Comparing Parallel Programming Languages: OpenMP in C++ vs Rust

N Jayanth Kumar Reddy (nr2686)
Stacy Chao (pyc298)
Manas Vegi (mv2478)

Abstract

In this project, we compare the performance and usability of OpenMP (C++) and Rayon (Rust) for parallel applications. Rust is a relatively new language but provides guarantees on better memory management and safety with minimal compromise in performance. This work will evaluate these features against a decades-old well-developed tool, OpenMP, and evaluate which language is better in what scenarios. We do this by testing on six benchmarks each aimed at evaluating different granular aspects of the languages. The results indicate that OpenMP performs better at programs that can be split into similar-sized compute-intensive sub-tasks, whereas Rust is a better choice for imbalanced loads or when the reduction step involves merging hashsets/hashmaps. Rust also has better programmability with its rich compile-time checks, ownership model, and library methods for parallelism.

Introduction

Rust[5] is a new programming language, launched in 2015 by Mozilla Research. It is a compiled language with great promise in parallel applications due to its memory safety features, which protect against bad references, memory leaks, race conditions, etc. Thus, it is a worthy endeavor to see if these safety features come at a performance cost.

Rust has its implementation of multithreading features in its *rayon* crate, a high-level library that provides a simple and intuitive interface for parallelizing collections. Unlike OpenMP which works with explicit management of threads and synchronization through pragma directives, rayon provides library methods to implement concurrency in code. Both of them are able to achieve great parallel functionality with minimal additional code.

Work Stealing in Rust's rayon crate[6] implements a work stealing algorithm. This is an automated way to ensure efficient parallelism by ensuring close to even distribution of work amongst all logical cores by allowing the stealing from another thread's tasks deque when free. There is a similar method of implementing these effects in OpenMP, however, the user must manually change the default of static scheduling to dynamic scheduling and customize the chunk size. Dynamic scheduling of OpenMP divides the work into smaller tasks, and each thread takes on a new task as soon as it becomes available.

In the rayon crate, the reduce operator works in conjunction with the work stealing algorithm provided in rayon to allow the start of reduction before all the threads complete their work, whether that be a fold or map operation. In OpenMP, the reduction clause that can be declared in the pragma directive, by default, forms an implicit barrier to the reduction operation that merges all the values only at the end of the parallel block. The nature of this barrier can be altered based on the user's own implementation. For instance, with critical regions (at a performance hit).

Literature Survey

The paper[1] by A. Bychkov and V. Nikolskiy compares Rust with C++ to understand if Rust is a capable programming language for supercomputing applications. The paper mainly focuses on mathematical benchmarks in its comparisons: matrix multiplication and numerical integration using the trapezoidal rule. They then compare C++'s MPI with the rust-mpi implementation and discuss the memory safety of Rust along with performance.

Pros:

- Compared results across both clang and g++ versions of the C++ compiler to test for C++ optimizations. They also compared a naive implementation in Rust (for matmult) with the one using high-level functions using Rust's parallel iterators. Moreover, for matrix multiplication they also compared popular parallel libraries in the languages. This provides a real-world comparison because most programmers would use pre-built libraries due to them being very optimized. This gives a good idea about the comparisons of the Rust and g++ libraries and the built in parallel implementations.

Cons:

- For multicore processing, the tests were performed on i7-8750H cpu which only has 6 CPUs and a total of 12 logical threads (with hyperthreading). This means they could not test scalability very well in terms of number of threads. Moreover, due to this being a laptop chip, it is hard to know how many other processes were running at the time of their test and if that interfered with their results in any way.
- Measured performance for Rust vs C++ in GFlops which may not be a true indicator for performance and just compiler specific details.
- Only did benchmarks on embarrassingly parallel mathematical calculations. Thus, we did more shared memory benchmarking with different contexts.

In another work by S. Nanz et al [2], they try to compare four languages(Chapel, Cilk, Go, and Threading Building Blocks) over six benchmark programs which are picked from the Cowichan Problems[7]. They compare languages in aspects of Usability and Performance. For Usability, they use metrics like source code size(Lines of code) and coding time. For performance, they use Execution time and speedup metrics.

Pros:

- They compared parallel languages with different approaches to parallel programming like message passing vs shared memory and Object-oriented (Chapel/Cilk) vs imperative(Go/Cilk) vs a C++ library(TBB) and also different programming abstraction levels like Partitioned Global Address Space(Chapel), Structured Fork-Join(Cilk), etc.
- They were able to define quantitative metrics (based on source code size and coding time) to measure usability of languages.
- They also got their experiments reviewed and enhanced by notable experts, which adds credibility to the results in terms of how an experienced professional would use the language.

