Comparing OpenMP and Rayon

Manoj Middepogu **Pranav Jangir** **Yaswanth Orru**

Abstract

In this project, we conducted a comparative analysis of various aspects of parallel programming, specifically focusing on OpenMP and Rayon. We implemented distinct approaches to matrix multiplication, encompassing standard full-level parallelization, blockwise multiplication, Strassen's algorithm, and a linearize way of multiplication method. Simultaneously, we monitored perf events to observe page faults, cache references, cache misses, and hits. Our aim was to examine aspects such as thread synchronization times, the efficacy of caching mechanisms, and the performance implications when parallelizing a very large sequential task

We observed that Rayon exhibited superior performance in terms of thread synchronization, while OpenMP showcased a notable advantage in leveraging caching strategies and reductions. Notably, as the size of the task, here matrix surpassed (2000 x 2000), both frameworks experienced a significant increase in execution times, highlighting a potential scalability threshold for the chosen matrix multiplication strategies. Our github code is available here

1 Introduction

Parallel programming has become increasingly crucial in harnessing the computational power of modern multi-core processors and high-performance computing environments. Two programming paradigms that address parallelism, OpenMP and Rust, offer distinct approaches to concurrent execution. In this comparative analysis, we delve into various aspects of OpenMP and Rust parallel programming, evaluating their ease of coding, thread management, synchronization mechanisms, runtime performance, and language-specific features.

2 Background

2.1 Rust

Rust is a general purpose programming language that aim to provide concurrency support and empower developers to write smooth parallel programs, its foundational goal revolves around ensuring memory safety without compromising on performance. Rust achieves this through its ownership system, borrowing rules, and lifetimes, effectively eliminating common memory-related errors such as null pointer dereferences and data races.

Ownership: In Rust, each value has only one owner at a time and when the owner goes out of scope, the value is dropped. Instead of passing ownership, Rust allows borrowing references to values with two types of references - mutable and immutable.

Borrowing Rules: Rust imposes strict rules for reference borrowing to prevent data race races and other memory safety issues. The rules ensure that at any given time, you can have either one mutable reference or any number of immutable references, but not both.

Lifetime System: Rust uses lifetimes(borrow-checker) to keep track of all the references and enforce scope of references during compilation.

No Dangling Pointers As a result of borrowing rules and maintaining lifetimes of references, there wont be any dangling pointers.

Rust is known for its zero-cost abstraction, where it provides all the above functionalities, and other functionalities like pattern matching and functional paradigms are provided in such a way that the performance is almost similar to writing low-level code. The emphasis on memory safety also makes Rust particularly well suited for systems programming where reliability and performance are critical.

2.2 Rayon

Rayon is a data parallelism library for Rust that simplifies parallel programming by abstracting away many of the complexities associated with multi-thread programming. Developers can parallelize computations with minimal code adjustments from sequential one, leveraging Rayon's rich functional abstractions that mirror their sequential counterparts. Rust facilitates the parallelization of simple data structures through par_iter and addresses more complex cases with par_bridge. Furthermore, Rust introduces par_sort() for parallel sorting and par_extend for extending operations in a parallel context. To aggregate results from diverse threads, Rust provides functions like map(), sum(), and collect().

```
let numbers = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

// Using normal iterator (sequential)
let sum_seq: i32 = numbers.iter().sum();
println!("Sum using sequential iterator: {}", sum_seq);

// Using parallel iterator
let sum_par: i32 = numbers.par_iter().sum();
println!("Sum using parallel iterator: {}", sum_par);
```

Listing 1: Code for computing sum of numbers in sequential program in rust vs parallel program in rayon. Conversion will happen by simply replacing sequential iter function(iter()) with parallel iter function(par_iter())

Some of the important rayon concepts and terminology are explained below.

Work Stealing: Rayon employs dynamic task scheduling through a mechanism known as Work Stealing. This technique involves breaking down a larger task into smaller, independent tasks and maintaining local task queues on each thread. As a thread completes its assigned tasks, it can "steal" tasks from the queues of other threads. Importantly, there are no race conditions in this process, as Rayon ensures that the tasks in the queues are inherently independent. This approach improves parallelism by efficiently distributing the workload among multiple threads.

Join Method: The join function allows the execution of two closures in parallel. It uses the above work stealing to optimally schedule different tasks.

scope Method: The scope function in Rayon enables asynchronous spawning of an arbitrary number of tasks. While it shares some functionality with join, scope provides finer control over variables used in different parallel tasks. However, this increased control comes at the expense of losing the dynamic scheduling feature of Rayon across different tasks.

Traits: In Rust, general types such as int and str automatically have traits such as Send and Sync implemented, ensuring their thread safety. This automatic implementation helps prevent undefined behavior and data races when these types are shared across different threads.

2.3 OpenMp

OpenMp is an API for writing multithreaded applications in C, C++, and fortan. It provides a set of compiler directives (pragma directives), that can be inserted into sequential code to make it

parallel, library routines, and environmental variables to control various parameters related to parallel programming.

```
    int numbers[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int n = sizeof(numbers) / sizeof(numbers[0]);

    // Using normal iteration (sequential)
    int sum_seq = 0;
    for (int i = 0; i < n; ++i) {
        sum_seq += numbers[i];
    }
    printf("Sum using sequential iteration: %d\n", sum_seq);

    // Using parallel iteration with OpenMP
    int sum_par = 0;
    #pragma omp parallel for reduction(+:sum_par)
    for (int i = 0; i < n; ++i) {
        sum_par += numbers[i];
    }
    printf("Sum using parallel iteration with OpenMP: %d\n", sum_par);
```

Listing 2: Code for computing sum of numbers in sequential vs parallel program in OpenMp. TO the sequential code we have to add compiler directives to make it parallel

2.3.1 Fork-join Parallelism

OpenMp internally uses the Fork-Join mechanism to introduce parallelism. The masted thread forks child threads to execute in parallel whenever it encounters '**#pragma omp and parallel**' and joins (synchronizes) all the forked threads to continue the sequential execution of the program. Barries such as '**#pragma omp barrier**' invoke this join mechanism.

3 OpenMp vs Rayon

While both OpenMP and Rayon facilitate the development of parallel programs, they diverge significantly in their underlying design principles. These distinctions lead to varying strengths and weaknesses, making one library preferable over the other in certain aspects. In this section, we will delve into a comparative analysis of various parallel programming aspects and do a performance evaluation of both the languages through the implementation of four matrix multiplication variants.

