Comparing OpenMP and Rust

Zhouyuan Chen
zc2952@nyu.edu
New York University

Qian Zhang
qz2570@nyu.edu
New York University

Tina Su
cs7483@nyu.edu
New York University

Hao Li
hl5262@nyu.edu
New York University

***Abstract*—This report presents a comparative study of parallel programming using OpenMP and Rust through the implementation of eight different algorithms. The primary goal is to evaluate the performance and usability differences between these two approaches. OpenMP, a well-established parallel programming model for shared-memory architectures, is compared against Rust, a modern systems programming language known for its emphasis on safety and concurrency. For each algorithm, we implemented using both techniques, and then analyzed their performance under identical conditions, focusing on execution time, scalability, and resource utilization. Additionally, the report examines the development experience, including ease of use, code complexity, and debugging challenges. The results highlight the trade-offs between these two approaches, offering insights into their suitability for different parallel computing tasks.**

I. Introduction

Parallel computing plays a crucial role in enhancing the performance of computationally intensive tasks across a variety of fields, from scientific simulations to machine learning [1]. By utilizing multiple processors or cores, parallel algorithms can significantly reduce execution time and enable the efficient handling of large datasets. Among the various approaches to parallel programming, OpenMP [2] and Rust [3] stand out as two prominent techniques with distinct strengths and trade-offs. OpenMP, a widely-used API for parallel programming on shared-memory systems, offers a straightforward approach to parallelization via compiler directives. Rust, a modern systems programming language, provides a powerful concurrency model with a strong emphasis on memory safety and performance.

This report compares the performance and usability of OpenMP and Rust by implementing eight common algorithms, each representing different aspects of parallel computing. The algorithms studied include Monte Carlo Simulation, CNN Forward Propagation, Merge Sort, Prefix Sum, N-Body Problem, K-Nearest Neighbors, Reduce Problem, and Matrix Multiplication.

For each algorithm, we implement parallel versions using both OpenMP and Rust, measuring key performance metrics such as execution time, scalability, and resource utilization. Additionally, the report examines the ease of implementation, code complexity, and concurrency model used by each framework, providing a comprehensive analysis of their suitability for different types of parallel tasks.

The purpose of this study is to evaluate how OpenMP and Rust perform in parallelizing diverse algorithms and to highlight the advantages and limitations of each approach. By comparing these two paradigms, we aim to offer insights into their practical applications and guide developers in choosing the appropriate parallelization tool based on the requirements of their specific projects.

For the complete source code, visit our GitHub repository: github.com/Zhouyuan-Chen/multicore.

II. Preparation Work

In this section, we will first provide the necessary background for the algorithms compared in our benchmark and then briefly discuss the fundamentals of OpenMP and Rust, highlighting key features relevant to the parallelization tasks.

Introduction to algorithm

A. Monte Carlo

Monte Carlo simulation [4] is a stochastic method commonly used to approximate π by randomly sampling points in a square and counting how many fall within a quarter circle. Due to its inherently parallel nature, it is well-suited for parallel programming techniques. We compute the π with the formula:

$$\pi \approx 4 \times \frac{\textit{Accept Samplings}}{\textit{Samplings}}$$

B. CNN Foward Propagation

Convolutional Neural Networks (CNNs) [5] are widely used in computer vision tasks due to their ability to learn spatial hierarchies of features. Forward propagation is the process of input data passing through the network layer by layer to generate an output, typically for classification or regression tasks. Each convolution operation applies a kernel to extract features from the input, and since these computations are independent of different kernels, forward propagation is highly parallelizable.

The convolution operation applies a kernel W to an input feature map X, producing an output feature map Y. At position (i, j), the convolution is computed as:

$$Y(i,j) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} W(m,n) \cdot X(i+m, j+n) + b,$$

Where M and N are the height and width of the kernel, and b is the bias term. In our implementation, we assume the kernel's width and height are the same.

C. Merge Sort

Merge Sort is a divide-and-conquer algorithm that recursively splits an array into two subarrays, sorts them, and then merges them back to fully sort the array. We implement parallelization by dividing the recursive calls into separate sections. The algorithm works as follows:

1) Recursively divide the array into two halves.
2) Sort the two halves in parallel threads until a predefined depth.
3) Merge the sorted halves.

D. Prefix Sum

Prefix Sum is an algorithm for computing the cumulative sum of an array. Given an array A of size n, the prefix sum array S is defined as:

$$S[i] = \sum_{k=0}^{i} A[k] \quad \text{for} \quad 0 \leq i < n$$

We implemented the Hillis-Steele Scan Algorithm [6], which iteratively updates the prefix sum in logarithmic steps using multiple threads. The algorithm works as follows:

1) Initialize the prefix sum array S with the original array A.
2) For each step size step $= 1, 2, 4, 8, \ldots$
    a) Create a helper array to store the current state of the prefix sum.
    b) Using parallel threads, for each index $i \geq$ step, update $S[i]$ as $S[i] + helper[i -$ step$]$.
3) Repeat the above steps until the step size exceeds the array length.

