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

Project Checkpoint 2

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

In Matrix multiplication, Rayon performs almost as well as OpenMP with Rayon giving a better speed up climb with increasing number of threads, hinting on load division and the difference in parallelism. Rayon performed better for a larger sized matrix, but C++ outperformed Rayon for a smaller sized matrix.

Progress

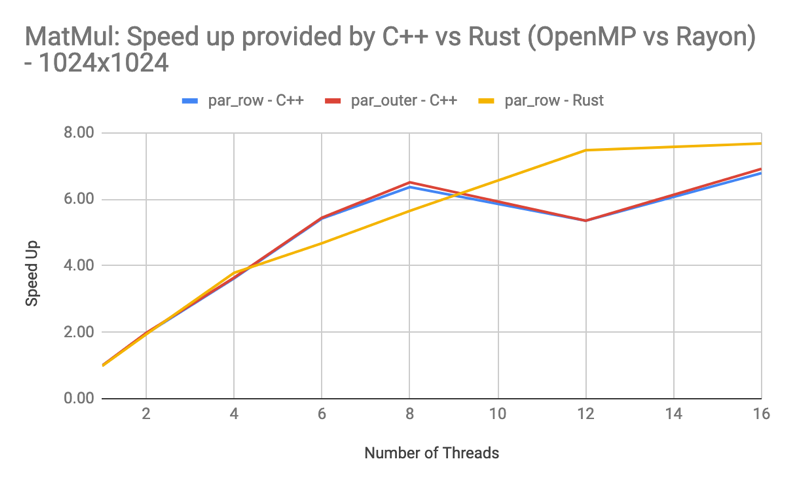
We have successfully implemented and benchmarked the multithread version of Matrix Multiplication and Stable-Unstable sorting for both Rust/Rayon and C++/OpenMP. Serial implementations were matched first to give fairness in comparison.

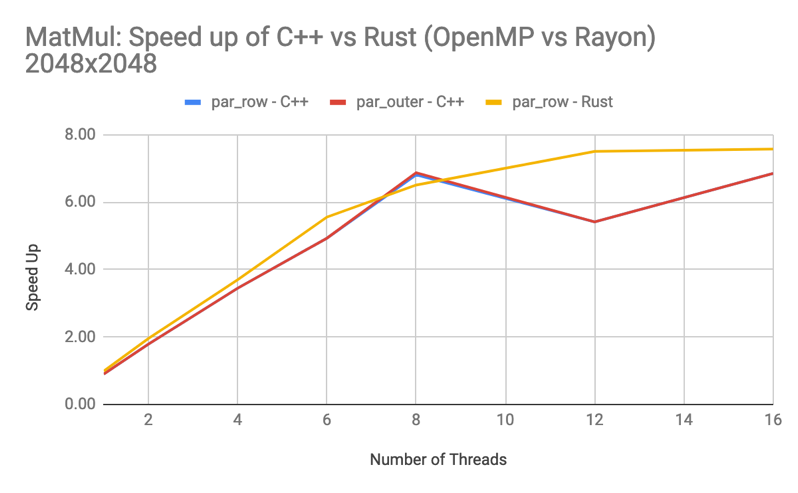
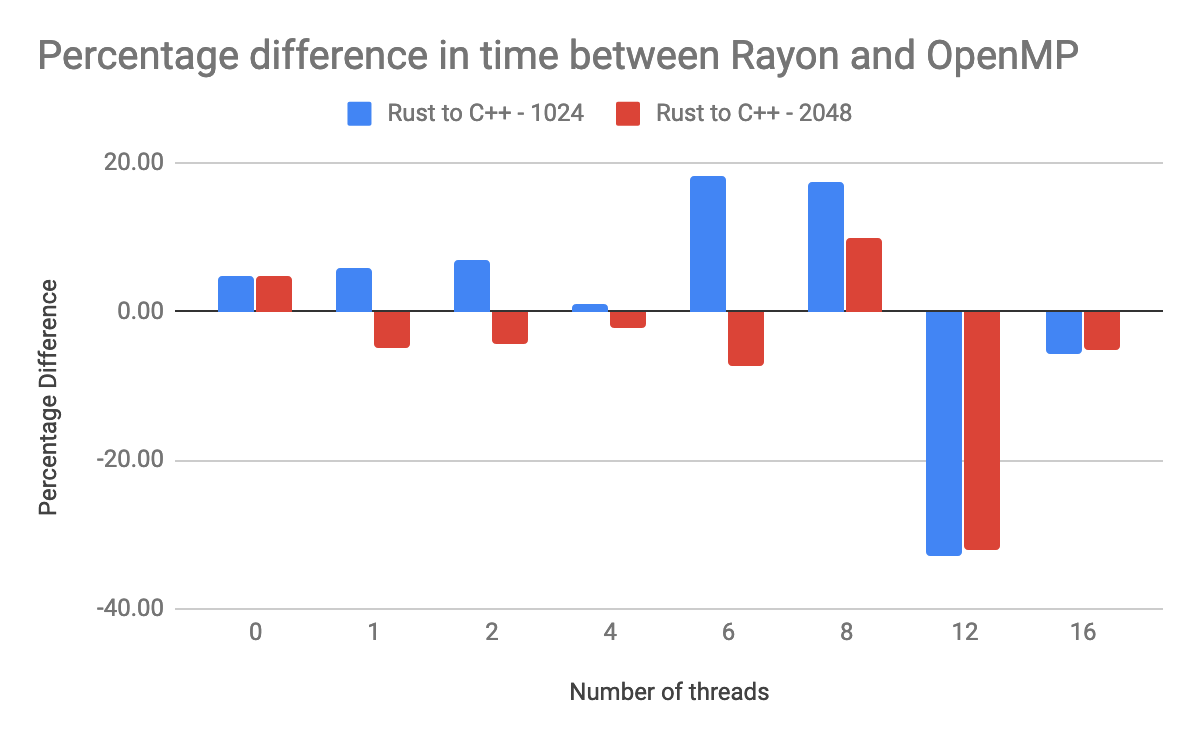
Similar to Mandelbrot experiment, we have used the same thread setup and analysis style so that readers can easily understand the information provided. In both the cases, we have used to single thread to understand boiler plate latency. In both cases, we have measured all possible parallel algorithms for the problem and measured timings for all of them.

