Comparison of Multi-threading between C++ and Rust (OpenMP vs Rayon)

Authors:
Nishal Ancelette Pereira <napereir@andrew.cmu.edu>
Supradeep Rangarajan <strangar@andrew.cmu.edu>

Summary

We implemented and compared four benchmarks in Rust and C++ using Rayon and OpenMP respectively. To provide in-depth comparison, we have used multiple configurations for each benchmark. Rayon performed as good as OpenMP in cases where the underlying algorithm or compiler gave an advantage or edge. The downfalls of Rayon are the under-optimized computing function, cost of creating splits of work, and stealing when compared to a possible static scheduling. Rayon performed better for sorting and multiplication of larger matrices. In all other benchmarks, OpenMP had the upper hand. Another advantage Rayon possessed was the failure in compilation of code that had unsafe sharing of variables between threads, allowing us to write correct code always.

Motivation

Over the course of 15-618, we have learned that in order to scale the performance of a software with the hardware, we as developers need to write concurrent, fast and correct applications. While decomposing a given algorithm into parallel workloads, developers need to be careful about new kinds of error including deadlock, livelock and data-races. Hence, in order to get improved performance through parallelism, we need to understand the nuances of these errors and anticipate them in advance.

What Rust believes to provide is an abstraction for thread-safety with zero-cost, thus becoming highly popular among industries and developers (Mozilla being the strongest influence). Rust enables safe concurrency with data locks and message-based communication channels. Furthermore, Rust performs compile time analysis on threads data behavior to determine potential problems. Rust's ownership construct and concurrency rules offer powerful compile time tool to help programmers write safe and efficient concurrent programs. Through this project, we want to get a deeper understanding of how Rust solves the issue of data-races and what speed it provides against C++, as it promises to solve many of the issues a programmer faces when parallelizing C++ code, at zero-cost. Rust provides various libraries (called as crates) for multithreading abstraction. Rayon and Crossbeam are two of the most popular ones, as suggested by the Rust community.

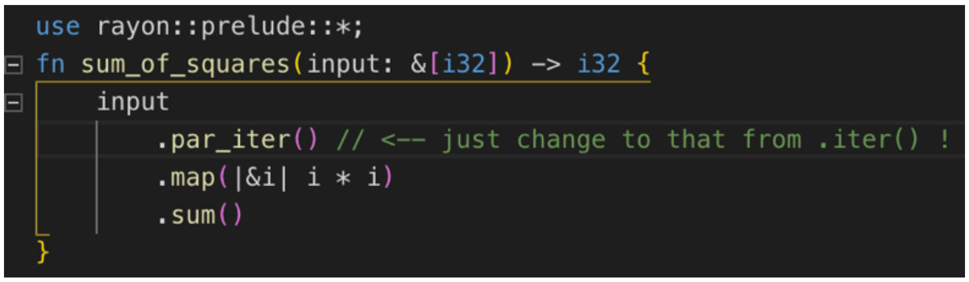
Background

What is Rust?

Rust is new coding language rising the ranks, aiming at safe concurrency, data safe programming at zero abstraction cost. Rust performs compile time analysis on threads data behavior to determine potential problems. Rust's ownership construct and concurrency rules offer powerful compile time tool to help programmers write safe and efficient concurrent programs. This resolves the classic problems of using a variable that is shared between threads without mutual exclusion. At the same time, it keeps you away from most segmentation fault. But it cannot save you from deadlocks, or poorly written code. There is a 'Rust' way of writing code which must be followed to write the most optimized code that will provide the same low-level code as C/C++. Such zero-cost abstraction is highly beneficial for developers, but it comes with the cost of fighting the compiler to get a compiled code (mainly the borrow checker). Also, Rust gives a guarantee that a code which compiles, will almost work perfectly.

Rayon

Rayon is a data-parallelism library for Rust. It is extremely lightweight and makes it easy to convert a sequential computation into a parallel one. It also guarantees data-race freedom. It provides an abstraction for data parallelism which is really simple and easy to implement. For example, if you write a serial code with an iterator, you can simply turn it parallel by using parallel Iterator method.



```rust
use rayon::prelude::*;
fn sum_of_squares(input: &[i32]) -> i32 {
    input
        .par_iter() // <-- just change to that from .iter() !
        .map(|&i| i * i)
        .sum()
}
```

Fig. Example code for sum of squares in Rust

How Rayon works?

Rayon uses the technique of "work stealing" that is very similar to what is employed by the Cilk abstraction for C/C++, hence very suitable for "divide and conquer" type of workload. Rayon when compared to Cilk is much easier to use and is being actively maintained by the developer community.

The basic idea is that, on each call to `join(a, b)`, we have identified two tasks `a` and `b` that could safely run in parallel, but we don't know yet whether there are idle threads. All that the current thread does is to add `b` into a local queue of "pending work" and then go and immediately start executing `a`. Meanwhile, there is a pool of other active threads (typically one per CPU, or something like that). Whenever it is idle, each thread goes off to scour the "pending work" queues of other threads: if they find an item there, then they will steal it and execute it themselves. So, in this case, while the first thread is busy executing `a`, another thread might come along and start executing `b`.

Once the first thread finishes with `a`, it then checks: did somebody else start executing `b` already? If not, we can execute it ourselves. If so, we should wait for them to finish but while we wait, we can go off and steal from other processors, and thus try to help drive the overall process towards completion.

Inherently, this process of stealing and coherently working adds dynamic balancing of load between the threads but adds slight overhead. These behaviors have been analyzed using benchmarks which are provided in the Result section.

What Rayon provides as features?

There are two ways of using Rayon:

1. *High-Level parallel constructs*: one of the most efficient way of parallelizing in Rayon.
    a. `par_iter()`: An abstraction over the join method of Rayon, which allows you to iterate similar to `iter()` method, with all the other abstracting functions like `map()`, `for_each()`, `sum()`, `fold()` etc.

    b. `par_sort()`: A parallel sorting abstraction, which works similar to a `sort()` trait, but has parallelism using Rayon. Rayon provides multiple sorting abstraction that allow sort by keys. This helps in using this API for different sorting situation. par_sort_by()

    c. `par_extend()`: This can be used to efficiently grow collections with items produced by a parallel iterator.

