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

Authors:

Nishal Ancelette Pereira <[napereir@andrew.cmu.edu](mailto:napereir@andrew.cmu.edu)>

Supradeep Rangarajan <[strangar@andrew.cmu.edu](mailto:strangar@andrew.cmu.edu)>

Summary

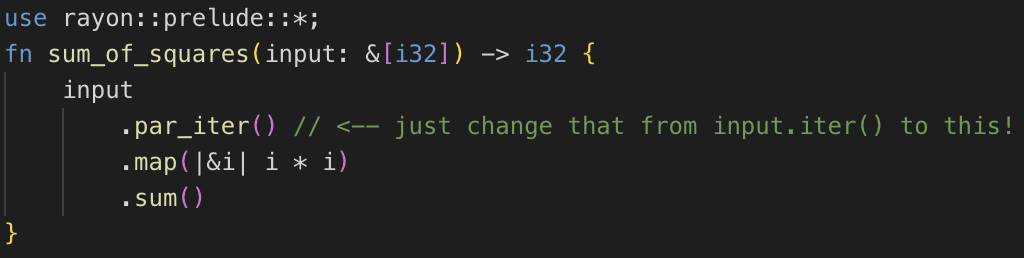
At this point, we have compared and performed preliminary analysis on the multithreading performance on a simple load imbalance workload (Mandelbrot) utilizing two multithreading libraries, Rayon and Crossbeam, which are implemented specifically for Rust to that of OpenMP implementation for C++.

Background

What is Rust?

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.



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 subheading.

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.
   1. 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.
   2. 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()
   3. 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.
   1. 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.
   2. 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.
   3. 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

Comparison of Rayon and OpenMP

Motivation for the project

OpenMP extension to C++ provides a very easy, straightforward way of parallelism.

Approach

To compare two different parallelism libraries of different languages.

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 chose:

1. Mandelbrot –
2. Matrix Multiplication –
3. Unstable-Stable Sorting –
4. Reduction – OpenMP added a reduction feature

Use best serial code 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 in Rayon will lead to bad parallelism while scaling to cores. Hence, we must either observe the assembly code of the compiled executable or try to see how to which written code is provides best optimized assembly.

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 assembly commands added to its flavor.

Re-optimization

Rayon provides a very simple way of parallelism, but as mentioned before different ways of writing the same code will give us different speeds due to the way the compiler is optimized.

Results

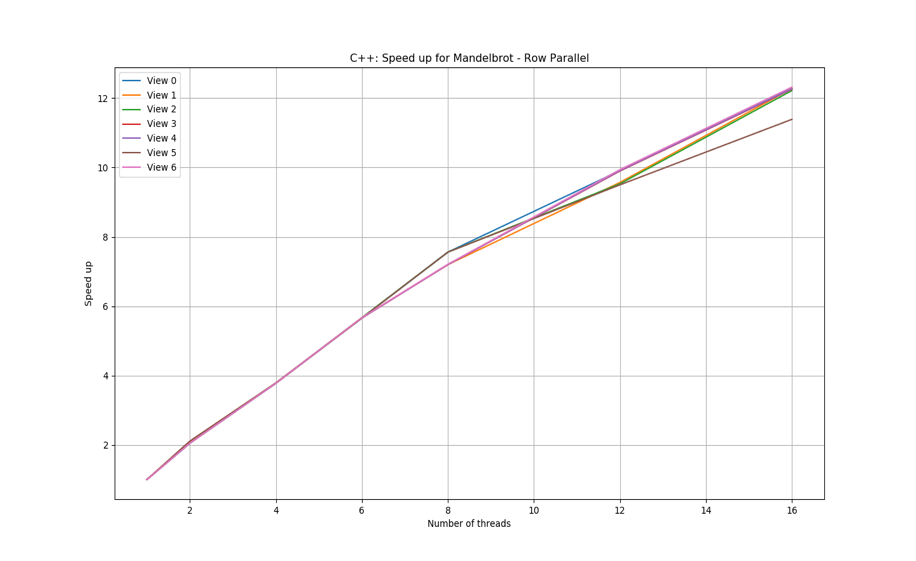
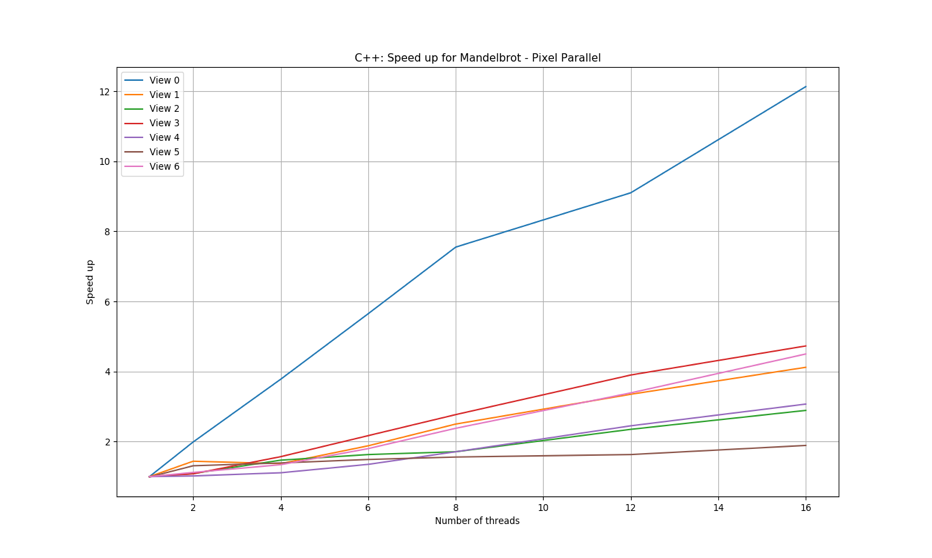