3.1 Parallel Programming Aspects

Feature	OpenMp	Rayon
Nested Paralleslism	✓	✓
Parallel execution of Different tasks	✓	✓
Collapse	✓	×
Thread Configuration	Manual/Automatic	Manual/Automatic
Thread Management	Manual/Automatic	Automatic
Thread Synchronization	Manual/Automatic	Automatic
Race Condition Resolution	Manual	Automatic
Reduction Abstraction	✓	✓
Differernt scopes	✓	✓
Type Checking	×	✓
nowait	✓	×
taskwait	✓	×

Table 1: How Rayon and OpenMp compares across various parallel programming aspects.

3.1.1 Nested Parallelism

Nested parallelism refers to the situation where parallel regions of code are executed within other parallel regions. Both OpenMP and Rayon support nested parallelism. In OpenMp nested parallelism is achieved by placing a parallel construct code block inside another parallel construct. The scope function in rust allows to create the nested parallelism.

```
    rayon::scope(|s| {
        s.spawn(|_| {
            // Outer parallel region
            (0..2).into_par_iter().for_each(|_| {
                println!("Outer parallel region, Thread ID: {:?}", std
    ::thread::current().id());

                // Inner parallel region
                (0..3).into_par_iter().for_each(|_| {
                    println!("Inner parallel region, Thread ID: {:?}",
     std::thread::current().id());
                });
            });
        });
    });
```

Listing 3: Sample template for nested parallelism code in rayon

```
    #pragma omp parallel num_threads(2)
    {
        printf("Outer parallel region, Thread ID: %d\n",
    omp_get_thread_num());

        #pragma omp parallel num_threads(3)
        {
            printf("Inner parallel region, Thread ID: %d\n",
    omp_get_thread_num());
        }
    }
```

Listing 4: Sample template for nested parallelism code in OpenMp

3.1.2 Parallel Execution of different tasks

In OpenMP, the sections construct is used to execute different sets of tasks concurrently. Rust, on the other hand, provides the join function to concurrently execute different closures.

```
    let mut matrix = vec![vec![0; SIZE]; SIZE];

    matrix.par_iter_mut().enumerate().for_each(|(i, row)| {
        row.par_iter_mut().enumerate().for_each(|(j, elem)| {
            *elem = i * SIZE + j;
        });
    });
```

Listing 5: Sample code snipped for parallelizing nested for loop in Rayon

```
    int matrix[SIZE][SIZE];

    #pragma omp parallel for
    for (int i = 0; i < SIZE; ++i) {
        #pragma omp parallel for
        for (int j = 0; j < SIZE; ++j) {
            matrix[i][j] = i * SIZE + j;
        }
    }
```

Listing 6: Sample code snippet for parallelizing nested for loop in OpenMp

3.1.3 Collapse

OpenMP provides collapse functionality to automatically collapse nested for loops into a single for loop reducing the synchronization overhead.

```
    int matrix[SIZE][SIZE];

    #pragma omp parallel for collapse(2)
    for (int i = 0; i < SIZE; ++i) {
        for (int j = 0; j < SIZE; ++j) {
            matrix[i][j] = i * SIZE + j;
        }
    }
```

Listing 7: Sample code snippet for parallelizing above nested for loop using collapse

3.1.4 Thread Configuration

Both OpenMP and Rust offer control over the number of threads created for parallel execution.

```
rayon::ThreadPoolBuilder::new().num_threads(4).build_global().unwrap()
    ;
```

Listing 8: Sample code snippet for setting number of threads in OpenMP

```
    omp_set_num_threads(4);
```

Listing 9: Sample code snippet for setting number of threads in OpenMP

3.1.5 Thread load Scheduling

The OpenMP schedule construct offers static, dynamic, and runtime scheduling options to configure thread workloads. In contrast, Rust utilizes work stealing as a mechanism to dynamically balance the workload among threads.

```
    pragma omp parallel for schedule(kind [, chunk_size])
```

Listing 10: Sample code snippet for setting number of threads in OpenMP

3.1.6 Thread Synchronisation

In OpenMP, synchronization between threads is achieved using barriers. The **#pragma omp barrier** can be used to manually insert a barrier into the code, and the closing braces "}" at the end of a parallel block act as an implicit barrier. In Rust, there is generally no need for explicit barriers, as the Rayon library handles scheduling and synchronization automatically.

3.1.7 Race Condition Resolution

OpenMP provides constructs like '**atomic**' and '**critical**' to manage the update of shared variables, preventing race conditions. In Rust, the ownership system and borrowing rules are employed to eliminate various types of race conditions.

3.1.8 Reduction Abstraction

OpenMp provide 'reduction' construct to aggregate the results from different threads. Rust provides functions such as '**map**', '**sum**' to collect the results from different threads.

3.1.9 Type checking

Type checking in rust allows for proper applying of rust borrowing rules, preventing any occurrence of data races.

3.1.10 Different scopes

Both languages support creating resticting the scopes of variables to a specific section of code. OpenMp sections

```
rayon::scope(|s| {
    s.spawn(|_| {
        // Task 1
    });

    s.spawn(|_| {
        // Task 2
    });
});
```

Listing 11: Sample code snippet for setting number of threads in OpenMP

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        // Task 1
    }

    #pragma omp section
    {
        // Task 2
    }
}
```

Listing 12: Sample code snippet for setting number of threads in OpenMP

3.1.11 nowait and taskwait

In OpenMP, constructs such as nowait and taskwait are used to control thread synchronization. The nowait clause is used to indicate that a thread should not wait for other threads to complete before proceeding to the next section of code and the taskwait construct is similar to a barrier but specifically waits for the completion of child tasks spawned within the parallel region by the current thread.

```
#pragma omp parallel
{
    #pragma omp for nowait
    for (int i = 0; i < N; ++i) {
        // Parallel work without waiting
    }

    // This section is reached without waiting for all threads to
    finish the loop
}
```

Listing 13: Sample code snippet for setting number of threads in OpenMP