2. *Custom tasks:* It lets you divide your work into parallel tasks yourself.
    a. `join()`: This method is similar to spawning two threads, one executing each of the two closures. You can use it to split a job into two smaller jobs. This join works on stealing style, so it incurs lower overhead than a simple spawn of two threads.

    b. `scope()`: This method creates a scope within which you can create any number of parallel tasks. We can perform any kind of parallel task using this, but they recommend using `join()`, as this is not as optimized. Though we can say that a static assignment will be faster.

    c. `ThreadPoolBuilder()`: It can be used to create your own thread pools or customize the global one. This can be used to modify number of threads. This can be used to create a thread closure while spawning threads.

OpenMP

Through the coursework, we have gained a good understanding of OpenMP abstraction for parallelism in C/C++ which allows compiler directives to be embedded into a serial code. Its API significantly simplifies writing multi-threaded programs in Fortran, C and C++. Although the ease of use and flexibility are the main advantages of OpenMP, it is up to the developer to write safe code. If one is not careful, they could easily run into the hazards of data-races and deadlocks. This could potentially limit the scalability of the code (one of the reasons why it is not employed in modern-day browsers).

How does OpenMP work?

OpenMP uses the fork-join execution model i.e. the master thread spawns a team of threads as needed to allow multiple threads of execution perform tasks. Threads are created in OpenMP using the ***parallel*** construct. The parallel construct itself creates a "Single Program Multiple Data" program. Although it ensures that computations are performed in parallel it does not distribute the work among the threads in a team. If the user does not specify any worksharing, the work will be replicated. On the other hand, the ***worksharing*** construct of OpenMP allows the user to split up pathways through the code between these threads. For example, ***#pragma omp for*** distributes iterations over threads. Scheduling of these iterations can be determined by static, dynamic, guided and runtime. All threads synchronize at an implicit barrier at the end of the loop unless ***nowait*** clause is specified. However, the programmer needs to use synchronization constructs to impose order constraints and to protect access to shared data.

Comparison of Rayon and OpenMP

Parallelism is hard is get right in most programming environments because normally it involves substantial refactoring of code. With Rust and its multithreading crates like Rayon, a programmer with little to no knowledge of parallelism can simply convert the sequential iterations of their code to parallel iterations by importing the crate and changing *iter* to *par_iter.* If its thread-unsafe to do so, the Rust code simply won't compile. This makes Rust code much more scalable than C++ and this is the main difference between Rust and C++. Besides, Rayon's work-stealing inherently divides the load almost equally amongst the threads providing good performance for imbalanced workload. However, since the Rayon crate is still in its nascency, it does not support auto-vectorization of the code unlike OpenMP which has directives (#pragma simd) to do so. In rust, we would have to import additional crates like *faster* to support forced vectorization of code.

Approach

To compare two different parallelism libraries of different languages, we need to see its practicality and see how they perform for the common uses. A developer will look for three things in such a library, ease of use, safe concurrency and speed. Here we choose to benchmark the Rayon library against simpler but fundamental benchmarks and try to see patterns in understanding and further assess it with deeper code research on the Github repository of the crate.

Benchmarks

Benchmarks for the comparison were chosen with few objectives in mind. We wanted to see the performance of the libraries for the inbuilt provided methods that compete directly. Apart from that, we wanted to how they handle dynamic load when provided with such a problem. Rayon inherently can handle dynamic load, but OpenMP needs the schedule(dynamic) directive to provide dynamic balance.

Benchmarks chosen:

1. **Mandelbrot** – It is a simple and embarrassingly parallel benchmark that generates images for visualization of a famous set of complex numbers called the Mandelbrot set. The benchmark allows you to vary the size of the generated image and the maximum number of iterations per point. Reason for use: This is an imbalanced workload since the work required for each pixel of the image varies. For this benchmark, it would be interesting to analyze how parallelism libraries schedule the work efficiently amongst threads. Because of the inherent dynamic scheduling feature of Rayon, we expect it to perform better than OpenMP.

2. **Matrix Multiplication** – A commonly used operation in the recent times, it is a simple benchmark which tests for task division for multiple threads. Reason for use: This is a compute bound problem; thus, we can see how the parallelism libraries behave when doing this task. We also can see how these algorithms behave when written in a cache friendly manner or multi-thread advantageous manner. Another comparison can be done on how both handle a job division when provided with the actual job division (i.e. row first dot products).

3. **Unstable-Stable Sorting** – We are comparing the stable (merge-sort) and unstable (quicksort) sort functionality of Rayon and GNU-parallel sort provided as an extension of OpenMP. In this benchmark, we have sorted 1M, 10M, and 100M elements ascendingly from randomly generated values. Reason for use: The divide and conquer nature of these sorting algorithms makes it a good benchmark for parallelization. Rayon could be more suitable because of “work-stealing” abstractions.

4. **Reduction** – OpenMP added a reduction feature which uses a reduction primitive `reduction(+:result_sum)`. This was specifically added to perform functions like a dot product which is used in Linear Algebra and Geometry. Not only that, but any kind of reduction

can be done, where we add result of some operation on a single element of the array and accumulate it. Reason for use: Though the problem may be memory bound, this is a common operation in Physics (or even in Matrix multiplication), whose speed will help achieve higher speed ups for complex problems. This can also give us insights on how Rayon behaves when given a load that can actually be divided statically.

Use the best serial code for Rust + Rayon

The first part of every benchmark is to write the most optimized serial code in both. The reason for this is that, an unoptimized code if used with Rayon and OpenMP will lead to bad parallelism while scaling to cores for such simple benchmarks. Hence, we must either observe the assembly code of the compiled executable or try to see which written code provides the best optimized assembly. This precaution is taken because both libraries provide a simple abstraction to parallelize by adding a few lines or a few method changes. An unoptimized code, will result in bad parallelism this way.

For example, in one of our benchmarks, Mandelbrot, we used a crate named `num`, which provides a wrapper for complex number variables (`num::Complex32`, `num::Complex64`). Though it is mentioned that this code should perform as well as simple serial, it has some extra assembly commands added to its compiled executable. We have a more detailed analysis for this in the result section.

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

Re-optimization

Rayon provides a very simple way to create parallelism, but as mentioned before, different ways of writing the same code will give us different speeds due to the way the compiler optimizes it. Though this is applicable to OpenMP too, we have the option to eliminate most issues by using `schedule(static)` on the correct For loop, to decide the static division of work between the threads.

