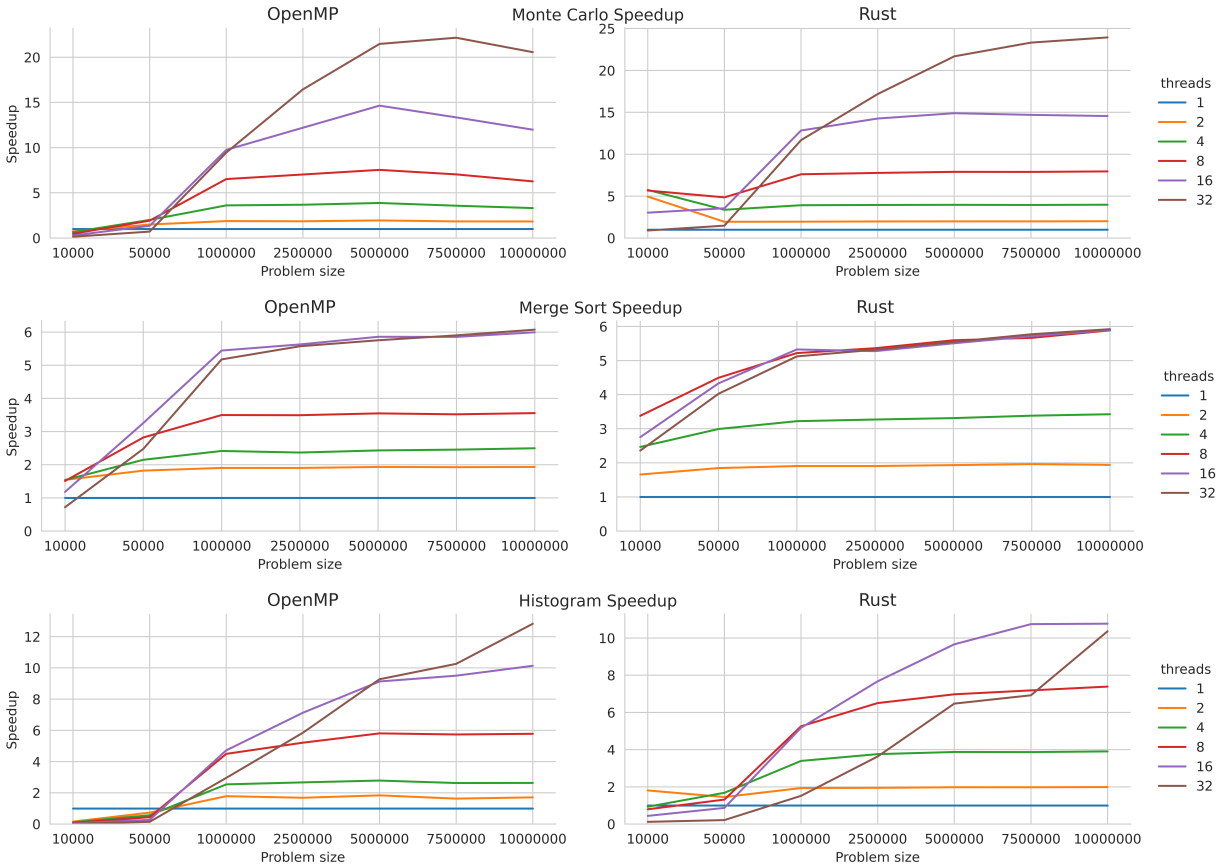
5.3 Performance

5.3.1 Memory Bound



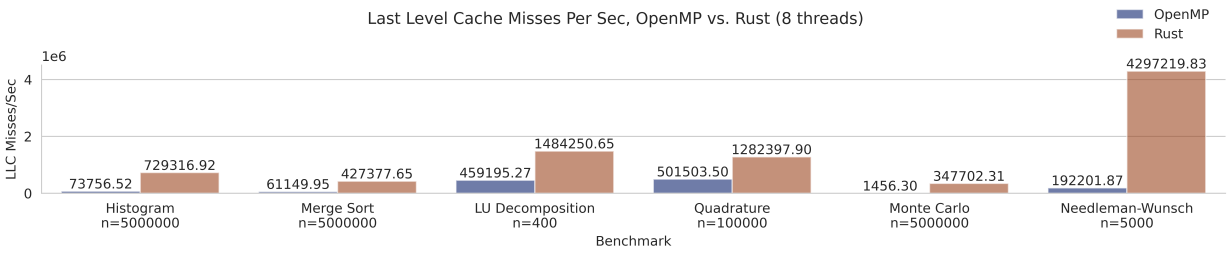
The memory bounded benchmarks again display relatively muted speed up as the thread count increases, though the increased problem size does benefit the speed up in most cases. LU Decomposition sees a relative decrease in speedup as the problem size increases, likely impacted by the degree of thread synchronization required at the end of each column modification. Because Rust and Rayon avoids a for loop implementation (necessitated by the iterator implementation), Rust does not see the performance degrade as problem size increases. Needleman-Wunsch sees poor performance again due to the specific implementation restrictions imposed as discussed in 5.2.1.
Overall, performance is better for Rust, as long as the algorithm implementation is relatively simple. As soon as complexity is introduced (as in the case of Needleman-Wunsch), implementation tradeoffs must be made that hinder and may actively reduce performance.

5.3.2 CPU Bound



The CPU bounded benchmarks all achieve a high degree of speedup as problem size increases, with no surprises. The difference between Rust and OpenMP here appears quite negligible, though the Histogram-Rust benchmark seems to struggle at the 32 thread count compared to the 16 thread count. This performance struggle