Cons

- The benchmark programs they picked seem to be chosen with usability and quick programming time in perspective which might induce an inherent bias and not evaluate the performance of languages to a good extent.
- They do not provide a granular understanding of which aspects of usability/performance each benchmark tries to quantify. Our work is able to provide this understanding.

[3] compares four languages C with OpenMP, C++ with TBB, C# with TPL, and Java with fork/join. The results indicate that programming in C with OpenMP gave the lowest execution time, and C++ with TBB gave the best speedup relative to sequential execution. C# with TPL gave the worst performance in all tests. The paper tests the performance at different levels of granularity in terms of splitting the input, results showed the best performance at a particular range of granularity and slower performance at lower and higher sizes. This paper runs comparisons over a single benchmark of SparseLU decompositions, which might not be testing all aspects of the language and definitely not at a granular level.

Proposed Idea

We compare Rust and OpenMP by running them against diverse benchmarks. We picked the benchmarks ensuring that they test the languages at a granular level.
The six benchmarks we used are as follows:

1) **Sorting**: We perform tests to compare stable(Merge sort) and unstable(Quick sort) sorting algorithms in the two languages. We test these with the existing sorting functionalities provided in the two languages, GNU Sort(__gnu_parallel::sort with appropriate tags) for C++ and Rayon par_sort/par_sort_unstable. The reason to use existing implementations instead of writing from scratch was to get a practical understanding of the performance and usability, since most professional programmers would use the library functions. Also, it is a way to compare each respective language's parallel implementations of the algorithm head-to-head.

   Reasons for use: Sorting is commonly used in many applications, and its inherent divide-and-conquer nature makes it a good benchmark to test parallel performance. We could potentially draw conclusions over whether one language performs better scheduling suited for such tasks.
2) **Matrix-Vector Multiplication:** This is a commonly used operation in many Machine Learning and Graphics applications. This was parallelized by performing the dot product of each row of the matrix with the vector on a different thread.

   Reasons for use: This is a compute-intensive test with equal load distributed to each task. Since there is no synchronization/communication involved and each thread will take the same unit of work, this test will isolate the raw performance and scheduling overheads of the two languages.
3) **Mandelbrot:** In this task, we generate visualizations for the famous Mandelbrot set, which is a set of complex numbers for which the iterative computation $z_{n+1} = z_n^2 + c$ does not diverge to infinity. Here the iteration starts with $z_0 = 0$, and c is the complex number for which we are checking the property. This is an embarrassingly parallel problem and was parallelized by computing each pixel value in a different thread task with no communication involved. After a set number of max iterations (100 in our case), if points have not started diverging we assume they converge.

   Reasons for use: This is a good test to simulate imbalanced workloads. For each pixel the decision whether it is diverging would be made after a different number of iterations, making the load different for each task. This also allowed us to experiment with the different scheduling configurations in the two languages and compare them.
4) **Word Count:** Wordcount is a very common task done in many Natural Language Processing applications. It outputs the frequency of words in a document. This task involves shared memory processing during the reduction because the different maps must be reduced into one. We utilize the in-house implementation of Hashmaps for both the languages.

   Reasons for use: This task is a great example of shared memory processing. There is a lot of communication between the threads and involves locking on the key during the reduce step.

5) **Create/Destroy threads:** In this experiment, we measure the time overhead for creating and destroying threads. This is a simple experiment where we create a fixed number of threads and perform a small operation like adding two numbers (this will ensure the compiler optimization does not ignore thread creation).

   Reasons for use: It is well known that creating and destroying threads frequently can slow down the program due to the system calls involved. This program allows us to measure and understand the magnitude of overhead, thereby indicating how careful a programmer needs to be when creating/destroying threads in each language.

6) **Remove Duplicates:** This experiment takes in a very large string with words separated by spaces and outputs a single string that contains the unique occurrences of words in our input. For this, we utilize the hashset implementations provided by C++ and Rust. In Rust, we have implemented a reduction in the reduce function of the parallel iterator. To keep the algorithm very similar, we declared our own reduction for hashsets (with the same logic) in OpenMP instead of using critical regions.

   Reasons for use: This is a task that uses hashing data structures for its implementation. Moreover, there is a bottleneck of reducing the hash sets together in the parallel algorithm. Thus, it allows us to test the reduction operations in OpenMp and Rust.

In each of the tasks apart from monitoring the performance we also try to draw heuristics over the ease of programming and debugging.