The reason we added this as a part of our design to test is because we faced this issue in both Mandelbrot and Matrix multiplication. Naturally we believed that if we write a working code for the serial case, we can easily parallelize it using the abstractions in each library. But in the case of Mandelbrot, we found that writing parallel code over pixels for Rust/Rayon gave a better speedup when compared writing parallel code over rows. This was contradictory to our understanding of the benchmark in C++ and Assignment 1 from 15-618 course. Nevertheless, such issues were observed, documented, and understood, which gave us interesting insights for Rayon.

During Matrix Multiplication, we were able to write multiple variations of the code, which allowed for a different style of parallelization. Only through speed-up graphs, we were able to analyze the difference in speed. Also, to eliminate the effect of cache miss, only for testing this speed we used smaller matrices.

Results

The results or insights we provide in each of the benchmarks will follow the given order for understanding and extending the thought process:

1. **Tests:** We provide the test configurations for the benchmarking.
2. **Insights**:
   a. Serial Comparison – Comparison for the serial code timings of both Rust and C++.
   b. *Speed up* - A comparison of the speeds between the libraries, and how they scale according to threads.
   c. *Time difference* – This will provide as to how efficiently are mapping to cores done, and what is the actual timing difference between the parallelism constructs.
   d. *Problems Encountered* - A description of problems that were faced in creating the benchmarks or generating the data.
3. **Deeper insights** (depending on what is needed):
   a. *Performance analysis using* `perf` - To see how the program behaves with the caches and the cores, perf gives us insight that helps us understand the behavior seen above.
   b. *Source code analysis* - By looking at the source code in the repository, we can explain most of the behavior by looking art overhead that is created due to the boilerplate code that is added by the parallelism construct.

In each of the benchmarks, we will first show the difference through benchmarked timings, giving an explanation for the behavior. As mentioned, we will provide a few graphs to identify and create insights. Using those insights, we will delve deeper and using source code analysis and other tools, we will attempt to explain why that behavior occurred.

We believe that a structured way of understanding and showing information, helps the reader not only grasp the information quickly, but also see how we did our experiments to suggest improvements and follow the methodology to perform their own experiments.

We used a best of three runs for reach benchmark, while building executables with command line options to run difference configurations seamlessly. We used shell scripts to generate the data. The generated data was consumed and formatted using Python script.

Mandelbrot

Tests

We have chosen two implementations for parallelizing Mandelbrot over threads as shown in the table below:

Problem	Types of parallel Implementation	Views	Image size
Mandelbrot	Threads parallel over rows of the image Threads parallel over pixels of the image	0	2048 x 2048
		1	4096 x 4096
		2	
		3	
		4	
		5	8192 x 8192
		6	4096 x 4096

Serial Code Comparison

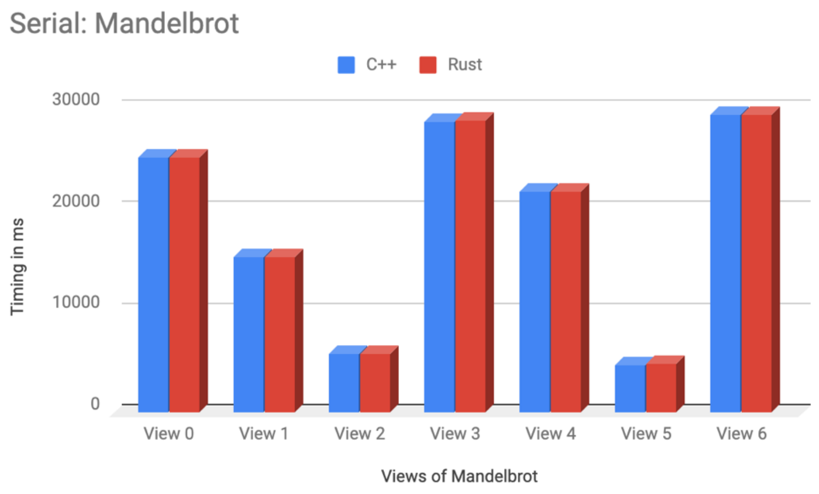


Fig: Comparison of Serial versions of Mandelbrot in C++ and Rust

Analysis: C++ code beats the rust implementation for all the different views of Mandelbrot, but only marginally. We were able to achieve this equality in the baseline code of the two programs after examining the object dump and eliminating additional assembly instructions generated by the Complex32 structure in the Rust implementation.

