Comparing Parallel Languages:

OpenMP in C++ vs Rayon in Rust

By: Manas Vegi, Jayanth Reddy,
Stacy Chao

Problem Statement

Rust is a relatively new language but provides guarantees on better memory management and safety with minimal compromise in performance, unlike C++ which is not does not guarantee safety in order to provide low-level control.

This work will evaluate Rust's against a decades-old well-developed tool, OpenMP, and evaluate which language is better for specific scenarios.

Literature Review

A. Bychkov and V. Nikolskiy. Rust Language for Supercomputing Applications

- Tested OMP for g++ and clang.
- Compared their implementations of algos with library methods
- Machine used did not have a lot of logical cores to test scalability (12 cores)
- Embarrassingly parallel programs (no shared memory use)

S. Nanz, S. West, K. Silveira, and B. Meyer. Benchmarking Usability and Performance of Multicore Languages

- Compared parallel languages with different design philosophies (Chapel, Go, Cilk, TBB in C++).
- Expert review and enhancement
- Benchmarking programs chosen with usability and quick programming time in mind
- Reasons for choosing each benchmark is not apparent

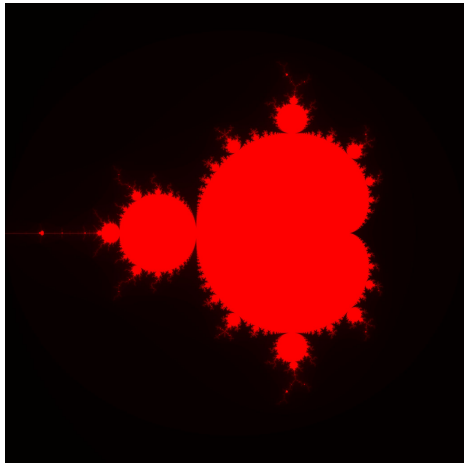
G. Kalderen and A. From. A comparative analysis between parallel models in C/C++ and C#/Java

- Compared C with OpenMP, C++ with TBB, C# with TPL, and Java with fork/join
- Compared the performance on different problem granularities
- Runs comparisons over a single benchmark of SparseLU decomposition, which does not all aspects of the language

Experimental Setup

Tested on Crunchy1 with 6 benchmarks each aimed at evaluating different granular aspects of the languages (Rust and OMP):

1. Sorting - MergeSort | QuickSort (100 M)
2. Matrix Multiplication (8192x8192)
3. Mandelbrot (4096x4096)
4. Remove Duplicates (1 M and 10 M)
5. WordCount (1 M and 10 M)
6. Creation/Destruction of Threads

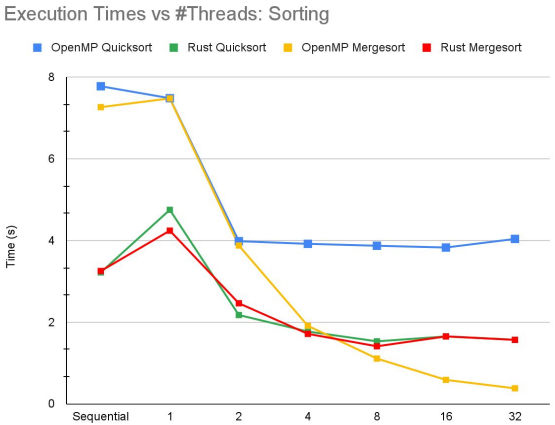


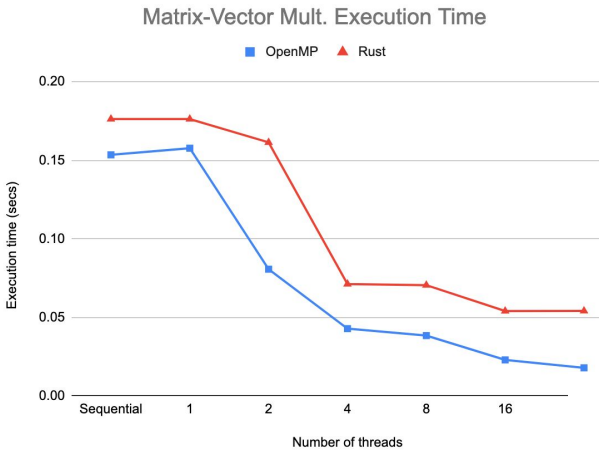
#of Threads for Testing Configurations: 1 - 32

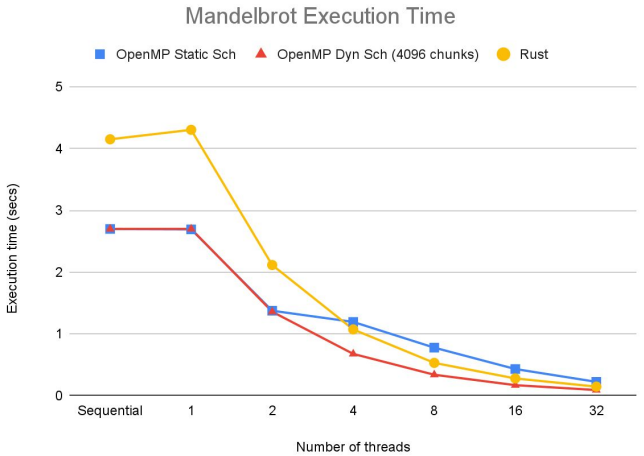
Compiler Versions: g++ version - 12.2.0 rust version - 1.68.1

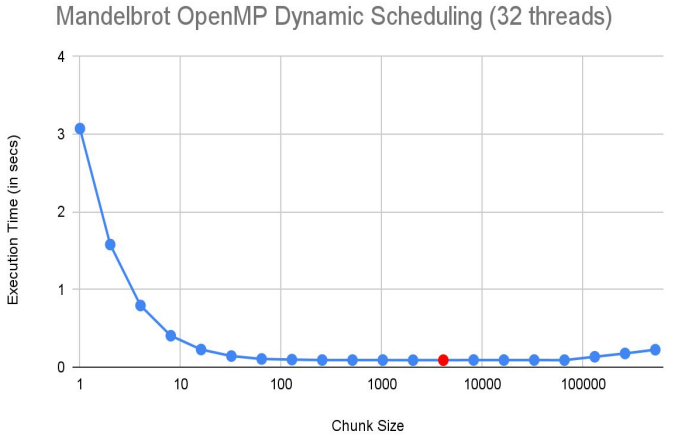
Implementation: Ensured to our best abilities, the algorithms followed very similar implementations.