Experimental Setup

G++ version: 12.2.0

Rust version: 1.68.1

Datasets link used for Wordcount and Remove Duplicate experiments

All experiments were run on the Crunchy1 machine at CIMS, NYU. Below are the details of the hardware

- Hostname: crunchy1.cims.nyu.edu
- OS: CentOS Linux 7 (Core)
- Kernel Version: 3.10.0-1160.81.1.el7.x86_64
- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 64
- On-line CPU(s) list: 0-63
- Thread(s) per core: 2
- Core(s) per socket: 8
- Socket(s): 4
- NUMA node(s): 8
- Vendor ID: AuthenticAMD
- CPU family: 21
- Model: 1
- Model name: AMD Opteron(TM) Processor 6272
- Stepping: 2
- CPU MHz: 2100.171
- BogoMIPS: 4200.34
- Virtualization: AMD-V
- L1d cache: 16K
- L1i cache: 64K
- L2 cache: 2048K
- L3 cache: 6144K
- NUMA node0 CPU(s): 0-7
- NUMA node1 CPU(s): 8-15
- NUMA node2 CPU(s): 32-39
- NUMA node3 CPU(s): 40-47
- NUMA node4 CPU(s): 48-55
- NUMA node5 CPU(s): 56-63
- NUMA node6 CPU(s): 16-23
- NUMA node7 CPU(s): 24-31
- Flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt pdpe1gb rdtscp lm constant_tsc

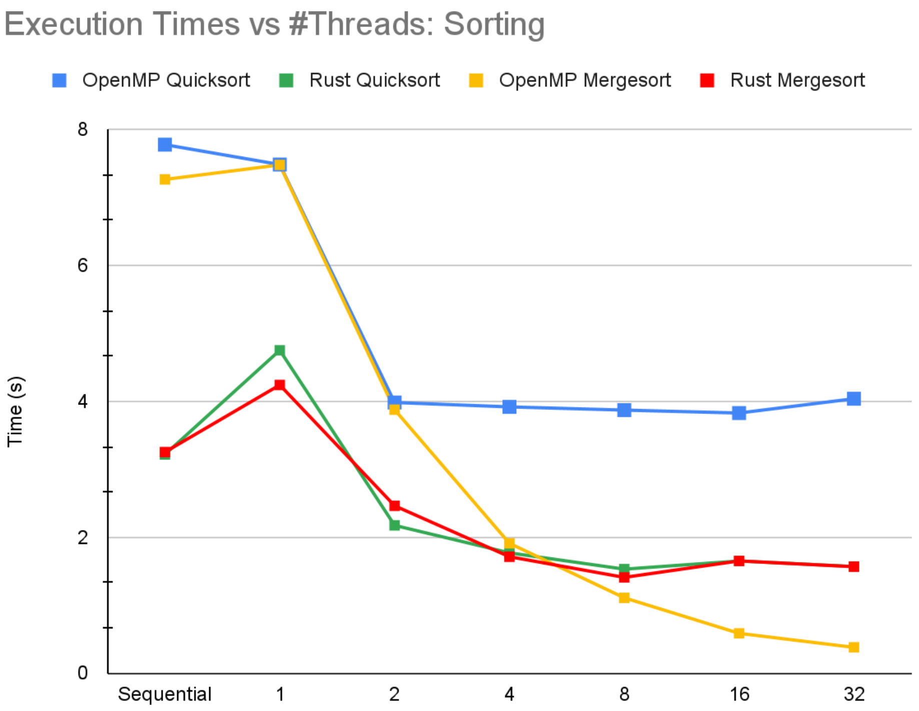
art rep_good nopl nonstop_tsc extd_apicid amd_dcm aperfmperf pni pclmulqdq monitor ssse3 cx16 sse4_1 sse4_2 popcnt aes xsave avx lahf_lm cmp_legacy svm extapic cr8_legacy abm sse4a misalignsse 3dnowprefetch osvw ibs xop skinit wdt fma4 nodeid_msr topoext perfctr_core perfctr_nb cpb hw_pstate ssbd rsb_ctxsw ibpb vmmcall retpoline_amd arat npt lbrv svm_lock nrip_save tsc_scale vmcb_clean flushbyasid decodeassists pausefilter pfthreshold