E. N-Body Problem

The N-body problem is a classical problem in physics that involves predicting the individual motions of a group of celestial objects interacting with each other gravitationally. The challenge lies in computing the forces between all pairs of bodies, which results in an O(N^2) computational complexity [7], [8]. Parallel computing techniques can significantly optimize these computations, making simulations feasible for large N [9].
In OpenMP, we write a regular simulation Loop to do force calculation: For each particle, compute the acceleration due to all other particles. This is parallelized using $\#pragma\ omp\ parallel\ for$. After updating velocities, update the positions of all particles in parallel. In Rust, we need to first clone particles to safely iterate in parallel without mutable aliasing. Note that this approach is memory-intensive for large N and can be optimized. For force calculation, we use $par\_iter()$ to compute the acceleration for each particle in parallel. We simultaneously update all particles using $par\_iter\_mut()$ and the computed accelerations.

F. K-Nearest Neighbors

K-Nearest Neighbors (KNN) [10] is a classic algorithm in machine learning and data mining that is highly amenable to parallelization. Leveraging parallel computing can significantly accelerate the computation, especially when dealing with large datasets. There are two phases for KNN. In the training phase, KNN has no explicit training phase. It simply stores the training dataset. In the prediction phase, for a given query point, we compute the distance between the query point and all points in the training dataset. Then we identify the k closest points based on the computed distances. To Classify, we assign the class most common among the k neighbours. Finally do the regression by computing the average of the k neighbors' values.
The primary bottleneck in KNN is the distance computation between the query points and all training points. Since each query's distance computations are independent, parallelizing over the queries or over the training points is straightforward. If we do data parallelism over queries, we could assign different queries to different threads. If we do data parallelism over training points, for each query, we compute distances to training points in parallel.

G. Reduce Problem

The reduction algorithm is a fundamental technique in parallel computing used to aggregate data by applying a specific operation [11]. It's often used for operations like summing all elements in an array, finding the minimum or maximum value, or computing the product of all elements. The algorithm takes an array of values and reduces them to a single value using an associative binary operation, like addition, multiplication, or logical operations.
In OpenMP, Reduce operations are implemented using the reduction clause, where the compiler automatically creates thread-private local variables and combines them after computation, simplifying the complexities of multithreaded programming. In Rust, libraries like rayon enable Reduce operations through parallel iterators and higher-order functions. Rust's ownership and type systems ensure thread safety and prevent data races, while providing high-performance parallel computation capabilities.

H. Matrix Multiplication

Matrix multiplication is a fundamental operation in linear algebra where two matrices are combined to produce a third matrix, following specific arithmetic rules involving the sum of products of their rows and columns. Parallelizing matrix multiplication [12] can significantly improve performance for large datasets. In OpenMP, this is achieved by distributing loop iterations across multiple threads using directives like #pragma omp parallel for , often with the collapse clause to parallelize nested loops, which allows efficient utilization of multicore processors while managing thread synchronization. In Rust, parallel matrix multiplication can be implemented using libraries like rayon, which offer parallel iterators and thread pools. Rust's ownership model and type system ensure thread safety and prevent data races, enabling high-performance parallel computations are both safe and efficient.

Introduction to OpenMP and Rust

OpenMP: OpenMP (Open Multi-Processing) is an API that facilitates parallel programming in shared-memory systems, allowing developers to parallelize their applications with minimal modifications to the original code [13]. Originally developed in the early 1990s, OpenMP provides a set of compiler directives, runtime routines, and environment variables for programming parallel tasks in C, C++, and Fortran [14]. The core idea behind OpenMP is to provide an easy way to implement parallelism by introducing simple annotations in the code, which the compiler then uses to distribute work across multiple threads [15]. Its key feature is the ability to add parallelism by inserting simple compiler directives like #pragma into the existing code, allowing for quick scalability on multi-core systems. However, this simplicity also requires developers to carefully manage parallel operations and ensure they are handling potential issues, such as data races and synchronization, appropriately.

Rust: Rust is a systems programming language designed with a focus on safety, performance, and concurrency [16]. Developed by Mozilla Research, it was first introduced in 2010, with the first stable version released in 2015 [17]. Rust was created to address the common pitfalls of traditional programming languages like C and C++, particularly around memory safety and concurrency. Its primary innovation is the ownership model, which ensures memory safety without a garbage collector, making it suitable for performance-critical and concurrent systems programming [18].