Figure 1: Speedup obtained on different views of Mandelbrot implemented using OpenMP for C++ for (a) Threads running parallel over the pixels (b) Threads running parallel over rows of the image

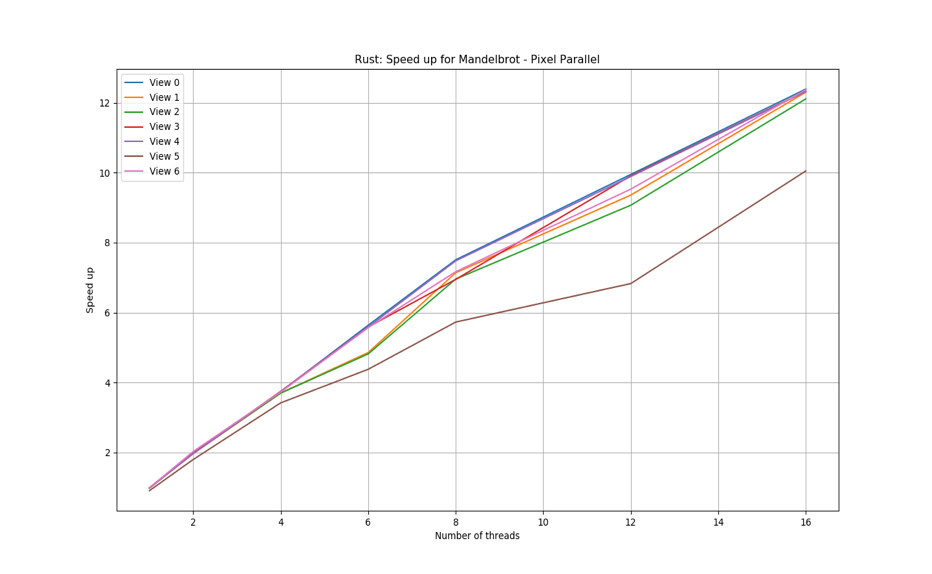
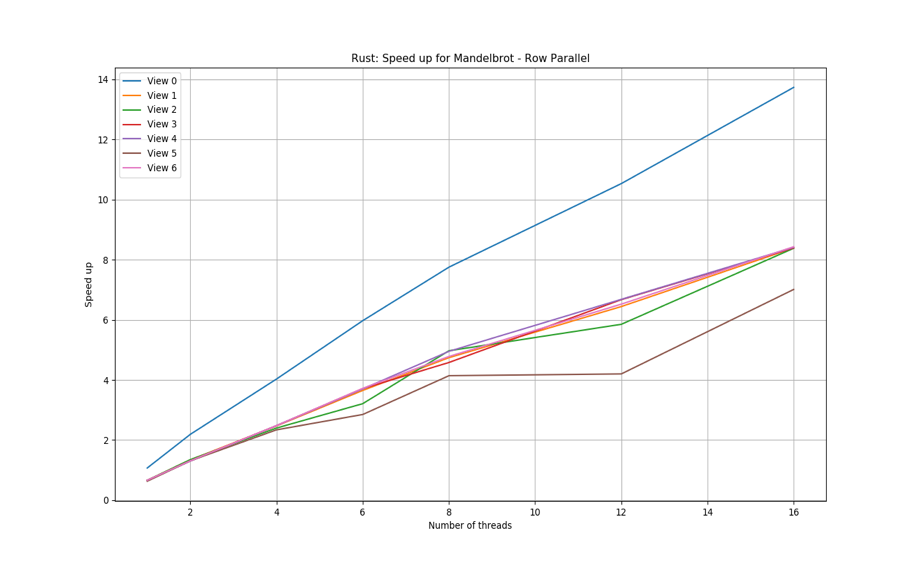