```
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task
        {
```

```
            // Task 1
        }

        #pragma omp task
        {
            // Task 2
        }

        #pragma omp taskwait
        // This section is reached only after Task 1 and Task 2 are
    completed
    }
}
```

Listing 14: Sample code snippet for setting number of threads in OpenMP

3.2 Performance Evaluation

We have tried to benchmark Runtime overhead and Matrix Multiplication variants on OpenMP and Rayon under different settings.

3.2.1 RunTime Overhead

To assess the basic overhead of thread management, we implemented a simple program using both OpenMP and Rust with the goal of measuring thread creation time and thread synchronization times. The program includes a sleep of 1 second to simulate the actual duration of the task. Notably, from Figures 1, 2 and 3 we observed that thread creation times and synchronization times (total time - creation time - sleep time) were significantly lower for Rayon compared to OpenMP. This can be attributed to rust ownership and borrwing rules.

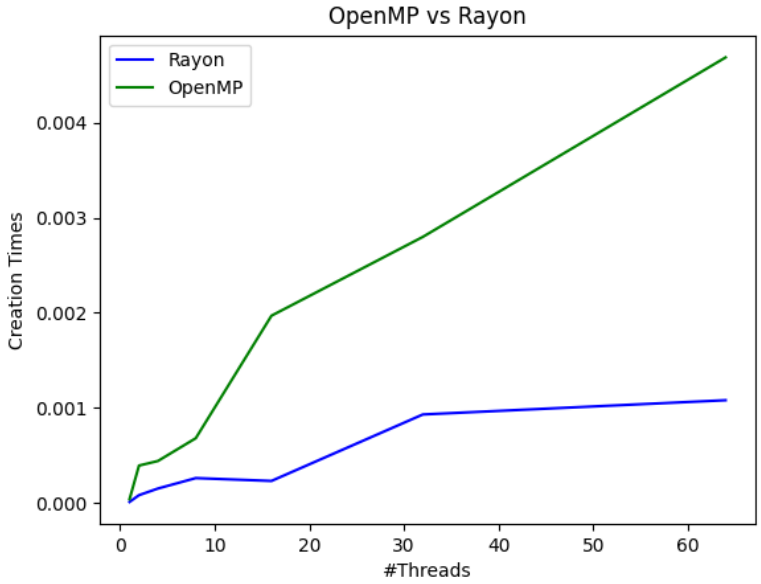

Figure 1: Creation Time vs Number of Threads

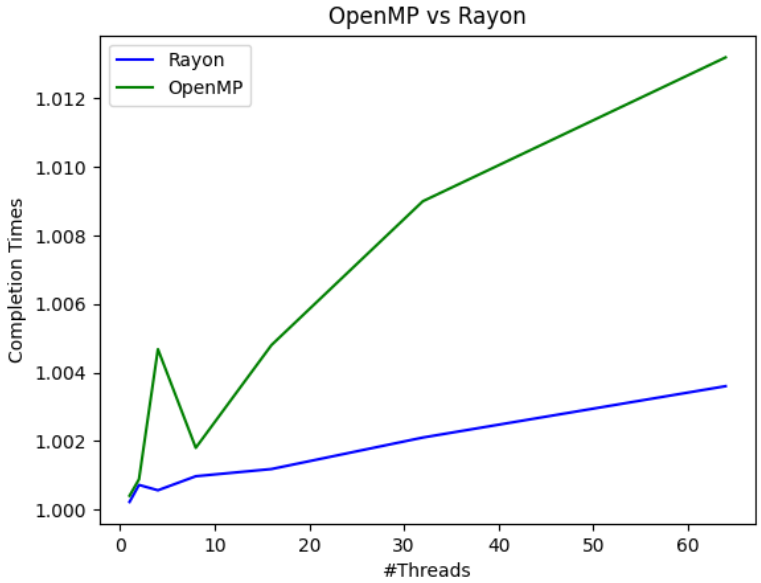

Figure 2: Completion Times vs Number of Threads

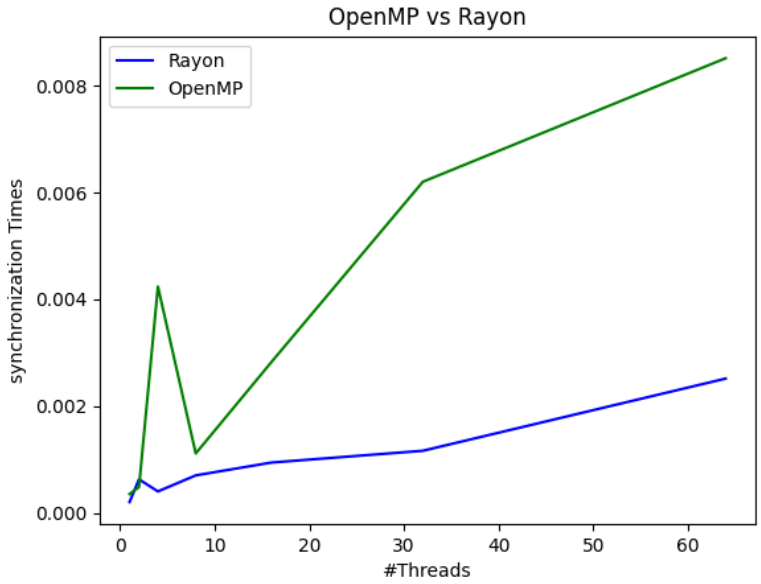

Figure 3: Synchronization times vs Number of Threads

3.2.2 Matrix Multiplication