III. Environment

Machine: We ran our code on crunchy2, which is equipped with four AMD Opteron 6272 processors (2.1 GHz, 64 cores), 256 GB of memory, and runs Red Hat Enterprise Linux 9.

Compiler Version and Library: Our GCC version is 9.5.0, and for Rust, we are using rust 1.82.0 along with the Rayon library [3] to parallelize the programs.

IV. Methodology

A. Performance

In this section, we evaluate the parallel performance of our algorithms under large problem sizes using various metrics, providing a comprehensive comparison of serial and parallel execution across various dimensions.

Metrics Description:

- **OMP Baseline & Rust Baseline:** The response time of the sequential implementation of the algorithm.
- **OMP Time & Rust Time:** The response time of the parallel implementations coded in OpenMP and Rust, respectively.
- **OMP Speedup & Rust Speedup:** Speedup [19] is a metric used in parallel computing to quantify the performance improvement gained by parallelizing a computational task. They are computed as:

$$\text{Speedup} = \frac{\text{Baseline Time}}{\text{Parallel Time}}$$

- **OMP Memory Usage & Rust Memory Usage:** These metrics track the peak memory consumption during the execution of the parallel algorithms. The measurements are obtained using the `/usr/bin/time -v` command provided by GNU.
- **Compilation Time:** This metric measures the time required to compile the source code. It is determined using the `time` command available on Linux, with the real time taken as the final metric.

B. Scalability

Scalability [20] is a crucial concept in parallel computing that describes how effectively a system can utilize additional computational resources to handle increasing workloads. It is important because it determines a program's ability to maintain performance improvements as problem sizes expand and more processing threads are employed. To represent scalability in our study, we measure the speedup achieved by implementing eight different algorithms across various problem sizes and thread counts. This approach allows us to evaluate how execution times decrease relative to the number of threads and the complexity of the problem. In the subsequent Results section, we will analyze the collected data on scalability, identifying representative examples that highlight the comparative strengths and limitations of OpenMP and Rust in scaling parallel computations. We made the tables for measuring efficiency [21] with the formula:

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Thread Number}}$$

C. Overhead

In our study, we evaluated the overhead by analyzing the cost associated with thread allocation. To get the results from these two tools, we measured the average cost by creating threads, recording the time, and averaging the total time. For OpenMP, we used:

```
#pragma omp parallel num_threads(N)
```

For the OpenMP, we used:

```
data.par_iter().for_each(| &N | {func(N)})
```

Where N is the thread number.

D. Programmability

Programmability in the context of parallel algorithms refers to the ease and flexibility with which developers can design, implement, and manage concurrent code that executes across multiple processing units [22]. When utilizing parallel programming frameworks like OpenMP or modern programming languages like Rust, programmability encompasses various factors, including coding difficulty, abstraction levels, and the degree of control programmers have over thread management and execution. Choosing between OpenMP and Rust for parallel programming depends on the specific requirements of the project, the desired level of control, the existing code, and the developer's familiarity with the respective tools and paradigms.

In this case, the most straightforward way to measure this is by analyzing the lines of code (LOC) [22] in each program. LOC has historically been one of the earliest and most commonly used software metrics. Additionally, we considered the number of characters in the code, as it provides insight into the flexibility and complexity of the tools. We wrote the sequential code version for two OpenMP and Rust respectively, and compared the additional efforts to parallelize the program with these two tolls.

Moreover, we presented an illustrative example demonstrating that OpenMP tends to result in a higher likelihood of implementation errors.

V. Result

A. Performance

To highlight the performance differences, we selected specific input sizes for each program and summarized the results in the table below:

Metric	OMP Baseline	Rust Baseline
Monte Carlo	0.075s	0.008s
CNN	13.23s	3.667s
Merge Sort	178.399s	29.146s
Prefix Sum	44.88s	8.11s
Reduce	0.016s	0.0094s
MM	13.647s	2.388s
N Body	87.468s	16.3517s
KNN	87.1124s	18.1849s

TABLE I: Baseline execution time for a single thread.

Metric	Problem Size (N)	Threads number (t)
Monte Carlo	1,000,000	16
CNN	1,000,000$\times 1,000,000$	16
Merge Sort	100,000,000	8
Prefix Sum	100,000,000	16
N Body	1,000,000,000	16
KNN	100,000,000	16
Reduce	1,000,000	32
Matrix Multiplication	1,000 $\times 1,000 \times 1,000$	64

TABLE II: Problem Size and threads number for measuring parallelism performance.