likely comes from a combination of hardware limitations, increased overhead from synchronization and thread management with 32 threads, cache contention, and inefficiencies in the reduction abstraction specific to the Rust implementation. While Rayon provides these abstractions for safe and efficient parallelism, (in this case the "reduce" function), they can also potentially introduce overheads that become significant at higher thread counts, where the benefits of parallel execution are outweighed by the overhead costs.

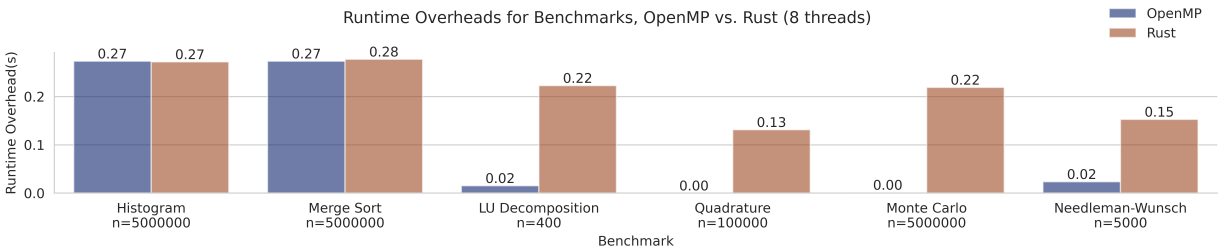
5.4 Memory Management



This graph examines the Last Level Cache (LLC) misses by benchmark compared on the two languages. Here, we see a clear trend of the Rust language seeing orders of magnitude higher LLC misses as compared to the OpenMP version. We examined this at a fixed thread count and problem size for each benchmark. We can see that a problem with higher memory accesses (like Needleman-Wunsch), sees a far greater disparity, whose impact can also be seen in the Performance and Scal-

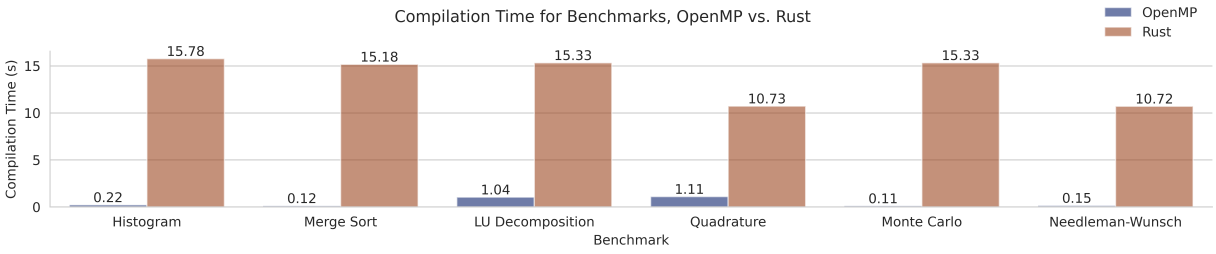
ability section. It's possible that the Rust runtime is still newer and not as fleshed out as C/C++, meaning that some attributes inherent to the benchmarks will lead to the same programs not being as cache friendly on Rust and therefore show a much higher LLC rate.