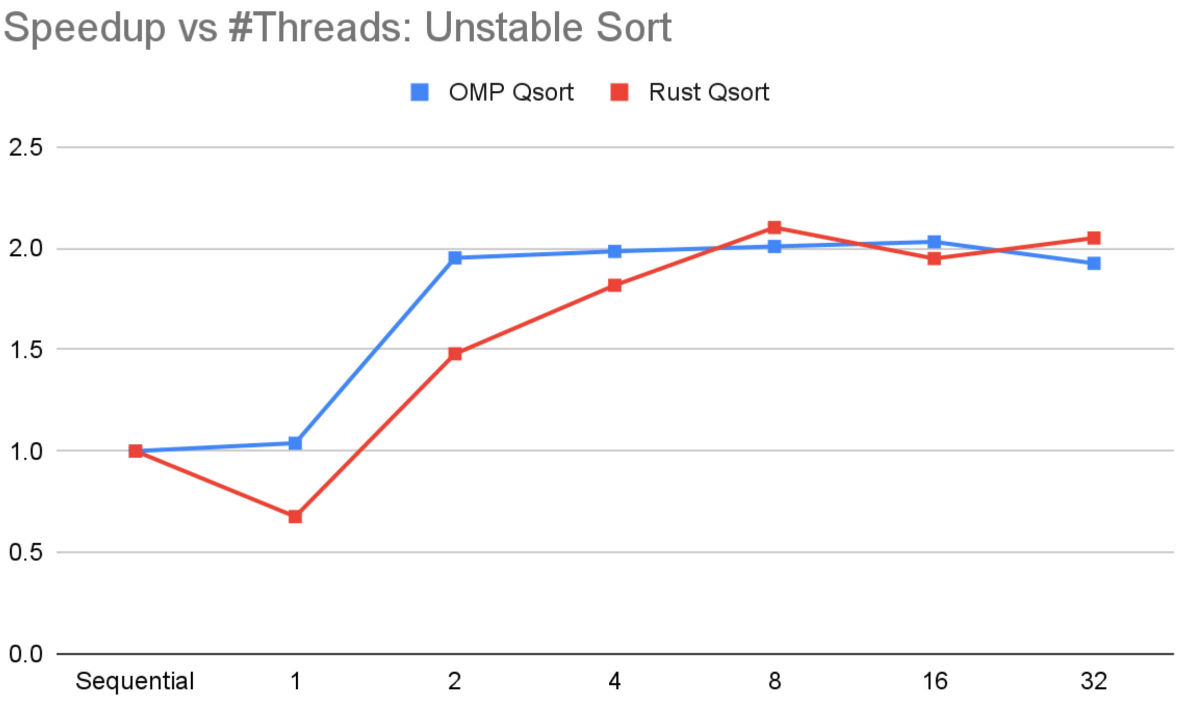
Experiments & Analysis

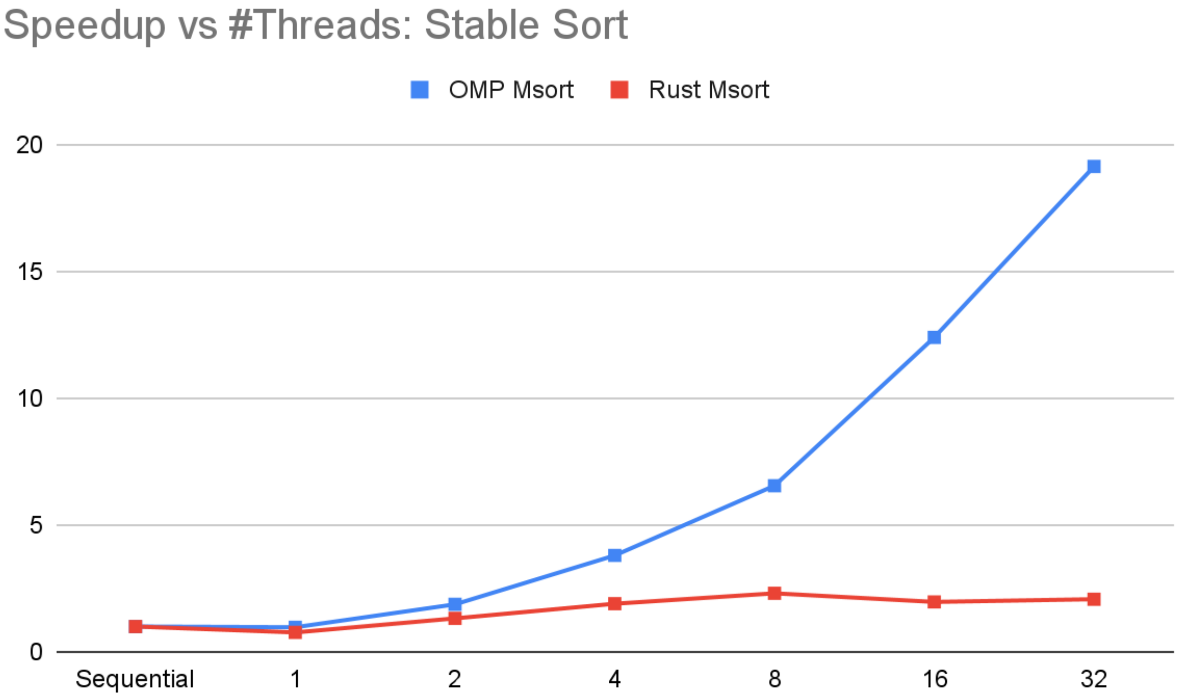
In this section we will go over the results obtained for each benchmark and draw conclusions over them.