Results and Analysis

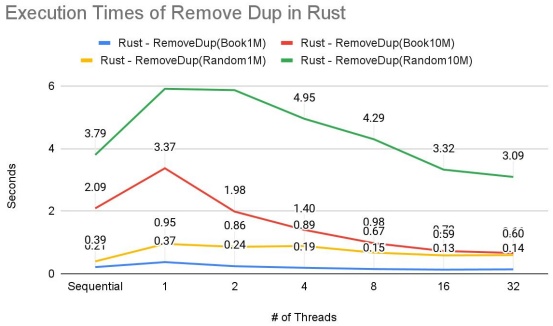


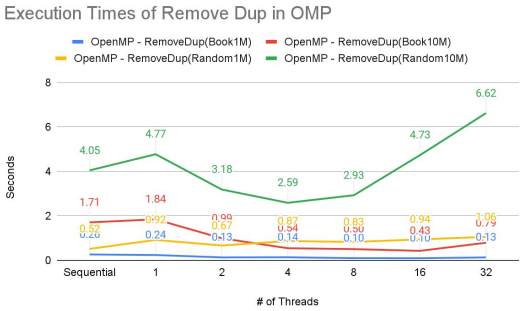


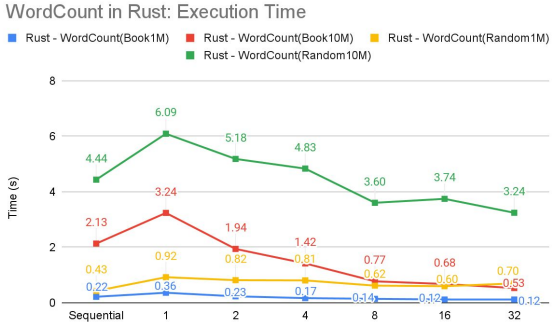


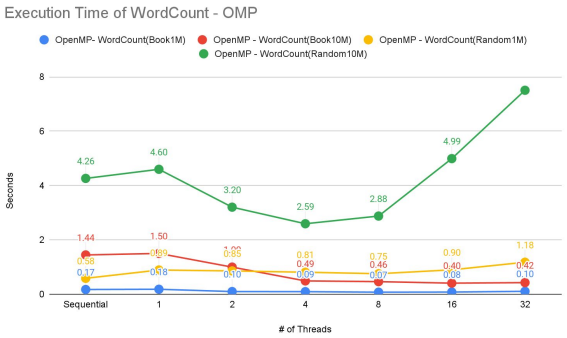


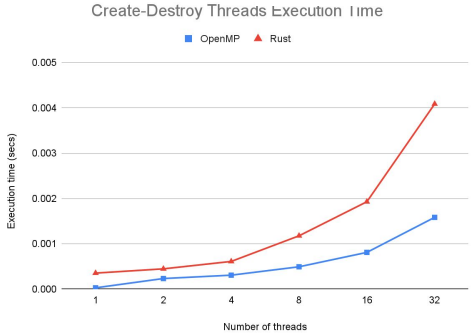
Rust › OpenMP for sequential **sorting**, and OpenMP MergeSort performs best with ›8 threads.
For **Matrix multiplication**, OpenMP is more performant at all thread count levels.
    Equal computation per task: overhead of dynamic scheduling and data ownership model in Rust.
Mandelbrot: Rust outperformed OpenMP (with static scheduler)
    Mandelbrot tasks have an unbalanced load
    Work-stealing based dynamic scheduling used by Rust performs better
    OpenMP (dynamic) however performed best overall, but requires careful tuning.











- RD and WC perform much better for a document with more duplicates as compared to one with fewer: <u>hashset reduction overhead</u>
  - Isolated the problem to the reduction step by timing only the reduction step
- Speedup is better for the higher duplicate document as we increase #threads in Rust
- At large number of threads, Rust performs better at hashset based reductions because it has ondemand reduction as opposed to the implicit barrier of the reduction step in OpenMP. This makes a difference when there are large imbalanced loads across the threads.
- OpenMP is faster than Rust at thread creation/destruction -› could be attributed to the difference in design and implementation and that OpenMP is more efficient.

Conclusions

- For purely computational tasks with equal work, OpenMP static scheduler showed better performance. For tasks with unbalanced workloads, Rust gives better performance due to work stealing . OpenMP however can perform better with the right tuning, but Rust seems more robust to unknown and unbalanced workloads.
- Rust is more efficient in our hashing benchmarks with reduction. This could be due to the differences in implementation of the reduce step across the languages: the continuous reduction on-demand in Rust may avoid the synchronization overhead when it comes to imbalanced hashset reductions.
- Rust is easier to program due to its strong memory management and ownership model. It also tries to catch most memory errors at compile time, which makes the code more robust for critical applications. Rust also provides higher-level abstractions for parallelism like iterators and closures. The additional features however come with overhead and need to be evaluated for the particular use case.