5.5 Runtime Overhead



Here, we see that some of the benchmarks have some level of difference in their overhead, with a general trend to having more overhead with the Rust language. Histogram and Merge Sort are even in their runtime, most likely due to the large amounts of randomness required for the histogram and merge sort example generation. which end up being the dominating factors and mostly hide the differences between the language.
For the other benchmarks, we see Rust having a clear lead in runtime overhead, especially for the Quadrature and Monte Carlo programs. This matches up with the general understanding that OpenMP and the C/C++ language in general are lighter weight than Rust. These results show that this is not only true in the language features that are checked at compile, but also the outside libraries needed like Rayon that deal with maneuvering around memory and parallelizing iterators. And this also most likely indicates that the Rust runtime itself to create/delete threads, allocate memory, etc. could be slower than the very mature C/C++ runtime. These findings indicate that if one is tackling a parallel problem, they should make sure the problem is sufficiently large enough to warrant using Rust in the first place.

5.6 Compilation Speed



When it comes to compilation, C/C++ heavily outperforms Rust. Even though the OpenMP library is linked and the pragmas are preprocessed, this is below 2 seconds for all benchmarks. The Monte Carlo benchmark takes only 0.11 seconds. On the other hand, Rust sees greater than 10 seconds for its compilation, with the Histogram taking a full 15.78 seconds to build. We can point to reasons like the fact that Rust has to compile and build together the Rayon library, include Rayon's own dependencies, and all of which are significantly slower with all of Rusts extra compiler checks. However, the biggest difference seems likely to come from the extensive memory-checking done by the Rust compiler. By default, the compiler carries an enormous workload in this regard, specifically to prevent errors from occurring. With no such construct in OpenMP, this is the likely the single biggest contributing factor that make Rust's compilation less light-weight compared to OpenMP.

5.7 Programmer Control

Rust with Rayon and OpenMP in C/C++ exemplify contrasting approaches to parallel programming, emphasizing different levels of programmer control.

5.7.1 Thread and Task Management

Rayon uses a global thread pool and handles task scheduling and execution automatically. For example, in the Merge Sort benchmark, the Rayon implementation utilizes **rayon::join** for parallel recursion:

```
if depth < max_depth {
    // Use rayon::join to parallelize recursive calls
    rayon::join(
        || merge_sort_parallel(left, left_buffer, depth + 1, max_depth),
        || merge_sort_parallel(right, right_buffer, depth + 1, max_depth),
    );
} else {
    // Sequential
    merge_sort_parallel(left, left_buffer, depth + 1, max_depth);
    merge_sort_parallel(right, right_buffer, depth + 1, max_depth);
}
```

In this implementation, Rayon automatically handles task creation, scheduling, and synchronization, making parallel recursion straightforward. The programmer does not have to manage threads or tasks. On the other hand, OpenMP requires explicit directives to manage threads and tasks. In the Merge Sort benchmark, the OpenMP implementation uses **#pragma omp task** and controls task creation depth to prevent excessive task spawning:

```
void mergeSortParallel(int arr[], int l, int r, int depth) {
    if (l < r) {
        int m = l + (r - l) / 2;

        // Create parallel omp TASKS
        #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);
    }
}
```

Here, the programmer must decide when to create tasks and manage their synchronization using **#pragma omp taskwait**. The "if" condition directive prevents task explosion by limiting task creation to a certain recursion depth (in this case 3). This level of explicit control requires careful planning to optimize performance and resource utilization but offers more control over these aspects to the programmer.

5.7.2 Synchronization and Data Sharing

Rust enforces data safety at compile time through its ownership and borrowing rules, reducing the need for manual synchronization. In the LU Decomposition benchmark, the Rayon implementation uses **par_iter_mut()** for safe concurrent mutations:

```
(&mut buffer[1..size]).par_iter_mut().enumerate().for_each(|(j, row_val)| {
if j < i {
    *row_val = 0.0;
} else {
    // Compute the value
}
});
```

Here, Rust's type system ensures that mutable access to data is safe, preventing data races. However, the programmer must rely on Rust's safety guarantees which minimizes synchronization concerns and the problems associated with concurrent data access.