Setup: We tested parallelized versions for Full matrix multiplication, block-wise matrix multiplication, Strassen's algorithm and variants on linear representation of the matrix. For every implementation we are plotting graphs 'time vs no.of threads', 'time vs size of the matrix', 'speed-up' and 'Efficieny' along with cache perf reports. We compiled both c++ and Rust code with optimizer flags, -03 for C++ and –release for Rust for optimized binaries. We also tried to implement the algorithms in a similar way as far as the constructs support in both languages.

Full parallelism:
As the name indicates, we implemented nested parallelism with 'for' loops to compute the product of the matrices in Rust and C++. We use "for collapse(2) schedule(static)" for C++, par_iter for Rust. Programming in Rust is a bit different with ownership and mutable reference restraints. Please refer to my code here for more details.

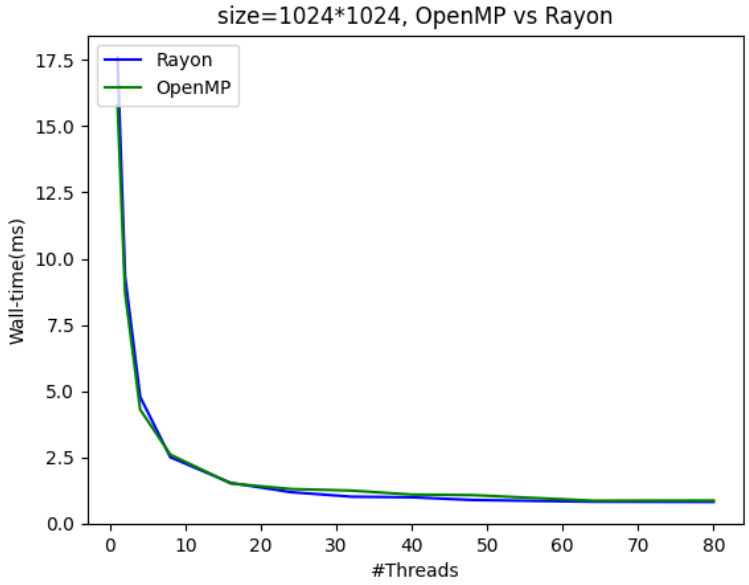

Figure 4: Time vs Number of threads

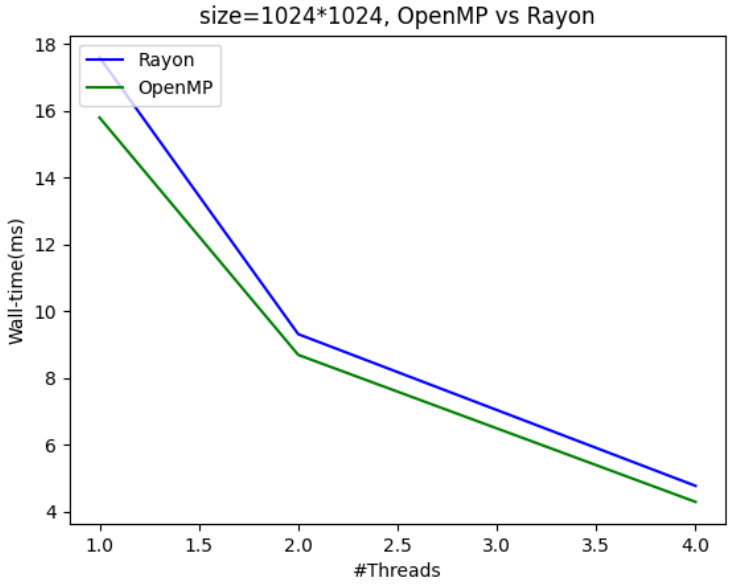

Figure 5: Zooming in Time vs Number of threads at less number of threads

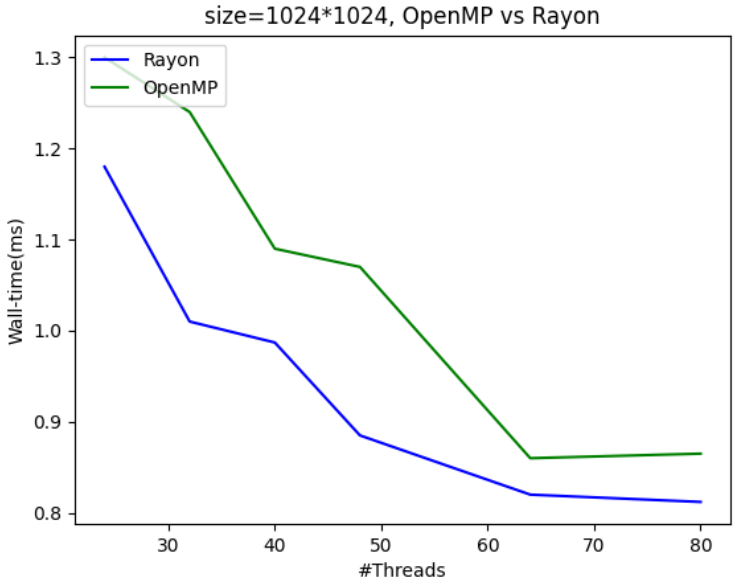

Figure 6: Zooming in Time vs Number of threads at higher number of threads

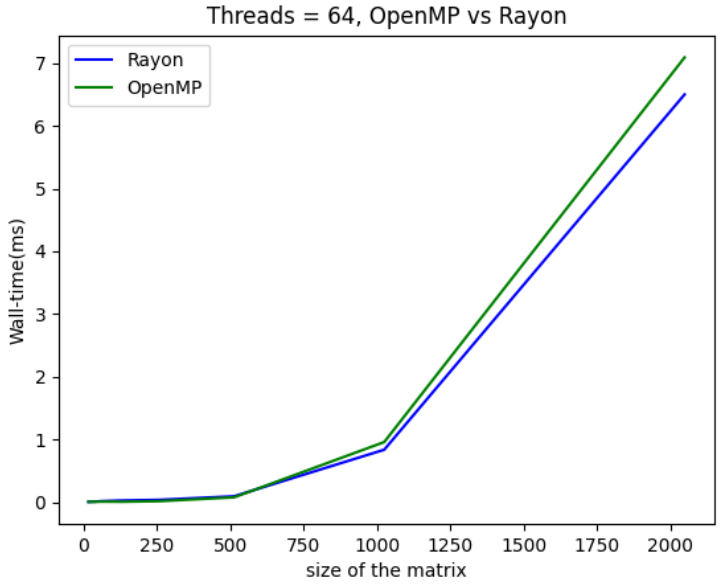

Figure 7: Time vs size of the matrix at threads=64

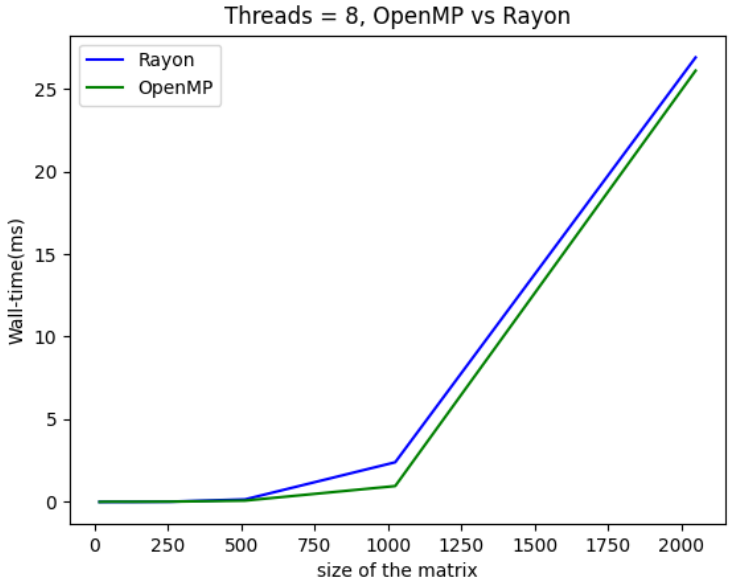

Figure 8: Time vs size of the matrix at threads=8

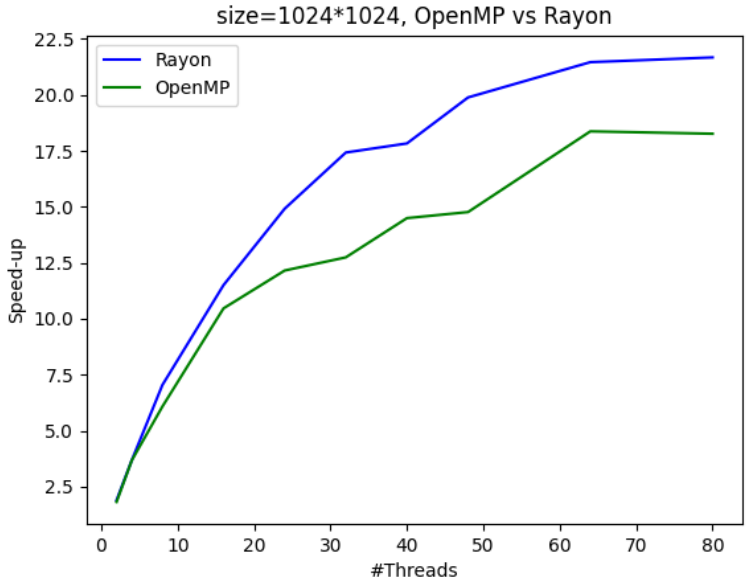

Figure 9: Speed up

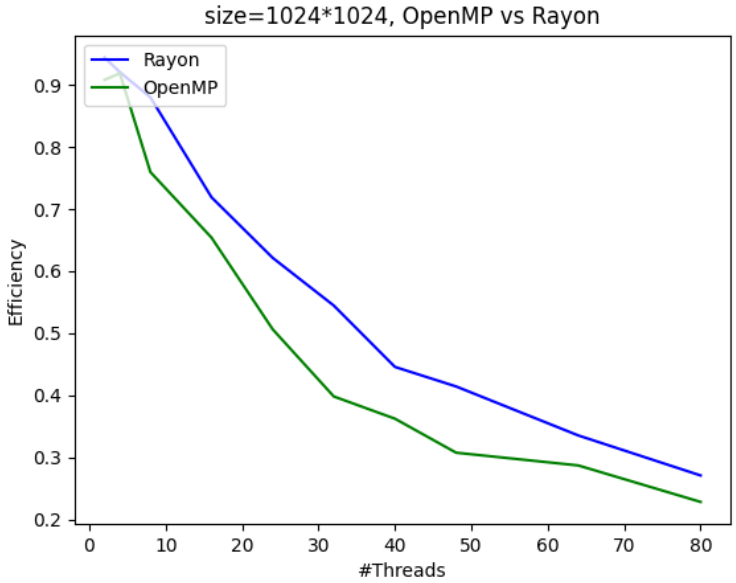