Speedup

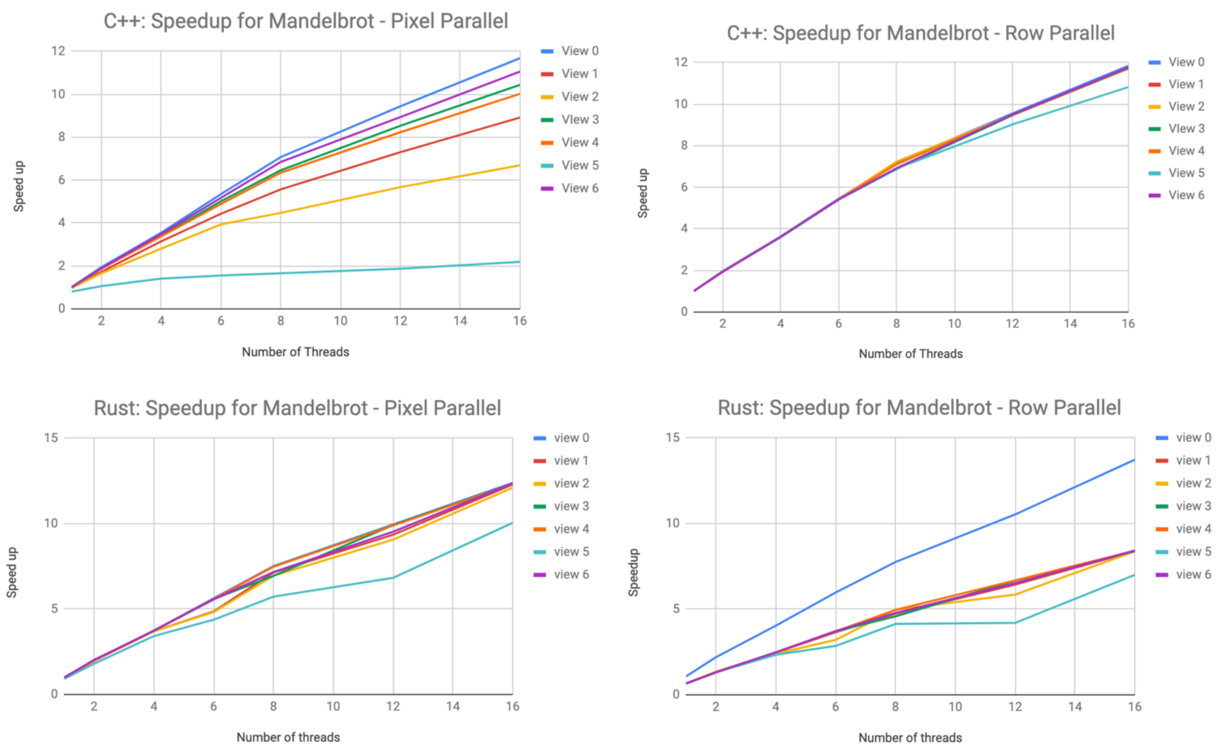


Figure: Speedup obtained on different views of Mandelbrot implemented using OpenMP for C++ and Rayon in Rust for (a) Threads running parallel over the pixels (b) Threads running parallel over rows of the image

Analysis: It is evident that Rust does not perform all that poorly in comparison to C++. The speedup obtained for Multithreaded version of Rust with threads running parallel over the rows of the image scales very similar to C++ version of Mandelbrot with threads running parallel over the pixels.

Time difference:

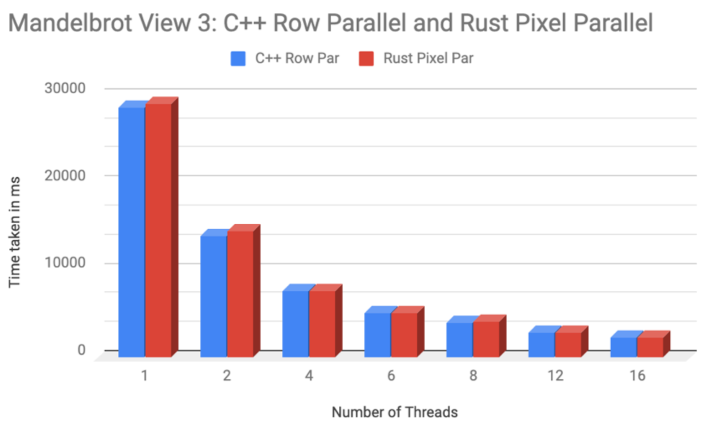


Figure 3: Comparison of time taken by multithreaded version of C++ and Rust Code for Mandelbrot

The above figure demonstrates the difference in execution times for C++ and Rust versions of Parallel Mandelbrot implementation for View 3 with threads being parallel over the pixel of the image in Rust and threads being parallel over rows in C++. The time taken by the Rust implementation of Mandelbrot code lies within 1% of the time taken by C++ implementation.

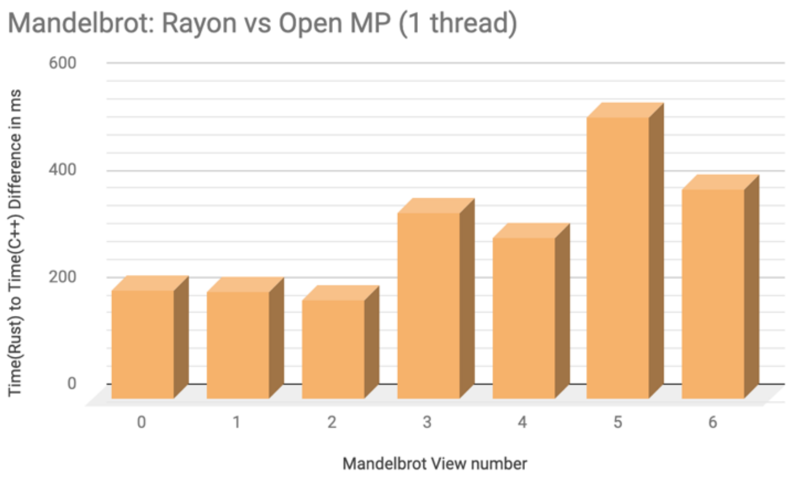


Figure 4: Difference in time taken by the single thread version of Mandelbrot in Rust and C++

Analysis: From the timing information showed in the graph above, it is observed that for a single thread version of Mandelbrot, Rust takes more time than the C++ version (Indicated by the positive bar graphs) across all views of the image. This result shows that Rust introduces significant overhead to the boilerplate code.

Problems encountered

At first, we weren't able to achieve the same speeds in Rust as in C++ even for the serial code. This leaked into the parallel code. For view 1 - 4096x4096, Rust took 19319.043 *ms* and C++ took 18813.754 *ms*, approximately 500 *ms* more than C++. We looked into the assembly, where we found that Rust has 4 extra instructions for Complex32 structure it internally uses. After eliminating it to perform the calculations similar to C++, we got the same speed as C++ (18758.684 *ms*).

Perf Insights

Analysis through perf reveals that the inner loop has different assembly order than C++, which attributes to the minute speed difference. There is overhead of splitting jobs that exists in the one thread version, but most difference is made by the instructions in the inner loop.

Although the speedup obtained with Rust is a more non-linear when compared to C++. After some analysis we found that this is due to the dynamic scheduling and splitting in Rust. Since the problem of parallelizing Mandelbrot is embarrassing parallel, we do not observe the benefits that Rust provides in making the functionality of code data-race safe.

