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

Nishal Ancelette Pereira (napereir@andrew.cmu.edu), Supradeep Rangarajan (strangar@andrew.cmu.edu)
Electrical and Computer Engineering, Carnegie Mellon University

Abstract

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.

Introduction

- While decomposing a given algorithm into parallel workloads, developers need to be careful about new kinds of error including deadlock, livelock and data-races.
- Rust provides an abstraction with zero-cost, thus becoming highly popular among industries and developers
- Rayon (data parallelism library for Rust) allows simple conversion from serial to parallel code like OpenMP. Rust does not allow thread unsafe code to compile making Rust more scalable than C++.
- 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 features like auto-vectorization of the code unlike OpenMP

Serial

```
int sum_of_squares(int* array, int len) {
    int sum = 0;
    for (int i; i < len; i++) {
        sum += array[i] * array[i];
    }
}
```

C++ → Parallel

```
int sum_of_squares(int* array, int len) {
    int sum = 0;
#pragma omp parallel for reduction(+:sum)
    for (int i; i < len; i++) {
        sum += array[i] * array[i];
    }
}
```

Serial

```
fn sum_of_squares(input: &[i32]) -> i32 {
    input.iter()
        .map(|&i| i * i)
        .sum()
}
```

Rust → Parallel

```
fn sum_of_squares(input: &[i32]) -> i32 {
    input.par_iter()
        .map(|&i| i * i)
        .sum()
}
```

Approach

We aimed to compared parallelism libraries (Rayon and OpenMP) in Rust and C++, respectively, keeping the following objectives in mind:

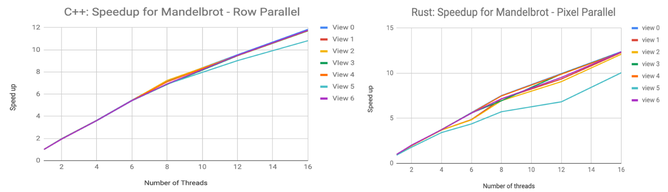
1. Ease of use 2. Safe Concurrency 3. Speed 4. Handling of dynamic workload

With the above objectives in mind, we choose the following simpler but fundamental benchmarks:

- **Mandelbrot:** An imbalanced workload to analyze scheduling amongst threads
- **Matrix Multiplication:** A compute bound problem
- **Unstable-Stable sorting**: Built-in sorting methods of these libraries
- **Reduction Sum**: A commonly used operation in Geometry and Physics that is memory bound.

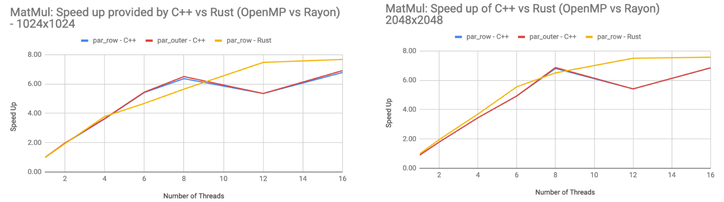
Results

Mandelbrot:



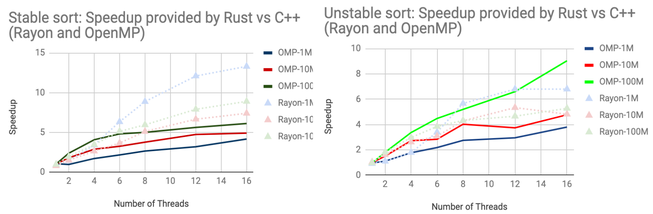
- OpenMP performs marginally better than Rust.
- The loop in `mandel_iter` is better optimized in C++ than Rust.
- Large overhead of splitting jobs in the single thread version.
- C++ used dynamic scheduling to scale in accordance to Rust

Matrix Multiplication:



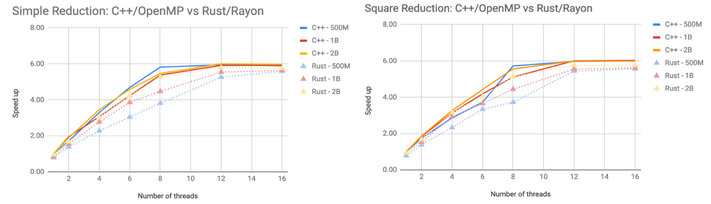
- OpenMP performs better at 8 threads, Rayon performs better for higher threads for 1024 sized matrix.
- Rayon performs better for 2048 size and has smoother scaling.
- Rayon has vector intrinsic while C++ does not.
- Rayon uses `par_chunks()` which decides statically the smallest split of data.

Unstable-Stable sorting:



- Rust scales better than C++ for stable sorting with increase in number of threads.
- Rayon benchmarked three mergesort techniques and implemented the best optimized.
- Rayon uses Pattern Defeating Quicksort (PDQ) which is faster than GCC implementation.
- Rust/Rayon uses block size 128, where as, C++/OpenMP uses 4 for smallest partition.

Reduction Sum:



- Rayon performs worse than OpenMP in all cases.
- The 500M curve shows the overhead in Rayon attributed to splitting of work.
- Under-optimized reduction function by Clang-LLVM compared to that of GCC.
- Rayon's splits are made until it is enough to feed the CPUs.

Conclusions

Key Remarks:

- From experimental results, 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 for sorting and multiplication of larger matrices, but in all other benchmarks, OpenMP had the upper hand.
- No segmentation faults were faced using Rayon because of guaranteed data safety.

Insights:

- 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 soon achieve as good or better performance.
- Rayon-adaptive is going to be merged, which will allow changing the underlying policies for the splitting.
- Another important modification for Rust would be changing the backend to *GCC* from *Clang-LLVM*, as generally GCC gives more optimized code.

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] Rayon adaptive, *https://github.com/wagnerf42/rayon-adaptive*
[5] OpenMP official documentation, *https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf*
[6] Cargo Book, *https://doc.rust-lang.org/cargo/index.html*
[7] Pattern Defeating Quicksort, *https://github.com/orlp/pdqsort*