Figure 2: Speedup obtained on different views of Mandelbrot implemented using Rayon in Rust for (a) Threads running parallel over the pixels (b) Threads running parallel over rows of the image

Figure 1 and Figure 2 show the speedup that was obtained on different views of Mandelbrot for threads ranging from 2 to 16 for both C++ and Rust implementations. It is evident that Rust does not perform all that poorly in comparison to C++ version of Multithreading. 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. 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 in Rust. The threads in Rust steal work from other thread when their load is idle. 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.

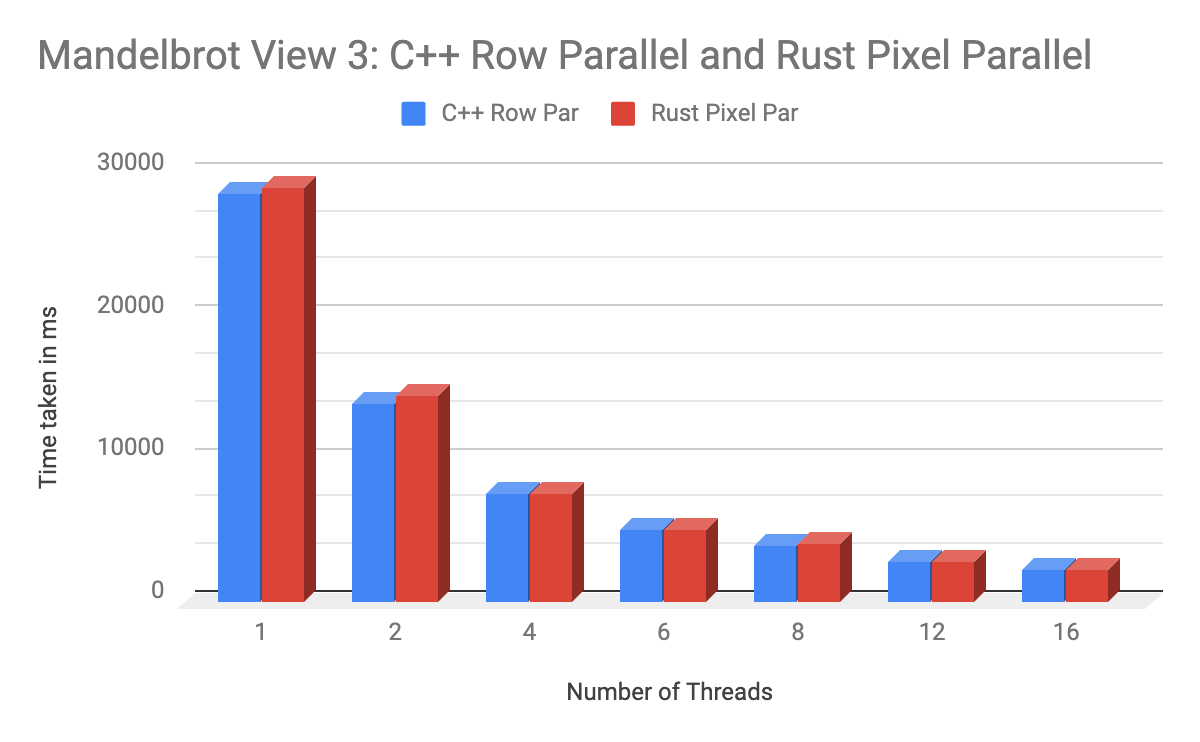


Figure 3: Comparison of time taken by multithreaded version of C++ and Rust Code for Mandelbrot

Figure 3 further iterates the points highlighted in figure 1 and figure 2. The time taken by the Rust implementation of Mandelbrot code lies within 1% of the time taken by C++ implementation.