Schedule

The progress is as per schedule with the completion of Matrix Multiplication Benchmark and Stable-Unstable sorting benchmark. We have finished benchmarking Matrix Multiplication and Stable-unstable Sorting. We have benchmarks for different matrix sizes and array elements size for both Rayon and OpenMP. As mentioned previously we are going to perform analysis this week with addition of one more benchmark.

We are not performing the Fibonacci benchmark as we have already implemented ahead of schedule in which we are facing stack overflow issues. We will instead compare the reduction functionality between OpenMP and Rayon.

Preliminary Results



Matrix Multiplication

A task which can be parallelized using a row level parallelism. This benchmark is compute bound so it would show us differences in communication overheads and boilerplate latency. In this case, we selected square matrices of 1024x1024 and 2048x2048 as they take a comparable time which eliminates the effect of turbo boosting of intel processors.

A new finding in this test was that Rayon provided different ways of parallelizing the same task. Different iterator movements which were possible were timed to see which performs the best. They all performed almost the same but one such iteration performs really well.

Another finding is that Rayon has a good scaling over increasing number of threads. You can see a dip in performance beyond 8 threads in C++. This could be because of well usage of CPU and being compute bound at 8 threads, where hyperthreading doesn’t help. But Rayon on the other hand, has an evenly rising speed up.

In the last graph, we have plotted the difference between Rust and C++ timings as a percentage. A positive percentage means that rust is slower than C++ and negative means vice-versa. We can see that for 1024 sized matrix, Rayon performs slightly slower, but it performs better for a larger 2048 sized matrix. We need to understand the load balancing mechanisms for this. For 12 threads, Rayon performs incredibly better than C++, which means that it is eliminating some kind of false sharing.

Stable-Unstable Sorting

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.

Challenges

Different ways of writing the same parallel task:

In rust, due to the iterators and the way they work, we can write the same parallelism task in multiple ways, which give different performance on profiling. The compiler, as its still young, knows to optimize some styles of code really well.

Solution: We had to profile every such way of writing that code. Each of the version was around 3% slower than the best. A parallel iterator over both array C and array A, such that you get a slice of each, provides the best speed for the array multiplication. The parallel iterator changed to serial iterator gives the best serial code too.

Failure of operf on GHC machines:

We are still searching for alternatives to profile and get some low-level information for these benchmarks. However, we were unable to do so on the GHC machines because of permission issues.

Solution: We are planning to use *perf* to profile and see the information we can extract from it. We will still look for more resources to generate more information for the comparison.

Issues and Concerns

Fibonacci was selected as a benchmark to measure the communication that undergoes in each of the parallelism tasks. But to do Fibonacci in a parallel sense, we need to do it recursively. But while we tested this both for C++ and Rust, we always kept having stack overflows for a higher number. We have to change the benchmark for this case, so we are planning to compare other functionalities like Reduction Sum.

Presentation

For the final presentation, we will be presenting charts and graphs that provide interesting insights for Comparison of Multithreading. Alternatively, we can also demo parts of our code. We will also show code snippets to give insight on how different and effective is coding in Rust.