OpenMP requires explicit handling of shared and private variables and synchronization mechanisms. In the Monte Carlo Pi Estimation benchmark, the OpenMP implementation must use reduction clauses and manages thread-local seeds for random number generation:

```
#pragma omp parallel
{
    unsigned int seed = omp_get_thread_num() + 3;

    #pragma omp for schedule(static) reduction(+:circle_count)
    for (long long i = 0; i < trial_count; i++ ){
        double coord_1 = (double)rand_r(&seed)/(double)RAND_MAX;
        double coord_2 = (double)rand_r(&seed)/(double)RAND_MAX;
        // Compute if point is inside the circle
        if (coord_1 * coord_1 + coord_2 * coord_2 <= RADIUS_SQUARED){
            circle_count++;
        }
    }
}
```

In this code, the programmer must ensure variables are correctly scoped and synchronized to prevent data races. The **reduction** clause safely aggregates the **circle_count** variable across threads, and the manual management of thread-local seeds ensures thread-safe random number generation.

5.7.3 Scheduling and Load Balancing

Rayon automatically balances the workload across threads using work-stealing algorithms. In the Needleman-Wunsch Algorithm, the Rayon implementation handles parallel execution over diagonals without manual scheduling:

```
for i in 1..input2_len + input1_len - 1 {
    let j_end = cmp::min(input1_len, i + 1) as usize;
    let j_start = cmp::max(1, (i as i32) - (input2_len as i32) + 2) as usize;

    (j_start..j_end)
        .into_par_iter()
        .enumerate()
        .for_each(|(_, j)| {
            // Compute scores
        });
}
```

In this implementation, the programmer does not need to specify scheduling; Rayon manages it internally. The parallel iterator distributes the work efficiently among threads, handling dependencies and workload variations.

OpenMP allows specifying scheduling strategies and chunk sizes to optimize performance. In the Monte Carlo Pi Estimation benchmark, the OpenMP implementation uses **schedule(static, 256)** to control iteration distribution among threads:

```
#pragma omp for schedule(static, 256) reduction(+:circle_count)
for (long long i = 0; i < trial_count; i++ ){
    // Random number generation and computation
}
```

By specifying **schedule(static, 256)**, the programmer controls how loop iterations are divided among threads, potentially improving cache performance and reducing synchronization overhead. Thus, this explicit control enables fine-tuning based on the characteristics of the problem at hand.

5.7.4 Programmer Control vs. Abstraction

The differences in programmer control between Rust with Rayon and OpenMP in C/C++ revolve around the trade-off between explicit control and abstraction. OpenMP offers fine-grained control over threads, tasks, synchronization, and scheduling, allowing performance optimization through detailed management. However, this increases complexity and the potential for errors, as programmers must manually manage memory, synchronization, and thread safety.