Metric	OMP Parallel Time	Rust Parallel Time	OMP Speedup	Rust Speedup
Monte Carlo	0.011s	0.004s	6.82	13.33
CNN	0.86s	0.169s	15.38	21.69
Merge Sort	14.746s	2.544s	12.098	11.457
Prefix Sum	5.22s	1.63s	8.60	4.98
Reduce	0.009s	0.00484s	1.7	1.9
MM	0.4247s	0.1935s	32	13.3
N Body	5.5876s	1.2829s	15.65	12.75
KNN	5.571s	2.2144s	15.64	8.21

TABLE III: Parallel program execution time and speedups.

Metric	OMP Parallel Memory	Rust Parallel Memory
Monte Carlo	5340KB	1536KB
CNN	786752KB	787628KB
Merge Sort	1003848KB	1044164KB
Prefix Sum	1174652KB	1172332KB
Reduce	1536KB	10008KB
MM	17412KB	37716KB
N Body	3752KB	23552KB
KNN	67532KB	106960KB

TABLE IV: Parallel program maximum memory usage.

For runtime performance, Rust programs outperform those using OpenMP due to Rust's superior memory management strategy. Rust has no runtime or garbage collection overhead, and memory is managed statically at compile time. Once the program is compiled, there is minimal overhead during the execution of the parallel code.

This also leads to another interesting observation: the parallel memory usage of Rust is larger than that of OpenMP for most programs. This aligns with the fact that, to ensure thread safety, Rust manages memory more conservatively, which often results in higher memory consumption.

From the tables in the next section, it is evident that parallel-friendly algorithms, such as Matrix Multiplication, Monte Carlo, and CNN, perform well and exhibit strong scalability in both OpenMP and Rust.

However, for runtime performance, programs written in Rust tend to be faster than those using OpenMP. This is largely due to Rust's memory safety model, which avoids a garbage collector and enforces strict ownership and borrowing rules at compile time. These features reduce runtime overhead, making Rust programs more efficient. In contrast, OpenMP relies on the programmer to manage memory, which can lead to performance bottlenecks if memory access is not properly optimized.

B. Scalability

1) Speedup Table: We created two tables, one for OpenMP and one for Rust, to explore the relationships between each problem, input size, and thread count. For each specific problem, we experimented with varying input sizes and thread counts. The input sizes were chosen based on the characteristics of each problem. The tables V and VI present the resulting speedup values and response time for these configurations.

2) Efficiency Table: Efficiency is calculated as the speedup divided by the number of threads (t). Higher efficiency indicates better scalability of the parallel algorithm. The results are in Table VIII and IX.

3) Overall Differences:

- OpenMP:
  - Superior Scalability: Consistently demonstrates high speedup and efficiency as thread counts increase.
  - High Efficiency: Maintains near-linear speedup up to higher thread counts.
- Rayon (Rust):

Algorithm	**Input Size (N)**	**t = 1**	**t = 4**	**t = 8**	**t = 16**
Monte Carlo	100	1 (0.006)	0.85 (0.007)	0.85 (0.007)	0.75 (0.008)
Monte Carlo	100,000	1 (0.013)	1.3 (0.010)	1.625 (0.08)	1.44 (0.009)
Monte Carlo	1,000,000	1 (0.075)	3.125 (0.024)	4.69 (0.016)	6.81 (0.011)
CNN	100$\times$100	1 (0.001)	1 (0.001)	0.83 (0.0012)	0.67 (0.0015)
CNN	1,000$\times$1, 000	1 (0.13)	3.17 (0.041)	8.67 (0.015)	14.44 (0.009)
CNN	10,000$\times$10, 000	1 (13.23)	4.99 (2.65)	9.59 (1.38)	15.38 (0.86)
Reduce	1,000,000	1 (0.0164)	1.4 (0.0117)	1.74 (0.0094)	1.74 (0.0094)
Reduce	1,000,000,000	1 (10.70)	4.8 (2.23)	8.4 (1.27)	14.45 (0.74)
Matrix Multiplication (MM)	$1000 \times 1000 \times 100$	1 (1.159)	3.9 (0.2972)	7.8 (0.1485)	15.3 (0.0758)
Matrix Multiplication (MM)	$10,000 \times 10,000 \times 10,000$	1 (14.02)	3.6 (3.857)	6.9 (2.040)	12.4 (1.129)
Prefix Sum	1,000,000	1 (0.322)	3.46 (0.093)	6.85 (0.047)	11.93 (0.027)
Prefix Sum	10,000,000	1 (3.82)	3.90 (0.98)	7.64 (0.50)	14.15 (0.27)
Prefix Sum	100,000,000	1 (44.88)	3.88 (11.56)	7.85 (5.72)	8.60 (5.22)
Merge Sort	1,000,000	1 (1.469)	3.99 (0.377)	6.739 (0.218)	8.957 (0.164)
Merge Sort	10,000,000	1 (16.358)	3.99 (4.120)	7.091 (2.307)	11.618 (1.408)
Merge Sort	100,000,000	1 (178.399)	3.928 (45.422)	7.246 (24.619)	12.098 (14.746)
N Body	1,000,000,000	1 (87.468)	3.98 (21.9638)	7.92 (11.0377)	15.65 (5.5876)
KNN	100,000,000	1 (87.1124)	3.96 (21.9951)	7.89 (11.0359)	15.64 (5.571)

TABLE V: Speedup metrics for OMP-based various algorithms, showing speedup (response time in seconds) with different thread counts.

