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

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

Summary

In this project we would be comparing multithreading performance in Rust to that of C++ against various benchmarks. We would be comparing multithreading performance by utilizing two multithreading libraries, Rayon and Crossbeam, which are implemented specifically for Rust to that of OpenMP implementation for C++. The objective of the project is to get an in-depth understanding of these Multithreading abstractions and explain why one would perform better than the other by profiling them.

Background

Over the course of 15-618, we have learned that in order to scale the performance of the software with the hardware, we as developers need to write concurrent 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.

Through the coursework, we have gained a good understanding of OpenMP abstraction for parallelism in C/C++ by allowing compiler directives to be embedded into a serial code. 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).

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.

Rayon’s goal is to take the existing for loops and iterations and make them run in parallel in a way that guarantees that no data-races will be introduced. 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.

Crossbeam is another crate provided by Rust for parallelism. Crossbeam allows for lock-free data structures that do not require a mutex, hence runs much faster. Crossbeam provides multithreaded queues and stacks, which can be used by the threads to produce and consume the data. Since crossbeam provides low-level concurrency primitives such as atomics and concurrent data structures, libraries like Rayon can be built on top of it. Crossbeam also provide scoped threads, which can be used to do a specific task or work.

Challenges

Understanding the multi-threading paradigm of Rust

Rust, as mentioned above, inherently addresses two specific problems, Memory Safety and Safe Concurrency while having zero-abstraction cost. To have a fair comparison and explain its difference from C++, we need to look into the assembly generated and also go through the documentation thoroughly to see the changes under the hood, which are abstracted away in Rust. Rust also provides unsafe parallelism; we should explore that too.

The learning curve (Coding complexity)

Though syntactically similar to C++, there are a lot of differences in the way Rust needs to be coded. The learning curve is steep for us to write code in Rust. There are two concepts, as mentioned above, in Rust that are new to us, borrowing and ownership. To code like we code in C++ doesn't directly work in Rust. We need to worry about function parameters and how they are passed.

Crossbeam vs Rayon

There are multiple Crates (libraries in Rust world) that provide parallelism. Crossbeam and Rayon being the best ones as suggested by the community. They both serve similar purposes but provide a different abstraction cost. We need to understand or even compare which libraries to use for which purpose. Rayon, with its Cilk like behavior, has a really good documentation. Crossbeam on the other hand provides more tools with AtomicCell and lock-free data structures, which can give higher performance. We need to choose right, and profile right. Rayon uses Crossbeam dequeues as a part of its fundamental core.

Profiling Rust

Rust produces an executable from LLVM. Using Oprofiler and valgrind we can get low level information, but to get a function level timing details, we need to experiment with other tools or possibly write our own code using "std::Instant" to profile.

Fairness in comparison

We are comparing two different programming languages which are different at its core. We need to see that we have optimized correctly for the language in both cases. In 15-418, we have learnt to optimize for C++ OpenMP, but we haven't learnt the same for Rust. Hence to ensure that we have optimized correctly, we need to dig into the assembly a bit and see the code. The major section of the code must almost compile to same for the serial code. This way, we can ensure that the changes which are added are only for the parallel code and see the speed up.

Resources

Rust

Rust doesn't have a good documentation, but it does have an active community. To learn the semantics we are using the Rust cookbook. We will also use the Reddit and Rust community for support at anytime.

Rust Resources:

1. Rust CookBook: https://rust-lang-nursery.github.io/rust-cookbook/
2. Rust Lang Book: https://doc.rust-lang.org/book/index.html

Rayon and Crossbeam Resources

We will be starting out to code from scratch to understand some of the coding syntax and nuances of Rust. Later, to speed up our coding, we will use a modified version of some of the codes present as demos in the Rayon github project and the resources mentioned below. There will be planning and understanding as to how to modify this code and a decision will be taken each time about when to use Rayon or Crossbeam.

Resources:

1. Rayon: https://github.com/rayon-rs/rayon
2. Crossbeam: https://github.com/crossbeam-rs/crossbeam

Profiling

Profiling is another important part of this project that requires attention and specific tools. Researching over Rust community feedback, we got to know about two specific profilers apart from the using operf.

Rust Profiling Resources:

1. Flame Profiler: https://github.com/TyOverby/flame
2. cpuProfiler: https://github.com/svenstaro/cargo-profiler

For C++, we will be using the cpu_timer library and gprof which we have used in the assignments.