Sorting

We ran sorting on a vector of size 100,000,000 integers between -100 to 100. These integers are randomly generated in both the languages to avoid any cache optimizations interfering with the parallelization results. The timer excludes the generation of the random vectors itself.







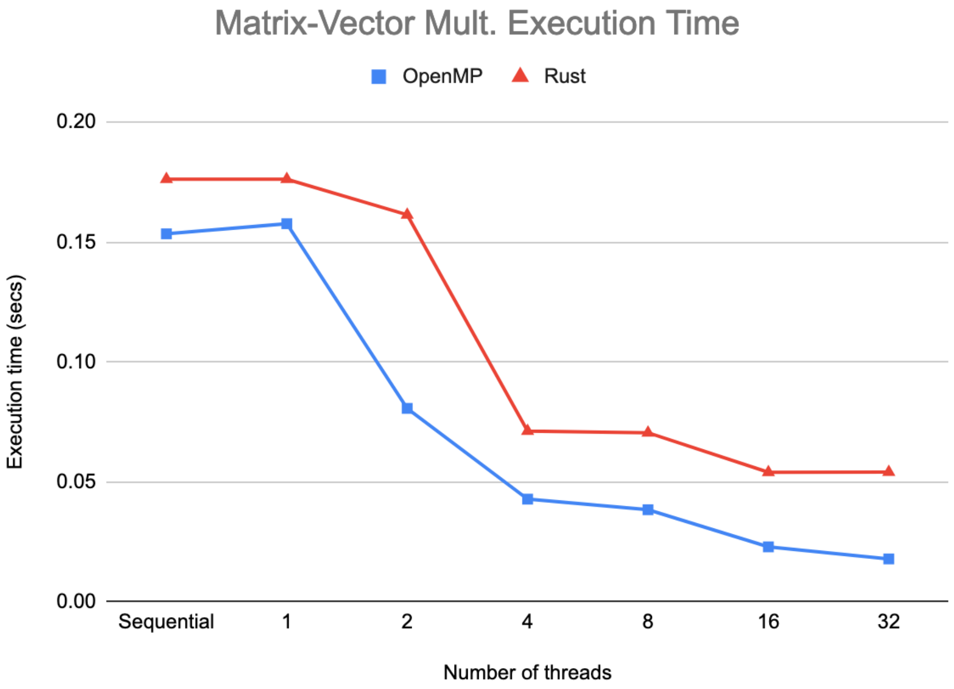
Comparing quicksort algorithms vs mergesort algorithms in OpenMP for C++, we notice a stark difference in the relationship between number of threads and speedup. Where the quicksort algorithm plateaus its speedup at 2 threads, the mergesort implementation shows a consistent speedup as we increase the number of threads all the way up to 32 threads. This is most likely due to the fact that quicksort (unstable) is an in-place algorithm. This means that the threads split work on the vector but access the same vector to do their sorting. As a result, there would be a lot more race conditions implying more time spent in locks (critical regions). Meanwhile, mergesort (stable) algorithm is not in place and each thread is able to create its own private sub-vector to sort. Meaning that the race conditions are very few (only during the reduction step).