Figure 10: Efficiency

Observing Figures 4, 5, 6, it becomes evident that Rayon performs better than OpenMP with an increasing number of threads. This observation is corroborated by Figures 7 and 8, where Rayon demonstrates better performance with a higher number of threads, while OpenMP exhibits better performance with a lower number of threads. In addition to that, Figures 9 and Figure 10 also clearly show that the speedup and efficiency is better for Rayon than OpenMP.
The observed performance difference can be attributed to the minimal synchronization cost in Rayon, thanks to Rust's ownership and borrowing rules. This highlights the efficient implementation of rust's zero-cost abstractions, which performs better compared to the slightly lower-level code of C++.

Block-wise parallelism: We implemented blockwise parallelism where we divided each row and column into a 16 block size which is the same as the size of the cache line (16 * 4 = 64 bytes). We found from multiple experiments with different block sizes along with perf report that 16 is the optimal block size aligned with memory layout, cache.

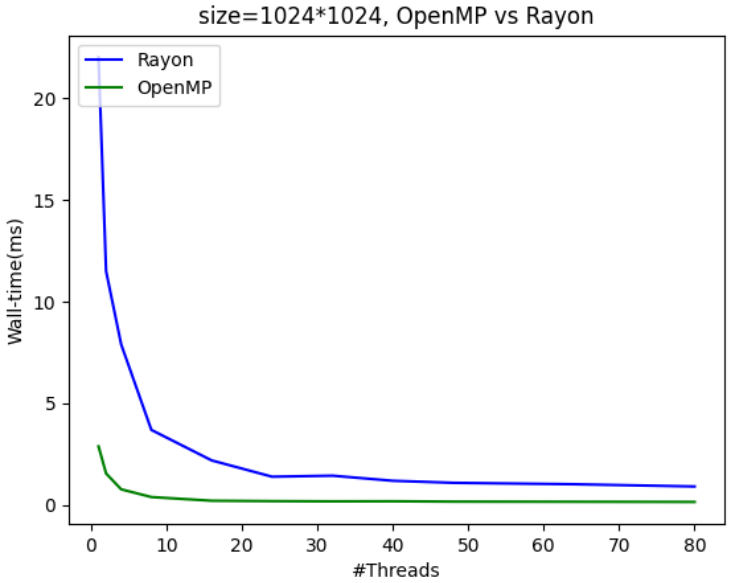

Figure 11: time vs no.of threads for array size of 1024*1024

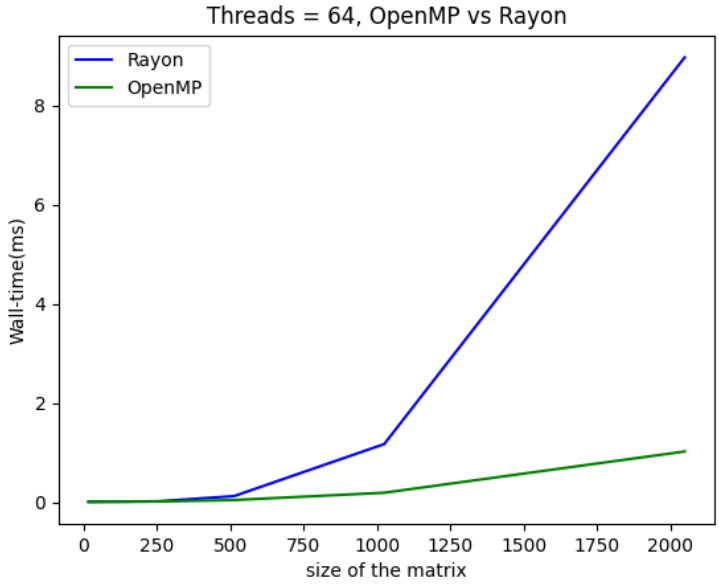

Figure 12: time vs size of array at 64 threads

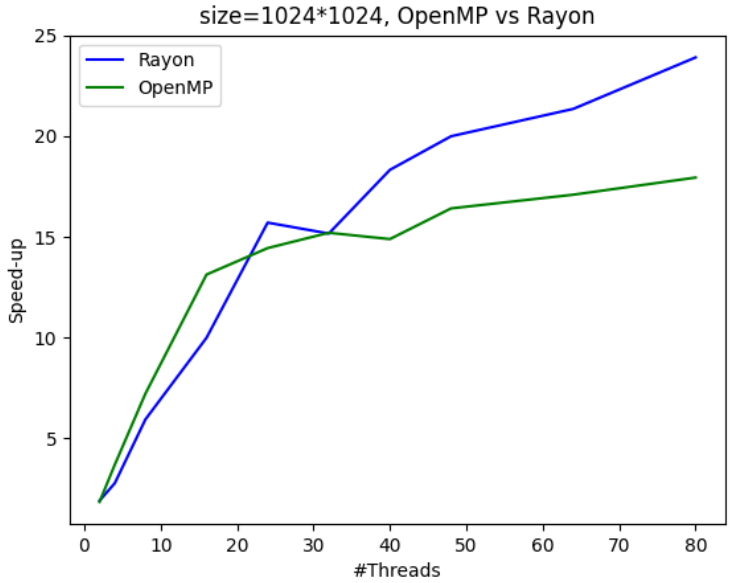

Figure 13: Speed up

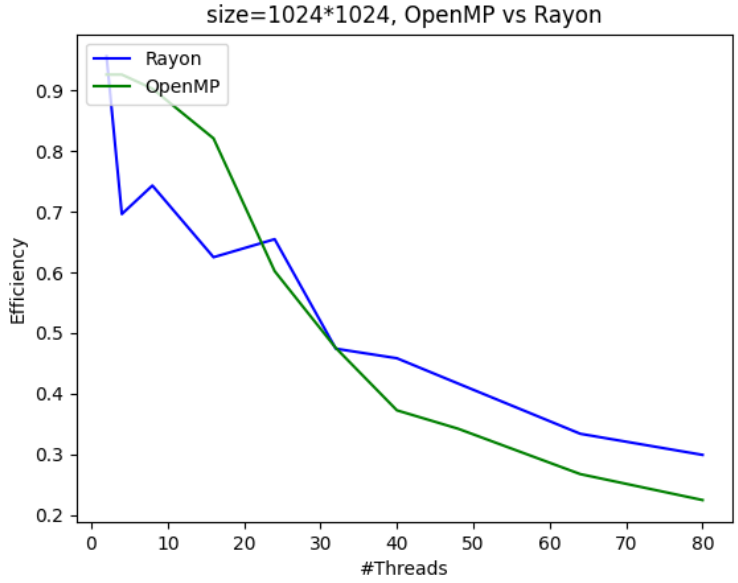

Figure 14: Efficiency