Algorithm	**Input Size (N)**	**t = 1**	**t = 4**	**t = 8**	**t = 16**
Monte Carlo	100	1 (0.003s)	0.75 (0.004)	0.6 (0.005)	0.06 (0.05)
Monte Carlo	100,000	1 (0.004)	1 (0.004)	1.333 (0.003)	1.333 (0.003)
Monte Carlo	1,000,000	1 (0.008)	2 (0.004)	2.667 (0.003)	13.33 (0.0006)
CNN	100$\times$100	1 (0.0003s)	2 (0.00015s)	1.5 (0.0002)	1 (0.0003)
CNN	1,000$\times$1, 000	1 (0.024)	4 (0.006)	8 (0.003)	12 (0.002)
CNN	10,000$\times$10, 000	1 (3.667)	3.602 (1.018)	5.99 (0.612)	21.69 (0.169)
Reduce	1,000,000	1 (0.007)	2.68 (0.00261)	3.01 (0.00232)	2.91 (0.00240)
Reduce	1,000,000,000	1 (8.349)	5.78 (1.443)	7.68 (1.086)	13.82 (0.604)
Matrix Multiplication	1,000$\times$1, 000 $\times$ 100	1 (0.2159)	3.9 (0.0553)	7.5 (0.0287)	13.4 (0.0161)
Matrix Multiplication	1,000$\times$1, 000 $\times$ 1, 000	1 (2.31)	2.8 (0.8242)	6.2 (0.3703)	8 (0.288)
Prefix Sum	1,000,000	1 (0.0833)	3.74 (0.0223)	3.43 (0.0243)	2.95 (0.0282)
Prefix Sum	10,000,000	1 (0.858)	2.50 (0.344)	4.74 (0.181)	4.01 (0.214)
Prefix Sum	100,000,000	1 (8.11)	3.50 (2.32)	3.44 (2.36)	4.98 (1.63)
Merge Sort	1,000,000	1 (0.255)	3.307 (0.0771)	5.183 (0.0492)	6.589 (0.0387)
Merge Sort	10,000,000	1 (2.834)	3.953 (0.717)	7.015 (0.404)	9.639 (0.294)
Merge Sort	100,000,000	1 (29.146)	3.827 (7.617)	6.108 (4.772)	11.457 (2.544)
N Body	1,000,000,000	1 (16.3517)	3.92 (4.1753)	7.47 (2.188)	12.75 (1.2829)
KNN	100,000,000	1 (18.1849)	3.72 (4.8947)	4.75 (3.8284)	8.21 (2.2144)

TABLE VI: Speedup metrics for Rust-based parallel algorithms, showing speedup (response time in seconds) with different thread counts.

Algorithm	**Input Size (N)**	**t = 1**	**t = 4**	**t = 8**	**t = 16**	**t=32**	**t=64**
Merge Sort-Rust	300,000,000	1 (90.888)	3.877 (23.441)	6.994 (12.985)	11.960 (7.599)	17.690 (5.141)	21.900 (4.150)
Merge Sort-OMP	300,000,000	1 (563.547)	3.921 (141.925)	7.780 (76.986)	12.280 (45.892)	17.920 (31.448)	22.527 (25.016)

TABLE VII: Scalability metrics for extra large input size. t is thread number.

  - Moderate Scalability: Shows initial speedup but struggles to maintain efficiency beyond a few threads.
  - Decreasing Efficiency: Efficiency drops significantly at higher thread counts with small problem size, indicating limited scalability.

The efficiency drop in Rust with higher thread numbers is likely to be attributed to its highly efficient baseline for some small input size cases. Given large input size we observe that its efficiency experiences a similar decline compared to the OpenMP version, as seen in the following case: Table VII of Merge Sort.

This analysis demonstrates that as the problem size increases, both Rust and OpenMP achieve significant parallelization gains, following a linear pattern within a limited range of thread counts (< 16). That is their scalability are both very well given the large input size. However, as the number of threads increases to 32, efficiency begins to drop drastically. This decline is primarily attributed to the overhead introduced by managing a larger number of threads, even though not all available cores are fully utilized.

Further explanation of this is provided in the next section. For smaller problem sizes, OpenMP proves to be a better choice for achieving higher efficiency through parallelization, showing better scalability.