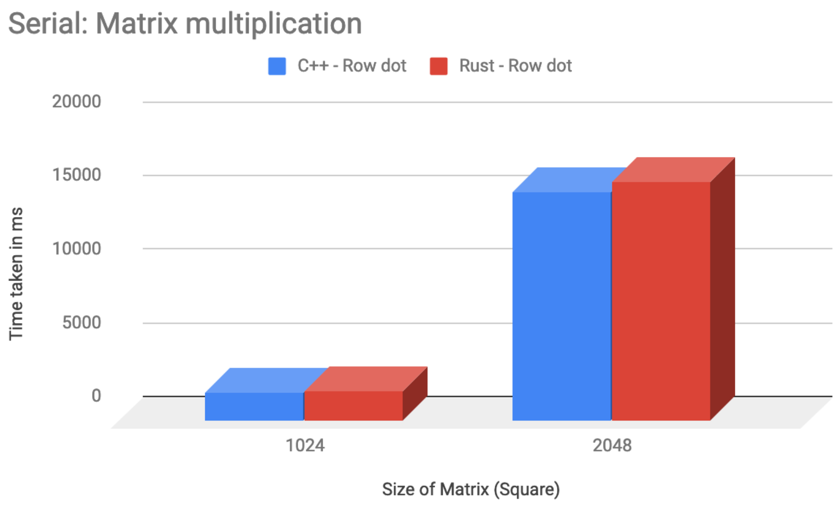
Matrix Multiplication

Tests

There are two quantitative checks with two different implementations for the matrix multiplication:

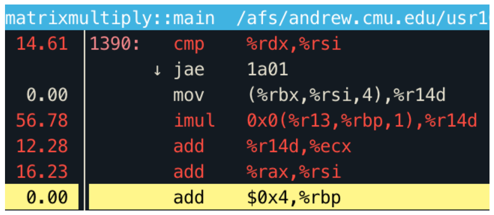
Problem	Size of matrix	Type of parallelism
Matrix Multiplication	1024 x 1024	C++ - Outer loop Par
		C++ - Row dot
		Rust - Row dot
	2048 x 2048	C++ - Outer loop Par
		C++ - Row dot
		Rust - Row dot

Serial code comparison

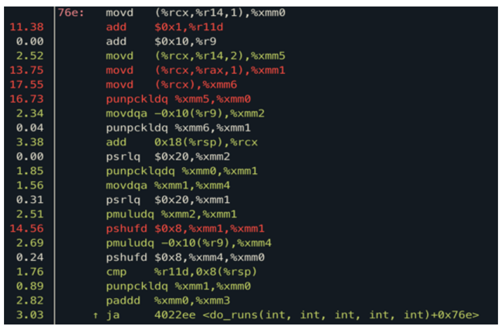


Analysis: Though very close, C++ still beats Rust in timings for both cases, but only marginally. We will see further if its effects are present in the parallel code. Surprisingly, when we observed the assembly, Rust did not vectorize the innermost loop (C++ did). We tried experiments to vectorize without intrusion, but were unsuccessful.

Screenshots of serial innermost function



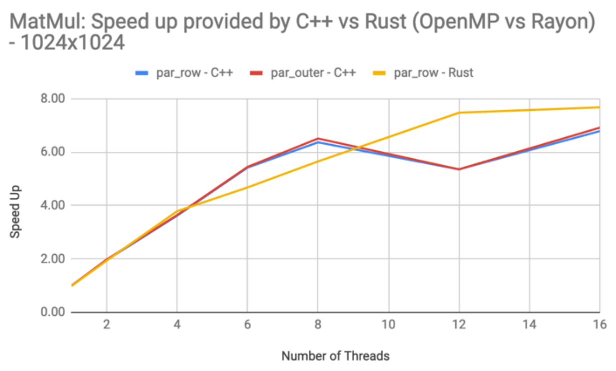
```
matrixmultiply::main  /afs/andrew.cmu.edu/usr1
 14.61 │1390:   cmp    %rdx,%rsi
       │      ↓ jae    1a01
  0.00 │        mov    (%rbx,%rsi,4),%r14d
 56.78 │        imul   0x0(%r13,%rbp,1),%r14d
 12.28 │        add    %r14d,%ecx
 16.23 │        add    %rax,%rsi
  0.00 │        add    $0x4,%rbp
```

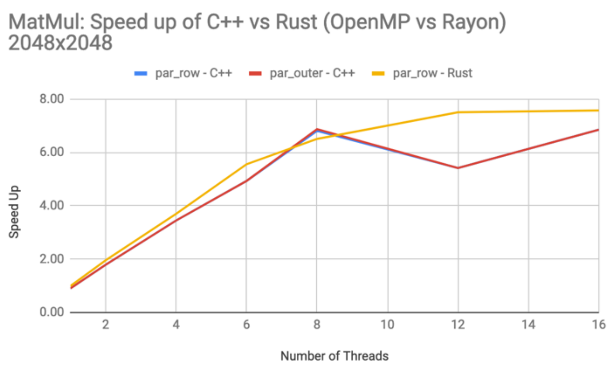


```
       │76e:   movd   (%rcx,%r14,1),%xmm0
 11.38 │       add    $0x1,%r11d
  0.00 │       add    $0x10,%r9
  2.52 │       movd   (%rcx,%r14,2),%xmm5
 13.75 │       movd   (%rcx,%rax,1),%xmm1
 17.55 │       movd   (%rcx),%xmm6
 16.73 │       punpckldq %xmm5,%xmm0
  2.34 │       movdqa -0x10(%r9),%xmm2
  0.04 │       punpckldq %xmm6,%xmm1
  3.38 │       add    0x18(%rsp),%rcx
  0.00 │       psrlq  $0x20,%xmm2
  1.85 │       punpcklqdq %xmm0,%xmm1
  1.56 │       movdqa %xmm1,%xmm4
  0.31 │       psrlq  $0x20,%xmm1
  2.51 │       pmuludq %xmm2,%xmm1
 14.56 │       pshufd $0x8,%xmm1,%xmm1
  2.69 │       pmuludq -0x10(%r9),%xmm4
  0.24 │       pshufd $0x8,%xmm4,%xmm0
  1.76 │       cmp    %r11d,0x8(%rsp)
  0.89 │       punpckldq %xmm1,%xmm0
  2.82 │       paddd  %xmm0,%xmm3
  3.03 │     ↑ ja     4022ee <do_runs(int, int, int, int, int)+0x76e>
```