```
[mm12799@crunchy1 open_mp]$ perf stat -e page-faults,cache-misses,cache-references ./matrix_multiply
1.03876 0

 Performance counter stats for './matrix_multiply':

            12,948      page-faults
        48,536,782      cache-misses             #    2.900 % of all cache refs
     1,673,781,754      cache-references
```

Figure 15: Perf Report for OpenMp with 64 threads with matrix size of 2048

```
Performance counter stats for 'cargo run --release -- 64 2048 1':

          84,086      page-faults
     122,206,414      cache-misses              #    0.573 % of all cache refs
  21,320,932,884      cache-references
```

Figure 16: Perf Report for Rust with 64 threads with matrix size of 2048

```
       6,670      page-faults
   4,982,688      cache-misses              #    2.240 % of all cache refs
 222,419,491      cache-references
```

Figure 17: Perf Report for OpenMp with 1 thread with matrix size of 2048

```
Performance counter stats for 'cargo run --release -- 1 1024 1':

          18,286      page-faults
       9,041,154      cache-misses              #    0.456 % of all cache refs
   1,982,333,557      cache-references
```

Figure 18: Perf Report for Rust with 1 thread with matrix size of 2048

The observed increase in the time taken from Figure 11 by Rayon contrasts with a significant decrease in the time taken by OpenMP. This can be attributed to the caching mechanism, when one element in a block is accessed, the entire block is moved to the cache. As OpenMP, by default, evenly divides tasks among threads, reaping the benefits of caching. On the other hand, Rayon dynamically assigns each task(computation using a single element) to different threads, increasing the likelihood that the same elements in the block are assigned to different threads, and thus, not fully capitalizing on caching, leading to no significant performance gain.
We tested our hypothesis by checking cache references using perf and we can clearly see from Figure 15, 16 that the cache references for openmp are decreased by a factor of 16. This shows that for every 16 elements(our block size) only the program refers to cache only once, which saves a lot of time.
But by examining Figures 13 and 14 we can see that rayon still have a better speed up OpenMp. This can be shown with Figures 17 and 18 where already openmp has the advantage of factor 5 on rayon and with increasing threads, this factor can increase to maximum 16. So after certain number of threads, the increase in speed up from cache hits stagnates and thread synchronization overhead becomes significant which is what we observe in Figures 13 and 14 where openmp performs comparably with rayon until 20 threads and decreases after that. This again shows the efficiency of rayon thread synchronization.