Nevertheless, Rust demonstrates a clear advantage over OpenMP in terms of response time. Using Merge Sort as an example, the reasons for this performance difference will be discussed in detail.

4) Specific Algorithm Analysis: Merge Sort

a) Efficient Work-Stealing Scheduler: Rayon employs a dynamic work-stealing scheduler [23] that balances load effectively among threads, which is ideal for recursive algorithms like merge sort, where task sizes vary.

b) Low Overhead in Task Management: Rayon has minimal overhead for creating and synchronizing tasks, with

Algorithm	**Input Size (N)**	**t = 1**	**t = 4**	**t = 8**	**t = 16**
Monte Carlo	100	1.000	0.2125	0.1064	0.0470
Monte Carlo	100,000	1.000	0.3250	0.2031	0.0900
Monte Carlo	1,000,000	1.000	0.7813	0.5863	0.8331
CNN	$100{\times}100$	1.000	1.000	0.1038	0.0419
CNN	$1{,}000\ \times 1,000$	1.000	0.7925	1.0840	0.9025
CNN	$10{,}000\ \times 10,000$	1.000	1.2475	1.1989	0.9613
Reduce	1,000,000	1.000	0.3500	0.2170	0.1080
Reduce	1,000,000,000	1.000	1.2000	1.0500	0.9030
Matrix Multiplication	1,000x1,000x100	1.000	0.9750	0.9750	0.9560
Matrix Multiplication	10,000x10,000x10,000	1.000	0.9000	0.8625	0.7750
Prefix Sum	1,000,000	1.000	0.8650	0.8560	0.7460
Prefix Sum	10,000,000	1.000	0.9750	0.9550	0.8840
Prefix Sum	100,000,000	1.000	0.9700	0.9810	0.5380
Merge Sort	1,000,000	1.000	0.9980	0.8420	0.6000
Merge Sort	10,000,000	1.000	0.9980	0.8860	0.7260
Merge Sort	100,000,000	1.000	0.982	0.906	0.756
N Body	10,000,000	1.000	1.005	0.98	0.95
N Body	100,000,000	1.000	0.99	0.99	0.98
N Body	1,000,000,000	1.000	0.995	0.99	0.978
KNN	1,000,000	1.000	1.013	1.005	0.95
KNN	10,000,000	1.000	0.99	0.99	0.99
KNN	100,000,000	1.000	0.99	0.986	0.9775

TABLE VIII: Efficiency of OpenMP Implementations for Various Algorithms. t is thread number

Algorithm	**Input Size (N)**	**t = 1**	**t = 4**	**t = 8**	**t = 16**
Monte Carlo	100	1.000	0.1875	0.0750	0.0600
Monte Carlo	100,000	1.000	1.0000	0.3325	0.3325
Monte Carlo	1,000,000	1.000	0.5000	0.3375	0.1250
CNN	$100{\times}100$	1.000	0.5000	0.3750	0.0625
CNN	$1{,}000\ \times 1,000$	1.000	1.0000	1.0000	0.7500
CNN	$10{,}000\ \times 10,000$	1.000	0.9005	1.4675	1.3556
Reduce	1,000,000	1.000	0.6700	0.3760	0.1820
Reduce	1,000,000,000	1.000	1.4450	0.9600	0.8630
Matrix Multiplication	$1{,}000{\times}1,000 \times 100$	1.000	0.9750	0.9375	0.8370
Matrix Multiplication	$1{,}000{\times}1,000 \times 1,000$	1.000	0.7000	0.7750	0.5000
Prefix Sum	1,000,000	1.000	0.9350	0.4290	0.1840
Prefix Sum	10,000,000	1.000	0.6250	0.5930	0.2510
Prefix Sum	100,000,000	1.000	0.8750	0.4300	0.3110
Merge Sort	1,000,000	1.000	0.8270	0.7280	0.4120
Merge Sort	10,000,000	1.000	0.9880	0.8770	0.6020
Merge Sort	100,000,000	1.000	0.9570	0.7640	0.7160
N Body	10,000,000	1.000	0.975	0.92	0.73
N Body	100,000,000	1.000	0.976	0.94	0.77
N Body	1,000,000,000	1.000	0.98	0.933	0.7969
KNN	1,000,000	1.000	0.82	0.8575	0.53
KNN	10,000,000	1.000	0.99	0.9675	0.716
KNN	100,000,000	1.000	0.93	0.5937	0.5131

TABLE IX: Efficiency of Rust Implementations for Various Algorithms.

detailed explaination in next section.

c) Optimized Recursive Parallelism: Rayon's `join` function optimally handles recursive parallel calls.

C. Overhead

We brutally create the thread with both Rust and OpenMP and compute the average thread creation time for OpenMP and Rust.