Fig. L: Rust, R: C++

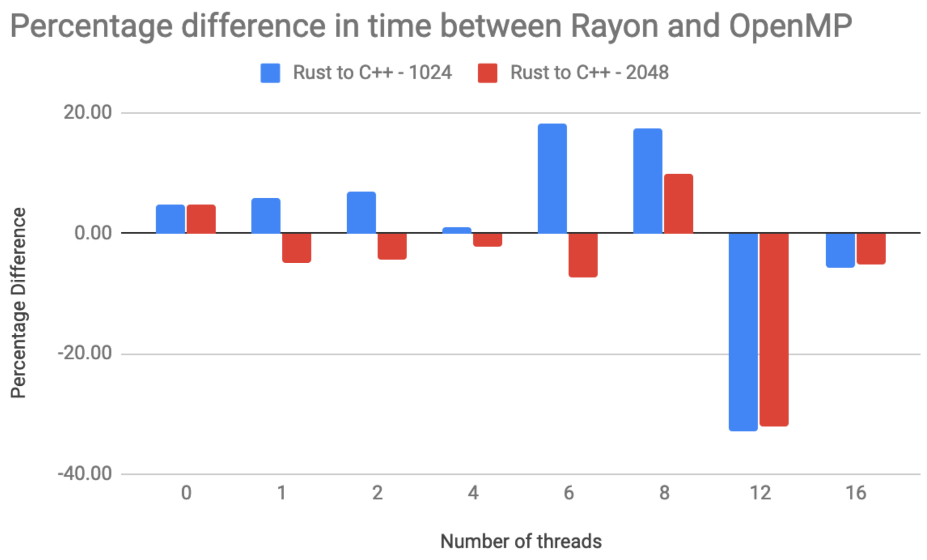
Speed up





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

Time Difference: Positive – C++ is better; Negative – Rust is better



Analysis:
C++ is faster for all cases of threads for size 1024. This is surely attributed not having the splitting overhead, and the vectorization of the innermost loop. But Rayon performs better for all cases of threads for size 2048. Rayon performs better when the number of elements is larger, as the sharing of cache actually helps work stealing algorithms. Also, the dip at 12 threads is due to the compute bound issue. Vector intrinsic reaches a compute bound on this system at 8 threads.

Problems Encountered

Rayon allows multiple ways of writing the same code. Here we show two different ways of writing the same function:

Though both should compile to the same, we observed the assembly and found some extra offset instructions in the less optimized code. This was only first observed when we got a much lower speed up than what we are getting now.

Source Code analysis

As mentioned before, the advantage Rust or Rayon has on C++ is the lack of vectorization. It reaches the compute bound much later. The advantageous part here is that we have decided the chunks or division of the jobs in Rayon. In the source code, if we use `par_chunks()` methods, we have decided the smallest splits with the size mentioned for the method. This allows for better parallelism because there is no fork-join overhead.

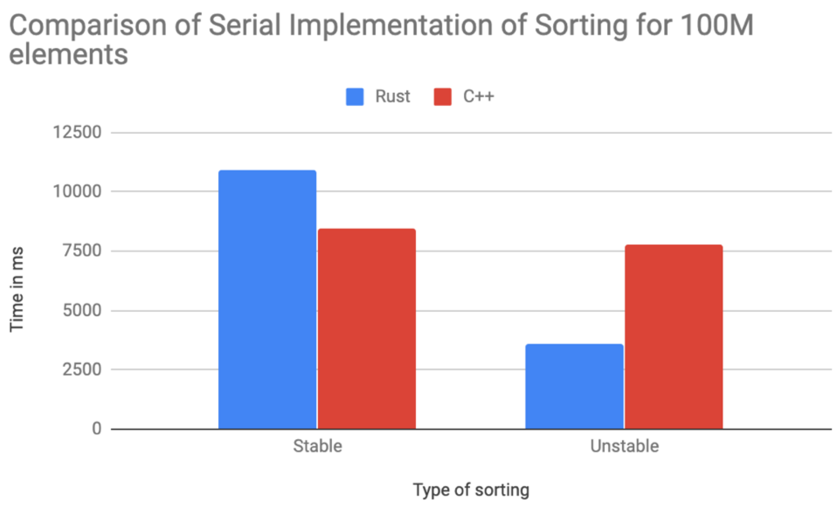
Unstable-Stable Sorting

Tests

Though in our writeup we had mentioned we will be comparing quicksort, the implementation was straight forward, instead we now compared the stable (merge-sort) and unstable (quicksort) sort functionality of Rayon and GNU-parallel sort provided as an extension of OpenMP. In this benchmark, we have sorted 1M, 10M, and 100M elements ascendingly from randomly generated values.

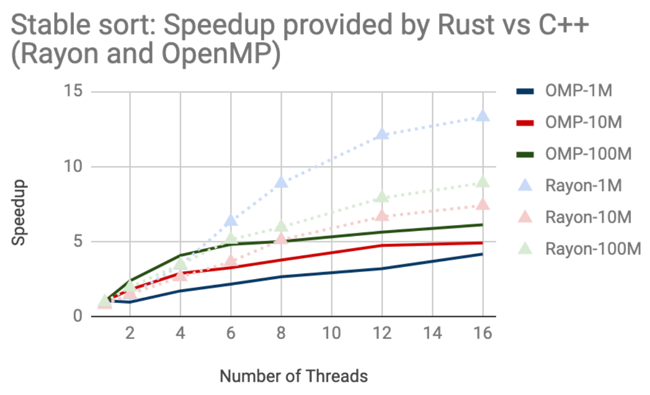
Problem	Function	Number of elements
Sorting	Unstable sorting (quick-sort)	1M
		10M
		100M
	Stable sorting (merge-sort)	1M
		10M
		100M

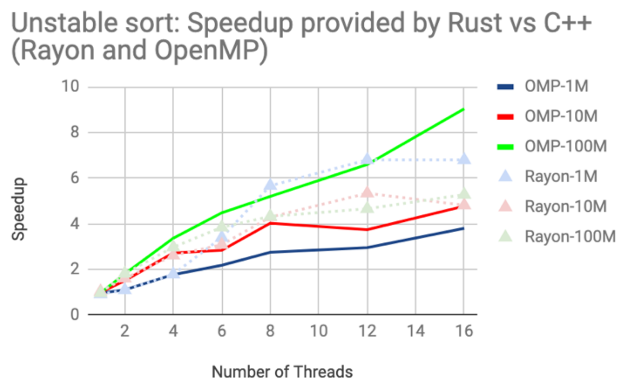
Serial Comparison



Analysis: Serial implementation of Stable sort in Rust takes significantly more time than the serial implementation in C++. After examining the assembly dump of these two implementations we found that Rust uses unoptimized assembly instructions like `cmp` and `mov` while the C++ code generates instructions like `cmovb` and `cmovab`. Also from analyzing the source code of `std::sort` and Rust's implementation of unstable sort, we found that Rust implements Pattern-defeating quicksort (*pdqsort*) is a novel sorting algorithm that combines the fast average case of randomized quicksort with the fast worst case of heapsort. Hence Rust implementation runs way faster than the C++ one.