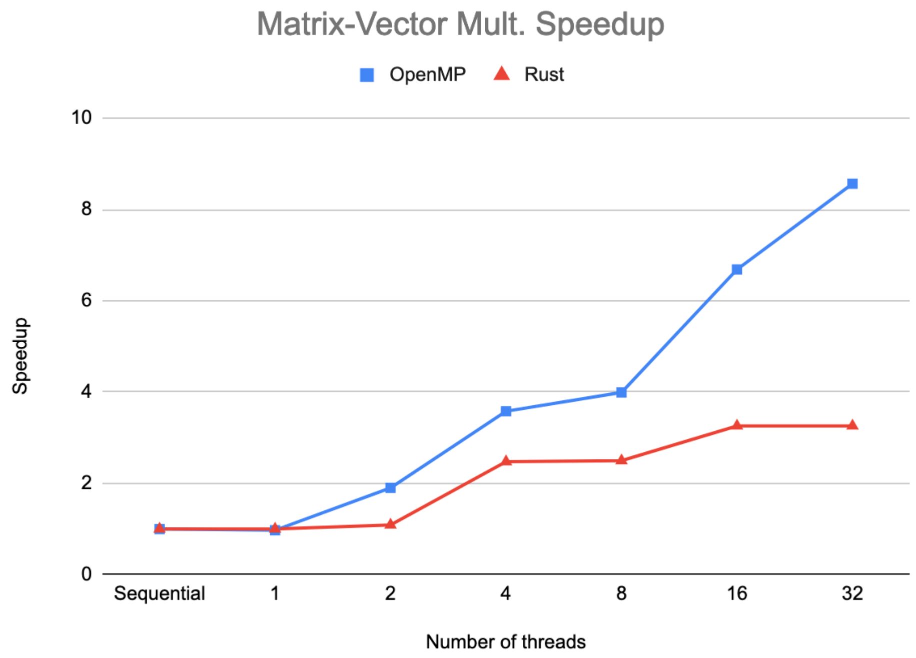
The results for Rust's stable vs unstable sort are very similar. Both of them go upto a speedup of 2 for 4 threads and then there is no more increase in speedup. The execution time is very similar too. This was contrary to our expectations: a likely cause could be the very quick sequential time and optimizations in the sort algorithm when compiling the code in release mode with the tag '-r'.

It is also interesting to note that Rust beats OpenMP's implementation at lower thread counts in both algorithms.

Matrix-Vector Multiplication

All experiments used a matrix size of 8192x8192 multiplied with a vector of 8192 elements.