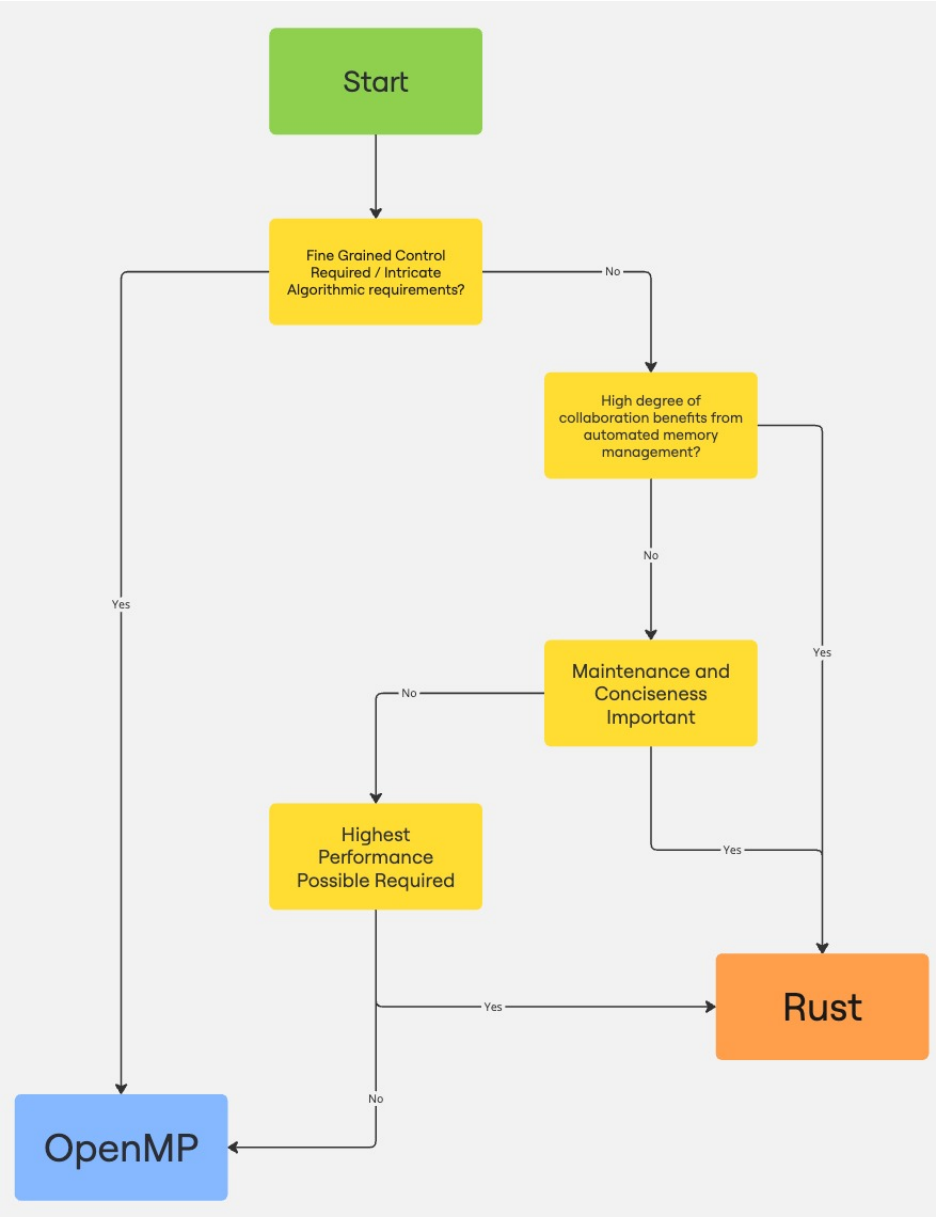
In contrast, Rust with Rayon provides safety guarantees through ownership and borrowing concepts, simplifying parallel programming with high-level abstractions. It automatically manages threads, synchronization, and memory, reducing the risk of concurrency-related bugs. The trade-off is less granular control over parallel execution and a steeper learning curve due to Rust's strict rules.

By analyzing these benchmarks, it can be concluded that the choice between OpenMP and Rust with Rayon depends on the project's requirements and the development team's expertise. OpenMP is suitable when fine-tuned performance optimization is critical, and the team is comfortable with low-level thread management. Rust with Rayon is ideal when safety and code maintainability are top priorities, and simple development of parallel code is desired.

6 Conclusion

- OpenMP provides more explicit control, which makes it suitable for experienced parallel programmers for more intricate and involved algorithms that need more granular abilities, especially more advanced algorithms in various domains. Rust with Rayon offers safer, more concise, and more maintainable parallel programming through various abstractions.
- Memory ownership mechanisms within Rust, it's primary distinguishing characteristic, enables safety but also impedes ease of parallel implementation due to the constraints it places on mutability.
- The speedup and scalability results show that the languages are comparable in their performance in parallel contexts, with Rust having a small edge in most cases where the Rust implementations aren't forced to be different due to Rust's memory constraints. This suggests that a programmer might find the factors surrounding programmability, degree of control, and maintenance ability as primary factors of consideration.

6.1 Flowchart



REFERENCES

[1] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," in IEEE Computational Science and Engineering, vol. 5, no. 1, pp. 46-55, Jan.-March 1998, doi: 10.1109/99.660313.

[2] Seeliger. Rust for HPC Programming

[3] Sudwoj. Rust programming language in the high-performance computing environment

[4] Javad Abdi, Gilead Posluns, Guozheng Zhang, Boxuan Wang, and Mark C. Jeffrey. 2024. When Is Parallelism Fearless and Zero-Cost with Rust? In Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '24). Association for Computing Machinery, New York, NY, USA, 27–40.