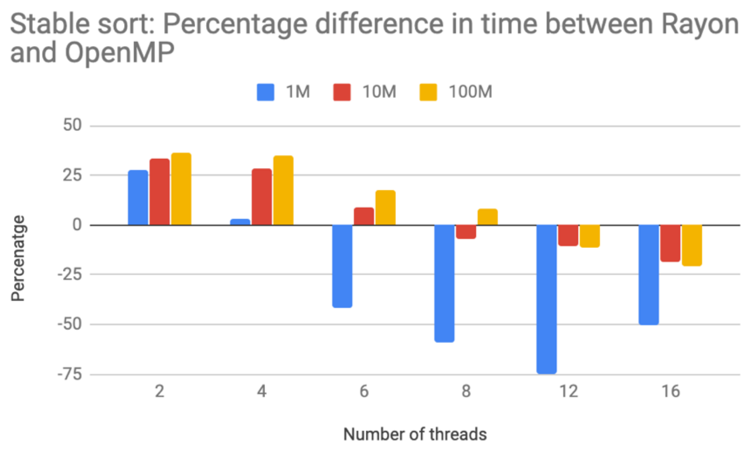
Speedup





From the Speedup graphs below, we see that Stable implementation of Sorting gives an evenly rising curve and shows that rust scales better than C++ with increase in number of threads. For the unstable version of sorting, we observe a non-uniformly rising curve and C++ implementation for larger elements performs better than Rayon's implementation. This can be attributed to the choice of magic numbers in the unstable sorting algorithm of Rayon.

Time difference



From the percentage difference in time graph for 1M, 10M and 100M elements, we observed a positive difference in Rust and C++ when the number of threads were small, implying C++ version is faster than the Rust implementation. As we increased the number of threads, we observe a negative difference in time, implying Rust gives better performance.

<u>Source Code Insights</u>

Observing the `mergesort.rs` file, we notice that they have tried 3 different merge-sort methods (Swap elements, insert and shift, insert hole-copy). After benchmarking each of the methods, they found that the most optimized version was the third one and stuck to it. This explains the vast improvement that stable sorting shows in the graphs. Also, it is mentioned in many blogs that GCC sorting isn't best optimized for multithreading.

Unstable sorting has been verbatim copied from its serial version, with the addition of recursion using Rayon. Hence, it is still using a pattern defeating Quicksort (PDQ) [9]. Also, it has the `block_size` equal to the magic number <u>128</u>, which means that partition of quicksort will go until 128 elements per block. This can have consequences when the array size becomes much larger. And this issue is observed in the speed up graphs.

Reduction Sum

Tests

We have chosen two qualitative and three quantitative tests. There will be a total of six configurations, which are given by the table below:

Problem	Function	Number of elements
Reduction	Sum of elements (f64/double precision)	500M
		1B
		2B
	Sum of squares of elements (f64/double precision)	500M
		1B
		2B

Serial code comparison

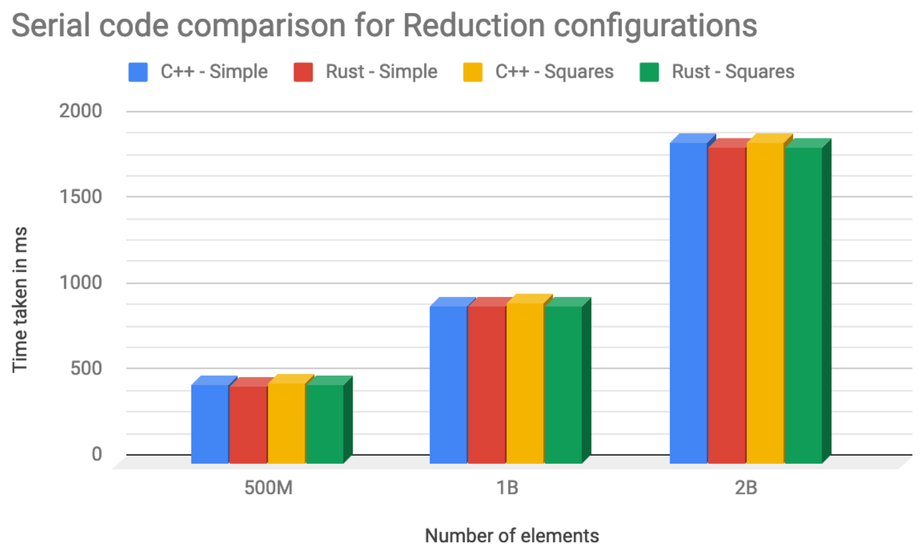


Fig. Serial code timing for each configuration

We can observe from the graph that both the C++ code and Rust code perform equally well for each of the case (although Rust performs minutely better in all cases). The optimizations were seen to be present in both the codes. This code was next parallelized with the parallel constructs of each of the libraries.

Speed Up

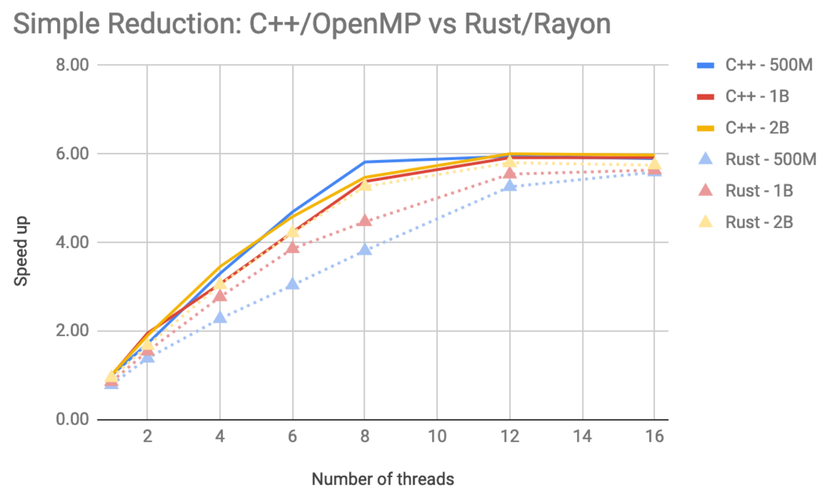


Fig. Simple reduction OpenMP vs Rayon

Analysis: You can see that Rayon performs worse than OpenMP is all cases. Comparing the 500M curve with 2B curve, we can clearly see that there some extra overhead occurring in Rayon. Having a larger number of elements in 2B, the overhead has a lower significance since the computation becomes larger chunk of work. This is happening due to the indecision of the splitting in Rayon. The granularity to which it must split is decided by Rayon itself, which can cause over-splitting. A larger workload seems to balance it out.