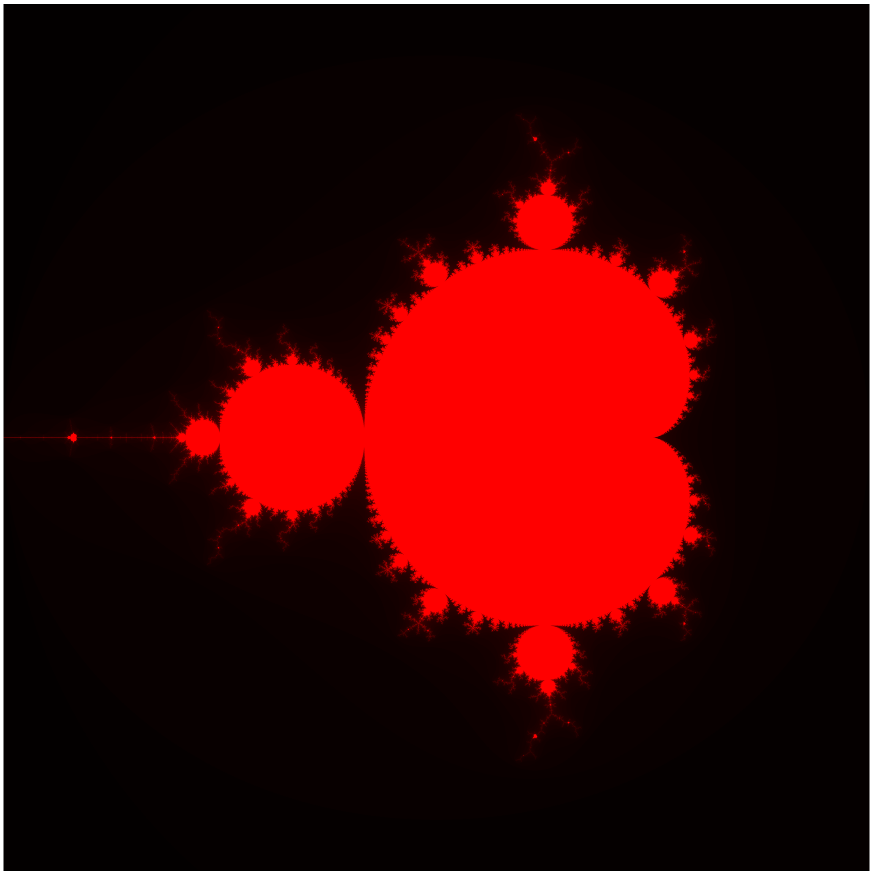




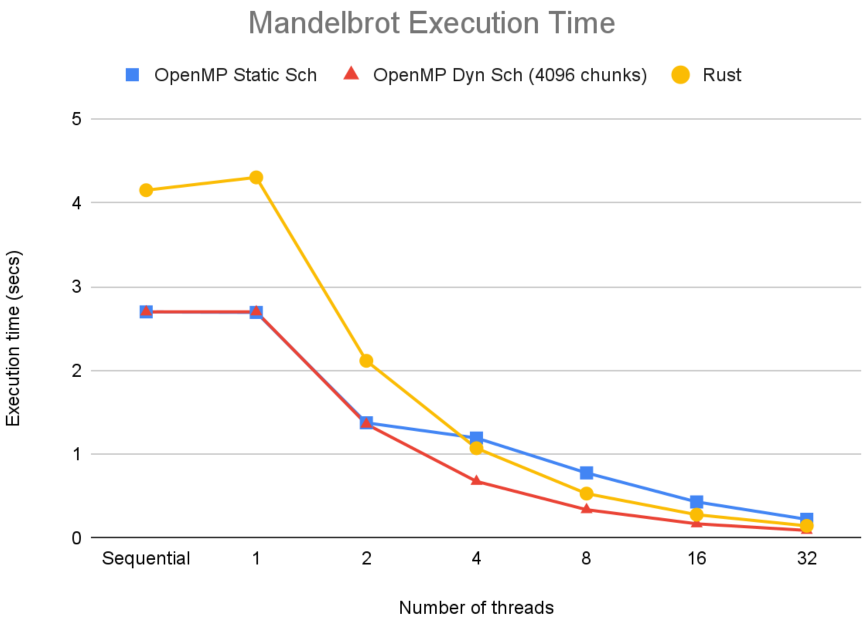
We observe that although OpenMP and Rust are very close, OpenMP is clearly more performant at all thread count levels. The speedups observed in OpenMP are also better. This could be attributed to the fact that each task has equal computation and the overhead of dynamic scheduling and data ownership model in Rust is slowing it down. Therefore, in scenarios where we can divide the task into equal size work portions, using OpenMP with its default static scheduler is a much better option. The drop in efficiency (Speedup/Num. of Threads) can also be attributed to the additional overhead in Rust for dynamic scheduling when there are more threads. Both OpenMP and Rust took similar LOC around 90-100. In terms of programmability, rust is easier since the strong memory management ensures there are no segmentation faults due to pointer issues. In OpenMP, the programmer needs to have a good understanding of pointers since it also involves a pointer of pointers for 2-dimensional arrays. Therefore, Rust beats OpenMP in programmability.

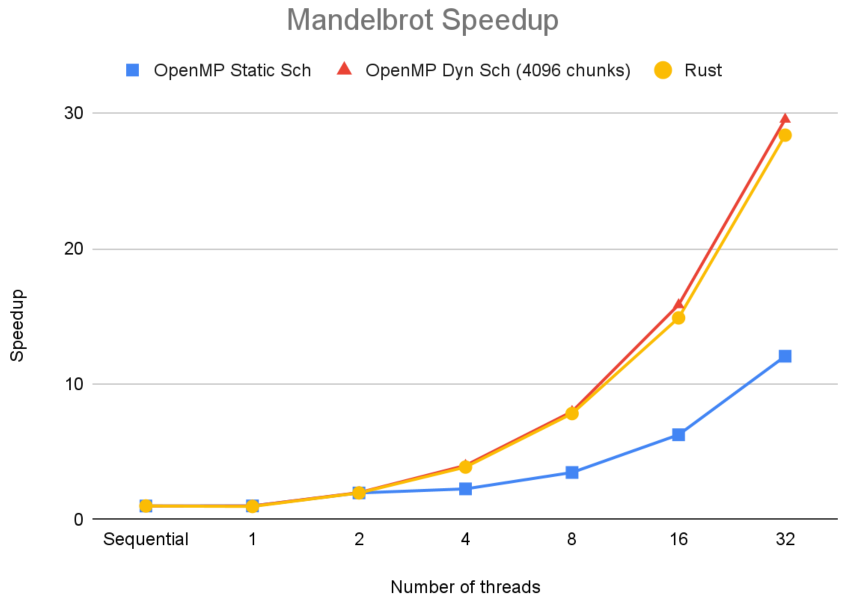
Mandelbrot

Image Size was set to 4096x4096. Max Iterations was limited to 100.