[5] Pfosi, Wood, and Zhou. A comparison of Concurrency in Rust and C

[6] Costanzo, Rucci, Naiouf, De Giusti. Performance vs Programming Effort bewteen Rust and C on Multicore Architectures: Case Study in N-Body

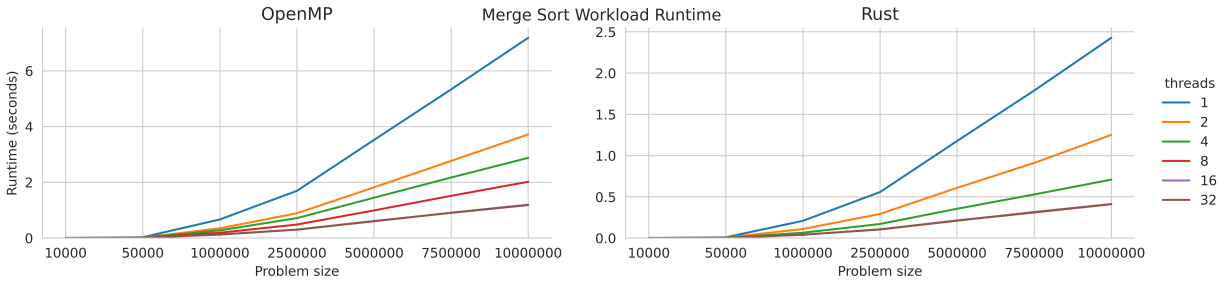
[7] Eghtesad, Adnan, et al."OpenMP and MPI implementations of an elasto-viscoplastic fast Fourier transform-based micromechanical solver for fast crystal plasticity modeling." Advances in Engineering Software 126 (2018): 46-60.

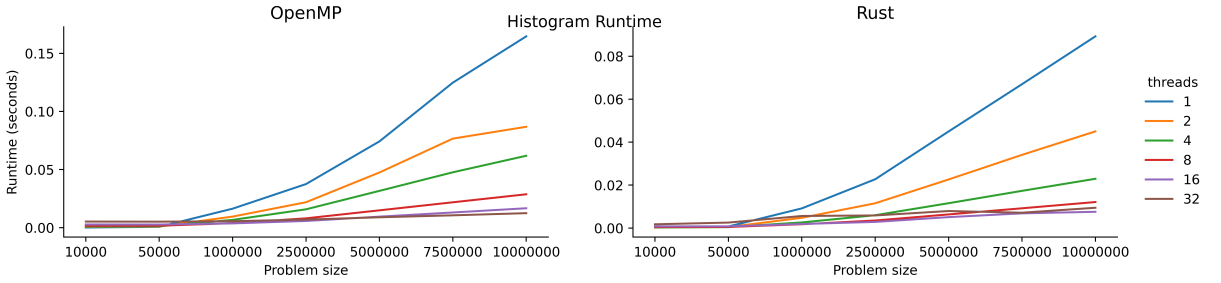
[8] Shi, Yan, and Chi Hou Chan. "An OpenMP parallelized multilevel Green's function interpolation method accelerated by fast Fourier transform technique." IEEE Transactions on antennas and propagation 60.7 (2012): 3305-3313.

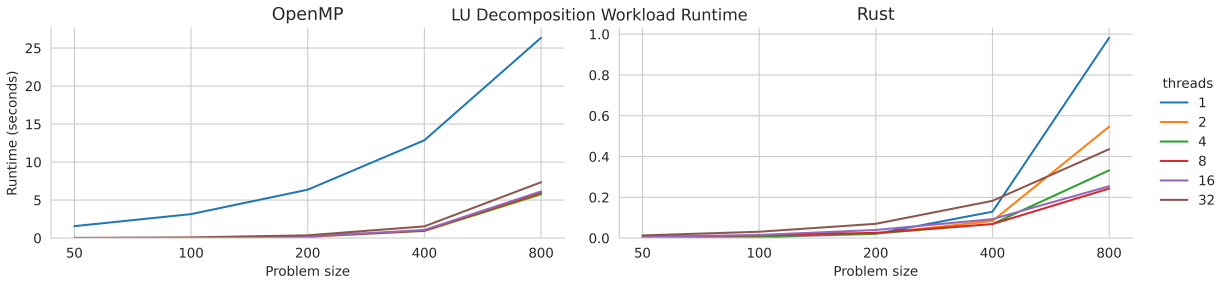
[9] F. Cantonnet, Y. Yao, M. Zahran and T. El-Ghazawi, "Productivity analysis of the UPC language," 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings., Santa Fe, NM, USA, 2004, pp. 254-, doi: 10.1109/IPDPS.2004.1303318