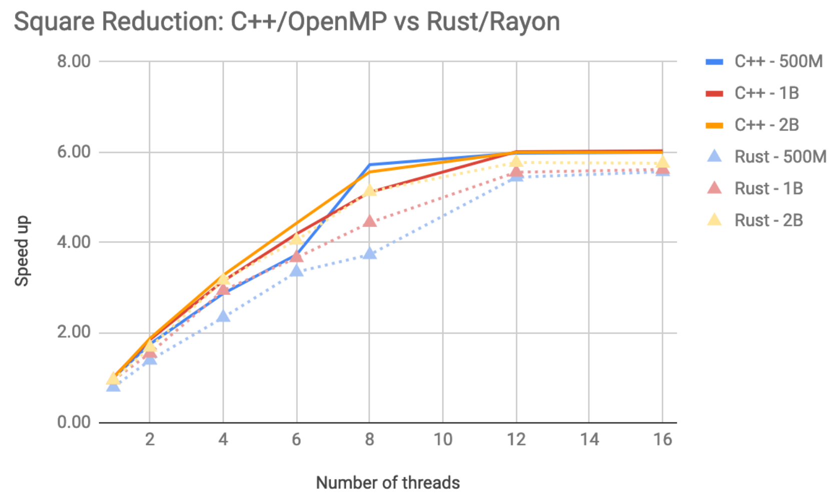
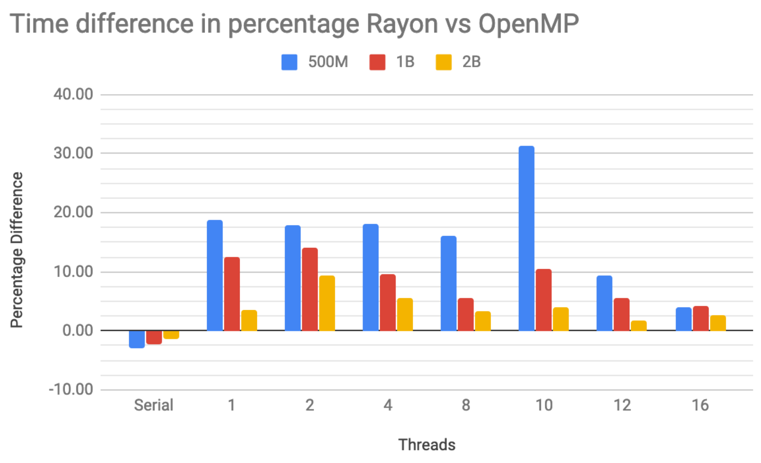


Fig. Square reduction OpenMP vs Rayon

Analysis: This shows the same behavior as the previous configuration. Notice the 500M graph and 2B graph for Rust, it reiterates the problem of poorer splitting criterion for lower number of elements.

Time Difference: Positive – C++ is better; Negative – Rust is better



Analysis: Though the serial code performed better for Rust, Rayon code performs worse than OpenMP. You can notice the same behavior that when the number of elements is larger, Rayon almost catches up with OpenMP. But still the difference is significant for a lower number of elements.

Perf insights

We recorded the cycles taken by 10 threads – 500M configuration on both Rust and C++. We noticed the large amount of instructions which is called to split the data into separate chunks before the reduction in Rayon. We timed the recursion until the first reduction operation occurs, and it turned out to be just 40ms. We suspected the compute bound nature of the problem and noticed that this is the opposite behavior of what we observed in Matrix multiplication. LLVM has optimized the reduction using SIMD, while GCC hadn't, which we didn't observe in the non-parallel version. The SIMD reaches compute bound for each core earlier as it has continuous dependent instructions, attributing the difference to GCC vs clang-LLVM.

Source Code

To understand the fineness of splitting, we looked into the `splitter.rs` file in their repository, they have mentioned that "By using `iter::split`, Rayon will split the range until it has enough work, to feed the CPU cores, then give us the resulting sub-ranges". We checked the `split_producer.rs`, and we do see that the split takes into account the `len()` of the array. It has a boilerplate code which defines the smallest size of the split dynamically as a numerical multiple of the number of cores available.

This has been also mentioned in the soon-to-be addition in `rayon-adaptative`. Due to varying depth in splits, rayon-adaptive will allow changing the policy (static, dynamic, different dynamic issuing). It is going to be added soon and will surely result in timings matching OpenMP.

Conclusions

In our experiments, Rayon performed as well as OpenMP in cases where the underlying algorithm or compiler gave an advantage or edge. The downfalls of Rayon are the under-optimized computing function, cost of creating splits of work, and stealing when compared to a possible static scheduling. Rayon performed better in Sorting, larger matrices, but in all other benchmarks, OpenMP had the upper hand. But at no point we encountered a segmentation fault in Rayon due to its guaranteed data safety, but we did have those issues in OpenMP.

Rayon is still under development, having reached only version 1.0.3, while OpenMP has been there for a very longtime. It is possible that Rayon can achieve as good or better performance. Rayon-adaptive is going to be merged, which will allow changing the underlying policies for the splitting. Having the advantage of being both data safe and language with syntactic sugars, Rust with Rayon might soon reach OpenMP performance. Another important modification would be changing the backend to GCC from Clang-LLVM, as it is proven with papers that generally GCC gives more optimized code.

We have provided the code and data in this Github Repository:
https://github.com/trsupradeep/15618-project

References

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

Division of Work

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