Strassen's Algorithm: This is a recursive variant of the algorithm is known to have lower time complexity than normal matrix multiplication in a sequential program. However, our objective is to see whether the same advantage translates to parallelized code. Also, as this is a divide and conquer algorithm this will have more interdependent reductions, as the number of threads increases this will provide insights on how efficiently reductions are performed in OpenMp and Rayon. Below is the Pseudo-code for Strassens Algorithm. Theoretically it is expected to work in $O(n^{2.8})$ rather than $O(n^3)$, but it has high constant due to more additions and substractions to reduce one extra multiplication. Our implementation uses task type parallelism in both C++ and Rust.

Algorithm 1 Strassen's Matrix Multiplication

Require: Two square matrices A and B of size $n \times n$
Ensure: The product matrix $C = A \times B$

function STRASSEN(A, B)
  if $n < 64$ **then**
    return $A \times B$ ▷ Base case: simple matrix multiplication
  end if
  Divide A and B into four equal-sized submatrices:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \qquad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

  Compute seven products recursively:
    $P_1 = \text{Strassen}(A_{11}, B_{12} - B_{22})$
    $P_2 = \text{Strassen}(A_{11} + A_{12}, B_{22})$
    $P_3 = \text{Strassen}(A_{21} + A_{22}, B_{11})$
    $P_4 = \text{Strassen}(A_{22}, B_{21} - B_{11})$
    $P_5 = \text{Strassen}(A_{11} + A_{22}, B_{11} + B_{22})$
    $P_6 = \text{Strassen}(A_{12} - A_{22}, B_{21} + B_{22})$
    $P_7 = \text{Strassen}(A_{11} - A_{21}, B_{11} + B_{12})$
  Compute the four quadrants of the result matrix C:
    $C_{11} = P_5 + P_4 - P_2 + P_6$
    $C_{12} = P_1 + P_2$
    $C_{21} = P_3 + P_4$
    $C_{22} = P_5 + P_1 - P_3 - P_7$
  return the result matrix C
end function

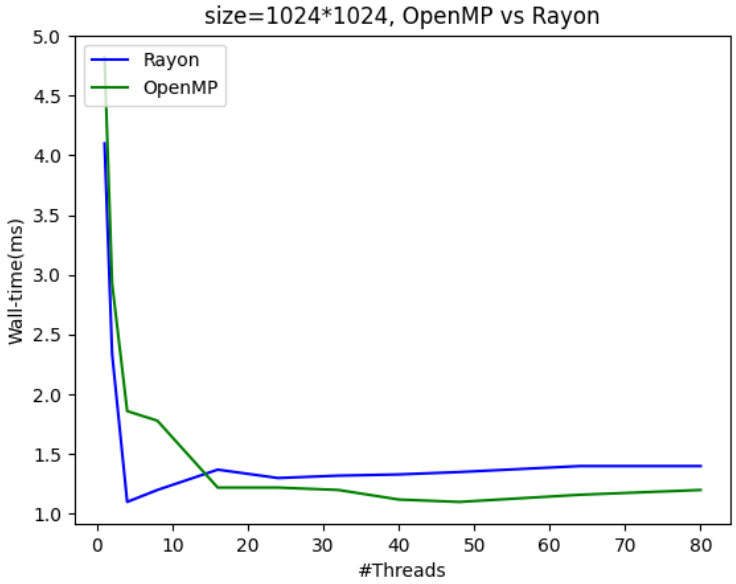

Figure 19: time vs no.of threads at 1024*1024 array size

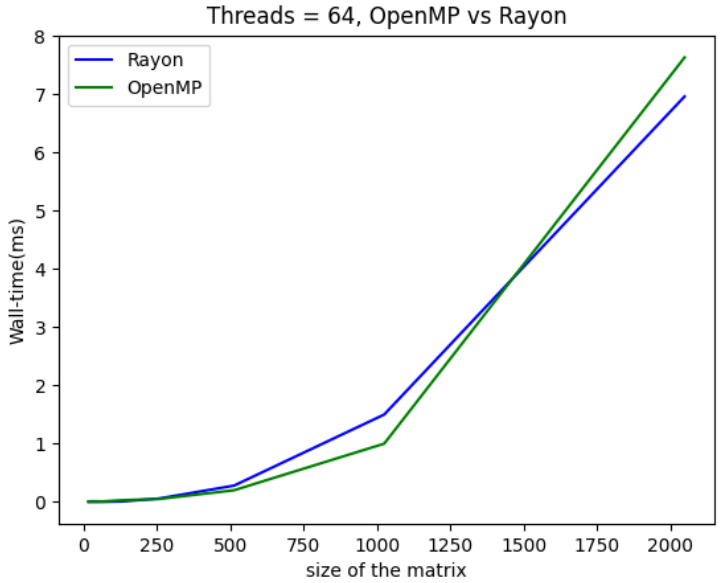

Figure 20: time vs size of array at 64 threads

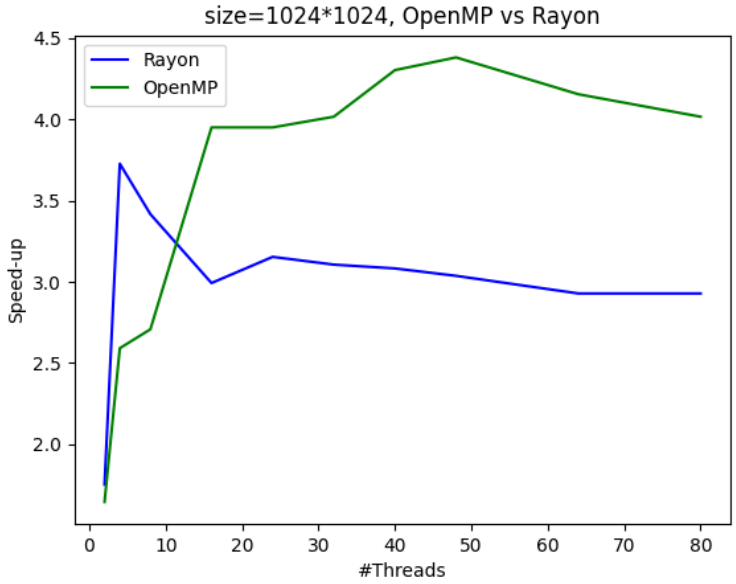

Figure 21: Speed up

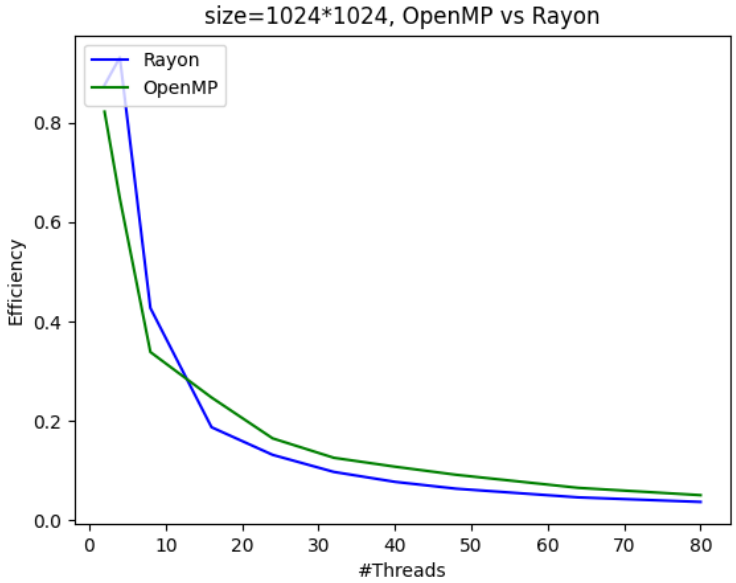

Figure 22: Efficiency

Given that this code involves significant thread communication, from figures 19 we can see that the initial performance drop for Rayon is significant, especially up to 4 threads, as previously demonstrated that rayon thread synchronization is better than OpenMp synchronization. However, as the number of threads increases, the impact of reduction overhead becomes more substantial. Unlike OpenMP, which experiences a gradual decrease in time, Rayon exhibits an increase in time. This observation underscores the inherent cost of Rayon reductions.

Linearised Matrix Multiplication We implemented a linear version of matrix multiplication where we linearly projected the matrix and computed multiplications on these two linear vectors.

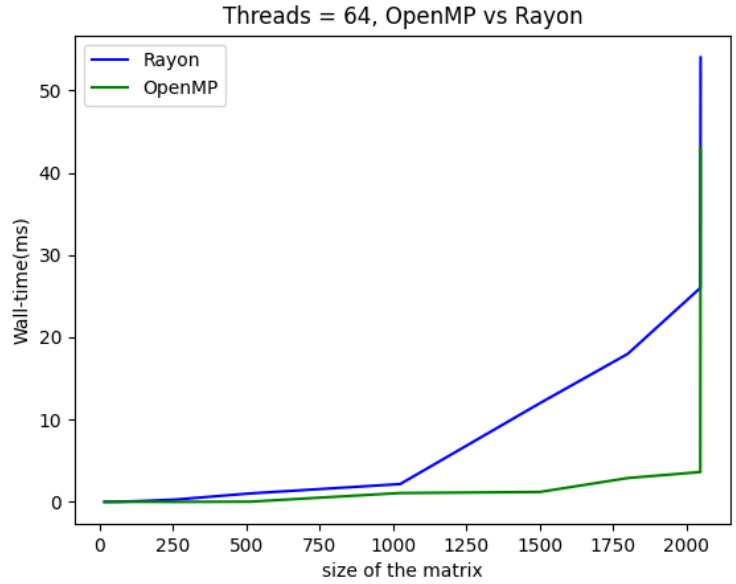

Figure 23: time vs size of array at 64 threads

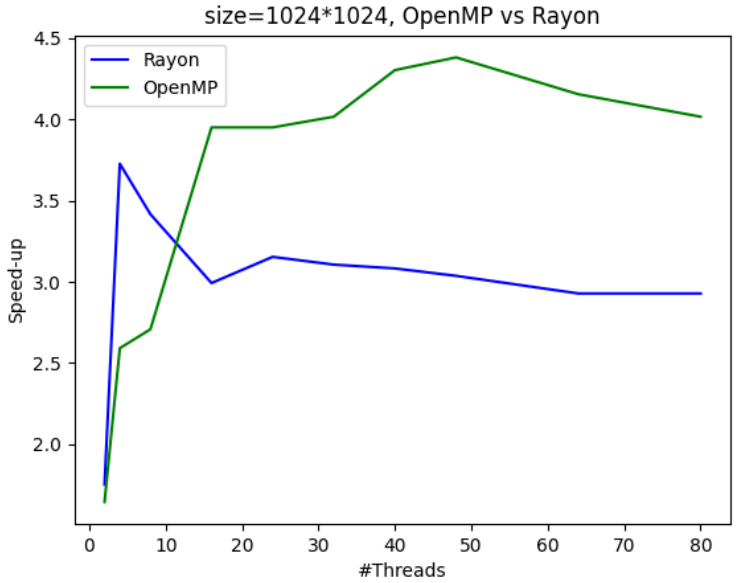

Figure 24: Speed up

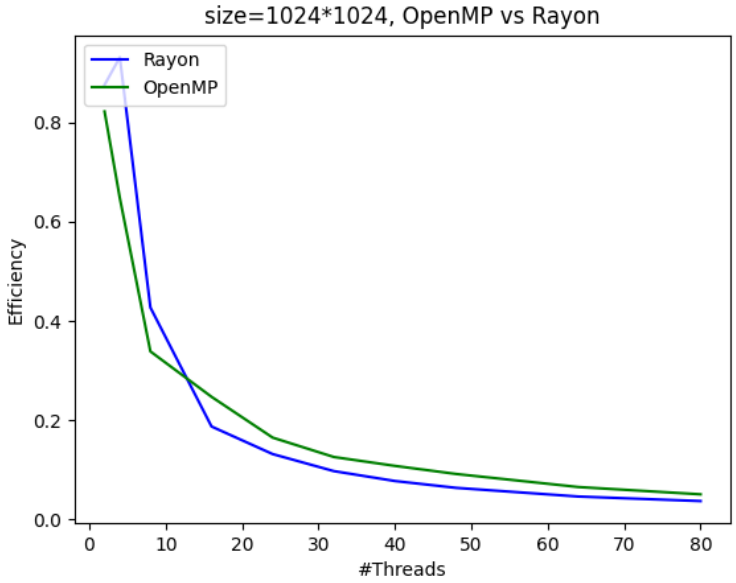

Figure 25: Efficiency