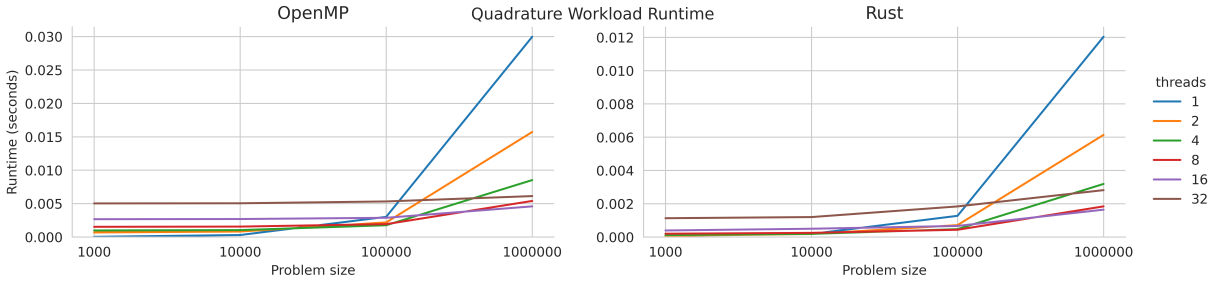
7 APPENDIX

7.1 RUNTIME GRAPHS

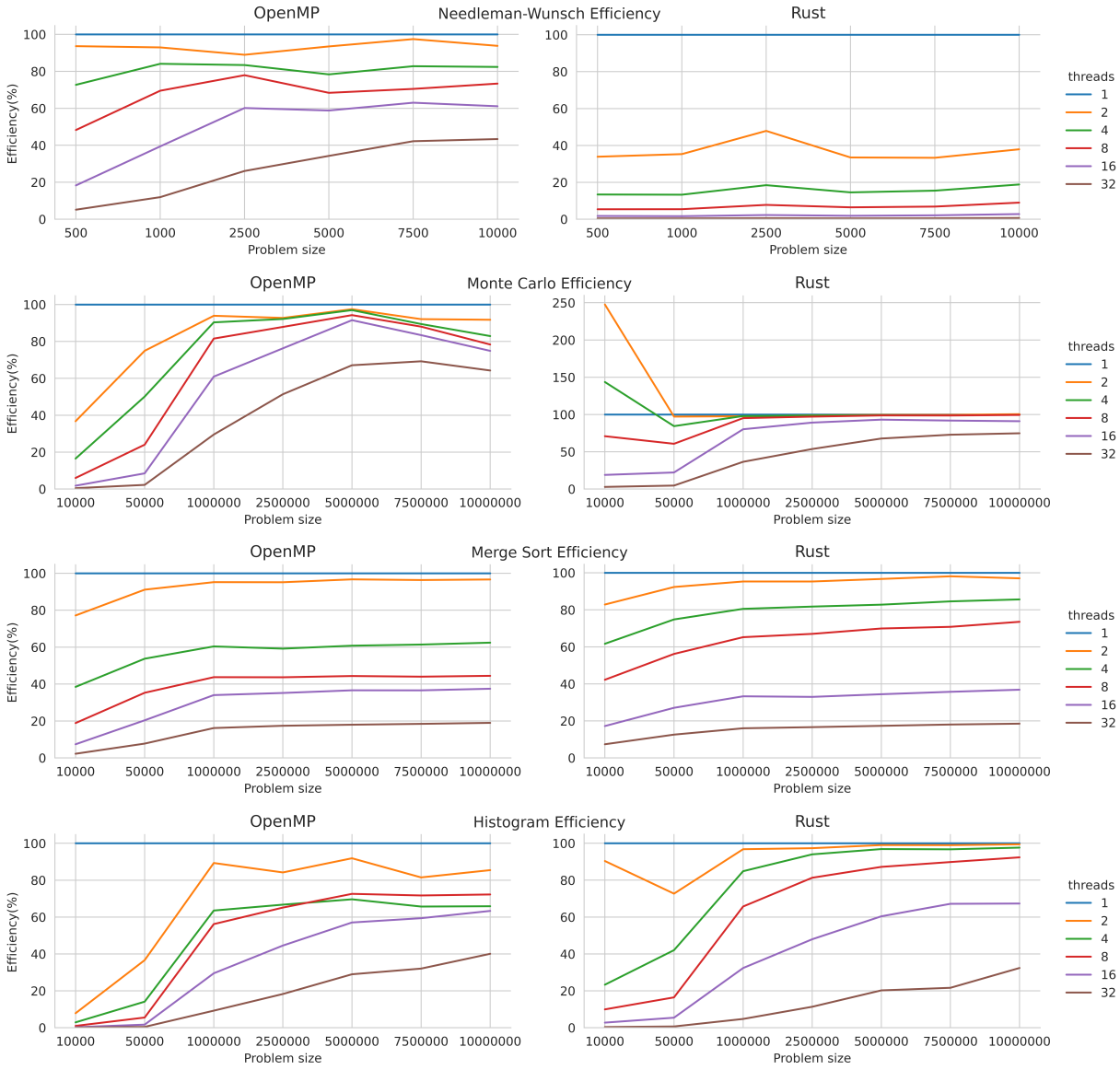

OpenMP
Merge Sort Workload Runtime
Rust
Runtime (seconds)
Problem size
threads
1
2
4
8
16
32

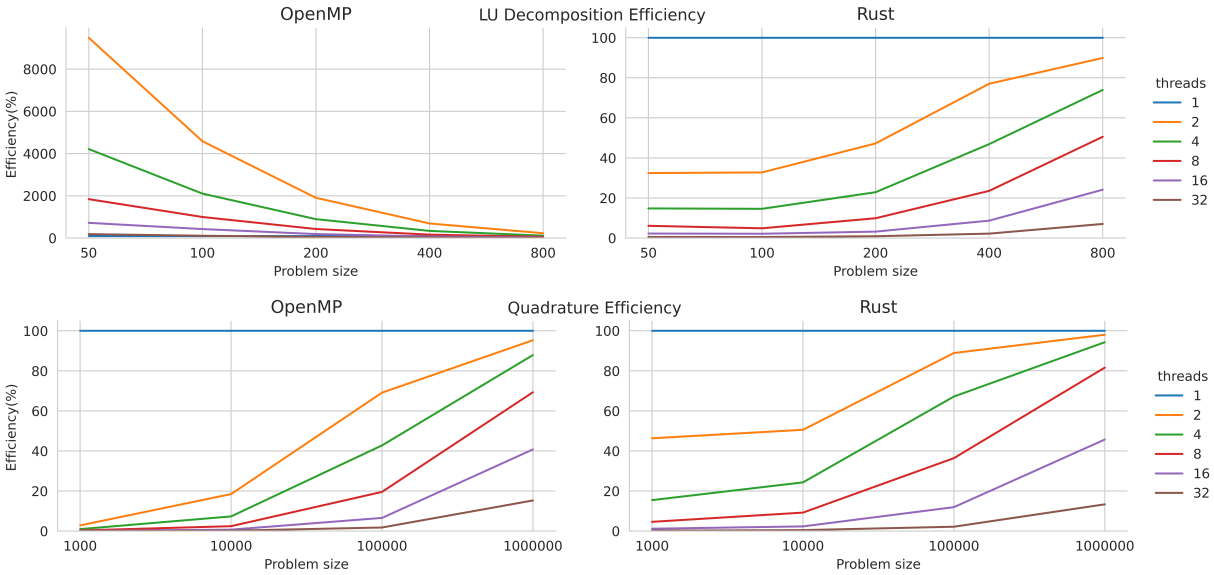

OpenMP
Histogram Runtime
Rust
Runtime (seconds)
Problem size
threads
1
2
4
8
16
32


OpenMP
LU Decomposition Workload Runtime
Rust
Runtime (seconds)
Problem size
threads
1
2
4
8
16
32


OpenMP
Quadrature Workload Runtime
Rust
Runtime (seconds)
Problem size
threads
1
2
4
8
16
32

7.2 Efficiency Graphs




OpenMP
LU Decomposition Efficiency
Rust
Efficiency(%)
Problem size
threads
1
2
4
8
16
32
OpenMP
Quadrature Efficiency
Rust
Efficiency(%)
Problem size
threads
1
2
4
8
16
32