Papers and Blogs

There are multiple blogs and papers that refer to the performance of Rust and C++. Few of them mention.

Blogs we are planning to refer:

1. Rayon Usage: http://smallcultfollowing.com/babysteps/blog/2015/12/18/rayon-data-parallelism-in-rust/
2. Few speed tests: https://benchmarksgame-team.pages.debian.net/benchmarksgame/faster/rust-gpp.html
3. Rust as Fast as C++: https://jackmott.github.io/programming/2016/08/30/think-before-you-parallelize.html

Having done an assignment using OpenMP, we are planning to write our own code and profile it as mentioned above.

Basic Project Requirements

Requirements for Rust:

- Compiler: rustc (installed with Rust)
- Package-manager: Cargo (preinstalled with Rust)
- Crates: Rayon, Crossbeam

Requirements for C++:

- Compiler: g++ compiler.
- Packages: OpenMP

System Specifications:

- Name: GHC Machine
- Core: Intel(R) Xeon(R) CPU E5-1660 v4 @ 3.20GHz
- Physical Cores: 8
- Hyperthreading: Yes (16 logical cores)

Goals and Deliverables

Plan to achieve:

- Explain multithreading boilerplate of Rust, in comparison to C++.
- Strongly assert the difference in OpenMP vs Rayon through measurements.
- Benchmarking tests on each of the benchmarks chosen, for different sizes and number of threads using Rayon or Crossbeam.
- Collate benchmark performance to explain the speed difference between C++ and Rust.

Stretch Goals:

- Extend the performance comparison using more benchmarks
- Compare ISPC usage in C++ and Rust. Both have ISPC support.
- Rust provides "faster" crate as competition to OpenMP's #pragma omp SIMD optimization, which will be compared.
- Create an evaluation of the ease of parallel programming for a beginner.

Benchmarks

1. Mandelbrot: A problem with inherent load imbalance.
2. Matrix multiplication: A high arithmetic intensity problem.
3. Quick sort: Another Sort that requires recursive partitioning.
4. Fibonacci: A task level problem that asks for OpenMP task functionality to be compared.
5. Radix sort: A low arithmetic intensity involving multiple barriers.

Platform Choice

For the language part, our experiments need all of our code to be written in Rust and C++. The written code specifically will use Rayon/Crossbeam crates and OpenMP libraries respectively.

For the system part, we choose to run all our multithreaded code on a single GHC Machine. The requirement for our experiments is that we need to have a multithread support and wide enough sweep of available cores. GHC machine provides us with a shared memory model with support for 16 threads (16 logical cores and 8 physical cores) would suffice for a (2,4,8,16) sweep for number of threads.

Schedule

Week Number	Start Date	End Date	Work
1	11-Apr-19	16-Apr-19	: Setup : Sequential code check : Mandelbrot
2	17-Apr-19	23-Apr-19	: Checkpoint 1 Report : Matrix Multiplication : Quicksort
3	24-Apr-19	30-Apr-19	: Checkpoint 2 Report : Fibonacci : Radix sort : Stretch Goals (Additional Benchmark)
4	1-May-19	7-May-19	: Rest of Stretch Goals : Charts and Visualization : Analysis : Final Report

<u>Week 1</u>

- Setup Rust on the GHC machine. Test Rayon, Crossbeam, and other depending crates.
- Write Mandelbrot program in Rust and C++ with basic speed benchmarking for serial code and check the assembly to have fair comparison.
- Understand the working of Rayon and usage of Crossbeam to compare it with the ideology of OpenMP in theory.
- Parallelize Mandelbrot and study the speed-ups and generate profiling report.

<u>Week 2</u>

- Create Checkpoint 1 Report.
- Write and test code for Matrix multiplication and Quicksort benchmarks for thread parallel code for Rust and C++.
- Benchmark comparison for Rayon/Crossbeam vs OpenMP.

<u>Week 3</u>

- Create Checkpoint 2 Report.
- Write and test code for Fibonacci and Radix sort benchmarks for thread parallel code for Rust and C++.
- Benchmark comparison for Rayon/Crossbeam vs OpenMP.
- Attempt for the Stretch goals (Additional Benchmarks).

Week 4

- Attempt other Stretch Goals.
- Collate the benchmark performance for a good visualization.
- Analyze data and consolidate reached goals with explanations.
- Create Final Report.