```
Performance counter stats for './matrix_multiply':

          22,243      page-faults
     525,573,690      cache-misses              #    3.736 % of all cache refs
  14,067,011,849      cache-references
```

Figure 26: Perf Report for OpenMp when size is 2047

```
 Performance counter stats for './matrix_multiply':

           47,259      page-faults
    8,319,514,100      cache-misses              #   37.996 % of all cache refs
   21,895,618,240      cache-references
```

Figure 27: Perf Report for OpenMp when size is 2048

```
Performance counter stats for 'cargo run --release -- 64 2047 3':

          35,947      page-faults
   1,180,516,468      cache-misses              #    6.886 % of all cache refs
  17,144,218,283      cache-references
```

Figure 28: Perf Report for Rayon when size is 2047

```
Performance counter stats for 'cargo run --release -- 64 2048 3':

          85,537      page-faults
   8,382,095,551      cache-misses              #   79.841 % of all cache refs
  10,498,439,290      cache-references
```

Figure 29: Perf Report for Rayon when size is 2048

From the figure 23 we can see a clear spike when it reaches a size 2048. This can be explained by Figures 26, 27, 28 and 29 as the cache misses almost from 2047 to 2048 in both the languages.

4 Conclusions

- Rust provides numerous abstractions atop lower-level code, and these abstractions are so efficient that the overhead of thread communication is even less than the slightly lower-level C++ code.
- Although Rust's high-level dynamic scheduling may seem convenient for programming, as demonstrated in blockwise matrix multiplication, there are cases where the OpenMP code, written with careful consideration of cache utilization, can be more efficient.
- In conclusion, based on the above analysis, neither OpenMP nor Rayon emerges as inherently superior. The choice depends on factors such as the nature of the problem, the algorithm in use, and the preferences of the programmer. Rayon proves to be more efficient for simpler algorithms without extensive reductions and when the programmer prefers a streamlined approach with fewer concerns about concurrent programming errors. Conversely, if a problem entails numerous reductions and the programmer desires greater coding flexibility, OpenMP may be the more suitable choice.

5 Challenges

- Writing optimized code in rust
- Using perf to verify the cache-miss hypothesis in blockwise matrix multiplication implementation
- Debugging the reason for sudden spyke in time at matrix size of 2048 in linearized implementation.