	Thread Creation	Total Time	Average Time
OpenMP	1000	0.104s	0.104ms
Rust	1000	0.019s	0.019ms

TABLE X: Overhead Measurements.

From the data table X above, the thread allocation cost in Rust is less expensive than in OpenMP. This is because the Rayon library uses a thread pool technique. With this approach, even when users create many threads, Rayon can reuse previously allocated threads, thereby optimizing performance by avoiding the overhead of repeatedly creating and destroying threads.

D. Programmability

Algorithm	OMP Compilation Time (s)	Rust Compilation Time (s)
Monte Carlo	0.637	0.815
CNN	2.025	19.330
Merge Sort	1.432	16.623
Prefix Sum	1.723	16.479
N Body	0.162	1.707
KNN	0.165	2.852
Reduce	0.132	9.600
Matrix Multiplication	0.412	12.800

TABLE XI: Compilation Time Comparison for OMP and Rust Implementations.

Table XI shows the compilation time, which is measured in seconds. Rust generally incurs a higher compilation time due to its additional safety checks and optimizations.

Algorithm	Metric	SEQ (Baseline)	OpenMP	OpenMP Effort (%)
Monte Carlo	#lines	32	44	37.5
	#characters	643	1130	75.7
CNN	#lines	48	55	14.6
	#characters	862	930	9.9
Reduce	#lines	52	56	7.6
	#characters	867	953	9.9
Matrix Multiplication	#lines	90	96	6.7
	#characters	1775	1916	7.9
Prefix Sum	#lines	14	15	7.1
	#characters	330	398	20.6
Merge Sort	#lines	42	66	57.0
	#characters	1099	1441	31.1
N Body	#lines	75	98	30.67
	#characters	2441	2632	7.82
KNN	#lines	90	118	31.11
	#characters	2858	3283	14.87

TABLE XII: Effort Metrics for OpenMP Parallelization.

Algorithm	Metric	SEQ (Baseline)	Rust	Rust Effort (%)
Monte Carlo	#lines	30	37	23.3
	#characters	858	1149	33.9
CNN	#lines	43	46	6.9
	#characters	1019	1153	13.2
Reduce	#lines	24	30	25.0
	#characters	486	697	43.4
Matrix Multiplication	#lines	76	84	10.5
	#characters	1667	1855	11.3
Prefix Sum	#lines	17	21	23.5
	#characters	421	507	20.4
Merge Sort	#lines	38	58	52.6
	#characters	1041	1375	32.1
N Body	#lines	83	100	20.48
	#characters	3205	3346	4.4
KNN	#lines	107	125	16.82
	#characters	3600	3957	9.92

TABLE XIII: Effort Metrics for Rust Parallelization.

The table XII compares the serial (SEQ) and parallel (OpenMP) code, showing the additional work required for parallelizing the algorithms in terms of lines and characters of code. It can be shown that most of the parallelism efforts are below 20%. Some algorithms only require less than 10% extra work. We could conclude that OpenMP is a user-friendly parallelism tool since it doesn't require too much extra effort. The table XIII compares the serial (SEQ) and parallel (Rust) code, showing the additional work required for parallelizing the algorithms in terms of lines and characters of code. Most of the parallelized algorithms require less than 30% effort. This number is still very low for programmability. We believe that with only, 30% additional work, a a programmer can easily parallelize their code with Rust. However, most of the algorithm programs in Rust require more extra work than OpenMP. The table shows that parallelism in Rust is a slightly higher workload than OpenMP.
Another factor we should consider is that OpenMP is easier to code, if people already know how to code C/C++, and Fortran, it can directly take over this tool and call the OpenMP macro definitions to parallelize the program. However, OpenMP is error-prone. When the users try to use the OpenMP, they have to make sure they call the built-in function correctly.

One example to explain this is the random number. We tried to run the function $rand()$ in different threads and this is significantly slower than using $rand\_r()$. This is because, in C, the $rand()$ function shares the same seed across all threads, leading to resource contention. As the number of threads increases, this contention becomes more pronounced, further degrading performance. This example highlights the importance of understanding the language's behaviour to avoid such issues when the user wants to use OpenMP, as improper usage can lead to performance problems. Compared with this, Rust provides the function such as $rand :: thread\_rng()$.

VI. Conclusion

In this study, we created a benchmark comparing eight algorithms: Monte Carlo Simulation, CNN Forward Propagation, Merge Sort, Prefix Sum, N-Body Problem, K-Nearest Neighbors, Reduce Problem, and Matrix Multiplication. These algorithms were implemented in both OpenMP and Rust. Additionally, we analyzed the advantages and concerns of each tool across various scenarios, considering factors such as performance, scalability, overhead, and programmability. Among these factors, the first three—performance, scalability, and overhead—are more related to programming, while the last one—programmability is more relevant to software development. Both sets of factors are crucial for real-world applications.