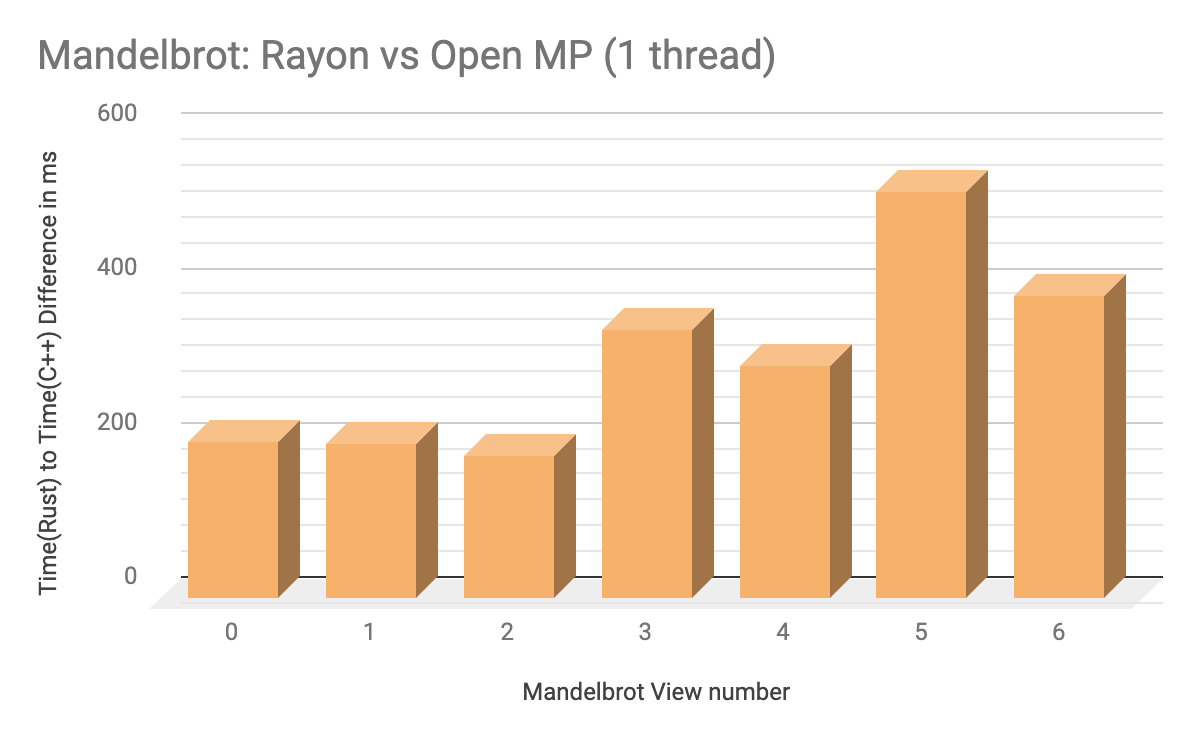


Figure 4: Difference in time taken by single threaded version of Mandelbrot in Rust and C++

Figure 4 analyzes the overhead introduced in spawning a single thread in both Rust and C++. From the timing information showed in the graph, it is observed that for a single threaded 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.

References

[1] Feature Request: OpenMP/TBB like Parallel For Loops, [*https://github.com/rust-lang/rfcs/issues/859*](https://github.com/rust-lang/rfcs/issues/859)

[2] Rayon, [*https://github.com/rayon-rs/rayon*](https://github.com/rayon-rs/rayon)

[3] Rayon documentation, [*https://docs.rs/rayon/1.0.3/rayon/*](https://docs.rs/rayon/1.0.3/rayon/)

[4] Rust Q-A, [*https://www.reddit.com/r/rust/*](https://www.reddit.com/r/rust/)

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.