From a programming perspective, OpenMP is a library that allows experienced developers to quickly parallelize their programs with minimal effort, but it still presents some challenges, particularly in ensuring correct memory management and thread synchronization. Our experiment found that OpenMP is more prone to user errors, especially when managing parallel execution details. In contrast, Rust is primarily a language that offers a variety of libraries for users to choose from and tries to protect the users from potential mistakes, but it comes with a steep learning curve due to its strict ownership and borrowing model, which provides stronger safety guarantees at compile time.

For software development, Rust's well-developed ecosystem for specific applications makes it a more suitable option for large-scale software development projects, such as web servers, distributed systems, and embedded systems, where long-term maintainability, performance, and memory safety are critical. In contrast, OpenMP is more suited for smaller-scale projects or legacy systems that require quick parallelization of existing code, such as scientific simulations, numerical computing, or image processing, where performance is a priority, but the focus is on rapid implementation rather than long-term safety and scalability.

For performance, Rust's memory management model, which focuses on ownership and borrowing, provides strong guarantees about memory safety but can result in higher memory consumption. On the other hand, OpenMP allows developers more control over the program's structure and parallelization, enabling them to fine-tune performance based on the specific needs of the application. Therefore, OpenMP offers a more flexible environment for programmers to optimize their code for better performance.

For future work, it would be interesting to extend our current study by focusing on finer-grained parallelism and conducting additional experiments to explore the synchronization and communication costs in more detail. This would provide deeper insights into how these factors affect performance and scalability in different parallel programming environments. Additionally, investigating hybrid approaches that combine the strengths of both OpenMP and Rust could be a promising direction to further optimize performance in complex applications.

References

[1] Ian Goodfellow. Deep learning, 2016.

[2] R Chandra. *Parallel Programming in OpenMP*. Academic Press, 2001.

[3] Kartik Chauhan. Parallel processing in rust, 2023. Accessed: 11/2024.

[4] Henri Casanova, Arnaud Legrand, and Yves Robert. *Parallel algorithms*. Chapman and Hall/CRC, 2008.

[5] Zhengqi Dong, Yunlu Deng, and Pingcheng Dong. Cnn optimizations.

[6] W. Daniel Hillis and Guy L. Steele. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, December 1986.

[7] Robert C. Hockney and Nicholas J. Eastwood. *Computer Simulation Using Particles*. CRC Press, 1981.

[8] John H. Makino. Particle simulations with large numbers of particles. *Computer Physics Communications*, 125(3):281–298, 1999.

[9] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, 2007.

[10] Keichi Takahashi, Kohei Ichikawa, Joseph Park, and Gerald M Pao. Scalable empirical dynamic modeling with parallel computing and approximate k-nn search. *IEEE Access*, 11:68171–68183, 2023.

[11] Walid Abdala Rfaei Jradi, Hugo Alexandre Dantas do Nascimento, and Wellington Santos Martins. A fast and generic gpu-based parallel reduction implementation. In *2018 Symposium on High Performance Computing Systems (WSCAD)*, pages 16–22. IEEE, 2018.

[12] Jack Dongarra and et al. The linpack benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2008.

[13] William Gropp, Ewing Lusk, and Anthony Skjellum. Openmp: An api for shared memory multiprocessing. In *Proceedings of the 1997 International Conference on Parallel Processing*. IEEE, 1997.

[14] Rohit Chandra, Leo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, Ramesh Menon, and Ananth Grama. Parallel programming with openmp. In *Parallel Programming in C with MPI and OpenMP*, pages 1–20. McGraw-Hill Education, 1999.

[15] OpenMP Architecture Review Board. *OpenMP Application Programming Interface (OpenMP API) Specification Version 5.0*, 2020. https://www.openmp.org/specifications/.

[16] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2018.

[17] Graydon Hoare. Rust: Safe, concurrent, practical systems programming. https://www.researchgate.net/publication/277108748_Rust_Safe_Concurrent_Practical_Systems_Programming, 2014.

[18] Aaron Turon. Why rust? a memory-safe systems programming language. *Communications of the ACM*, 58(6):50–59, 2015.

[19] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[20] Eric Barszcz and et al. The perfect club benchmarks: Effective performance evaluation of supercomputers. *International Journal of High Performance Computing Applications*, 5(3):63–75, 1994.

[21] Roger W Hockney. *The science of computer benchmarking*. SIAM, 1996.

[22] François Cantonnet, Yiyi Yao, Mohamed Zahran, and Tarek El-Ghazawi. Productivity analysis of the upc language. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, page 254. IEEE, 2004.

[23] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.