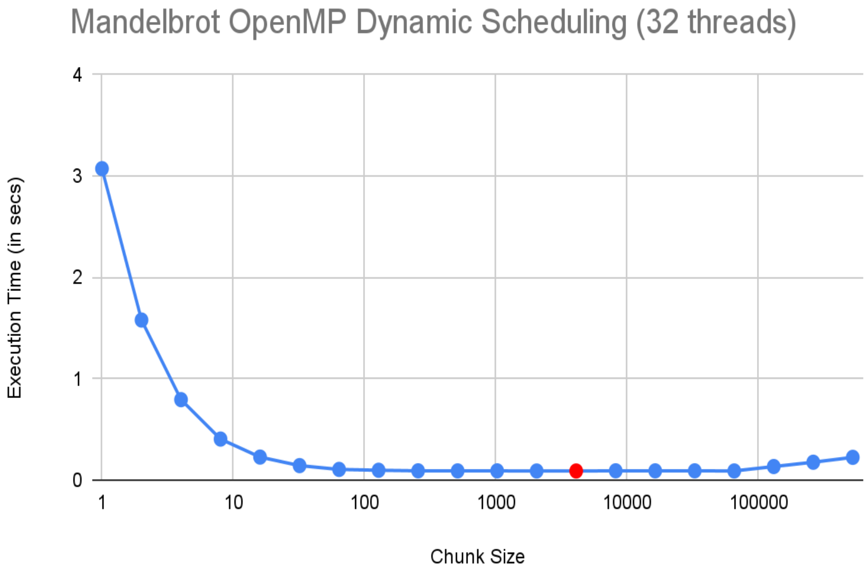


Visualization generated of the mandelbrot set





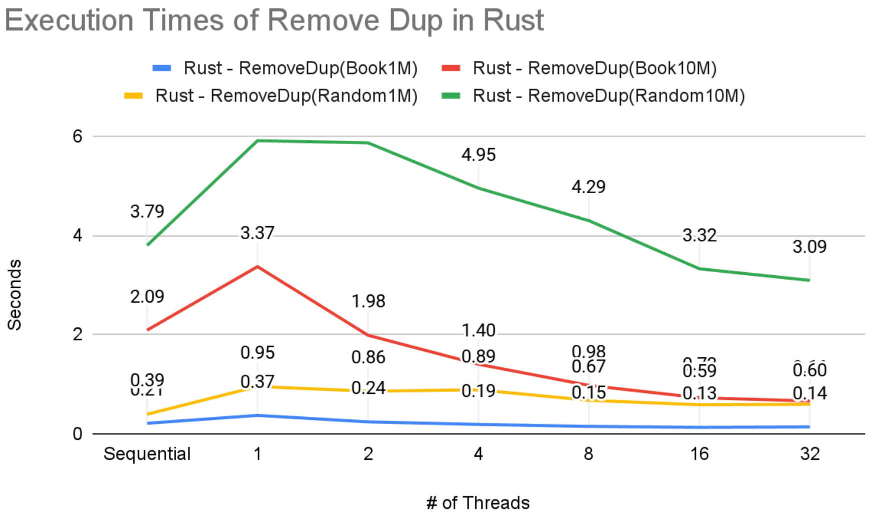
For sequential code, OpenMP outperforms Rust, which is inline with our expectations since Rust has additional memory management overhead. We initially compared OpenMP using the default(static) scheduler with Rust, and as expected rust performed better. This is because Mandelbrot tasks have an unbalanced load and the work-stealing based dynamic scheduling used by Rust performs better in this setting. To further confirm our understanding we experimented with using dynamic scheduling in OpenMP. We experimented with multiple chunk sizes as shown in the below figure, and found 4096 chunk size to have the best performance. OpenMP with 4096 chunk size gave a good speedup of 29.5 at 32 threads, even higher than Rust.

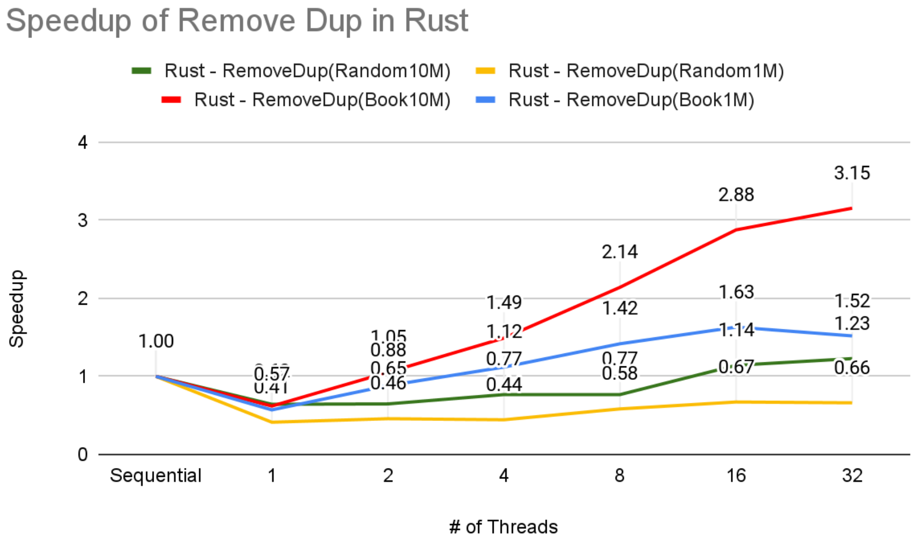


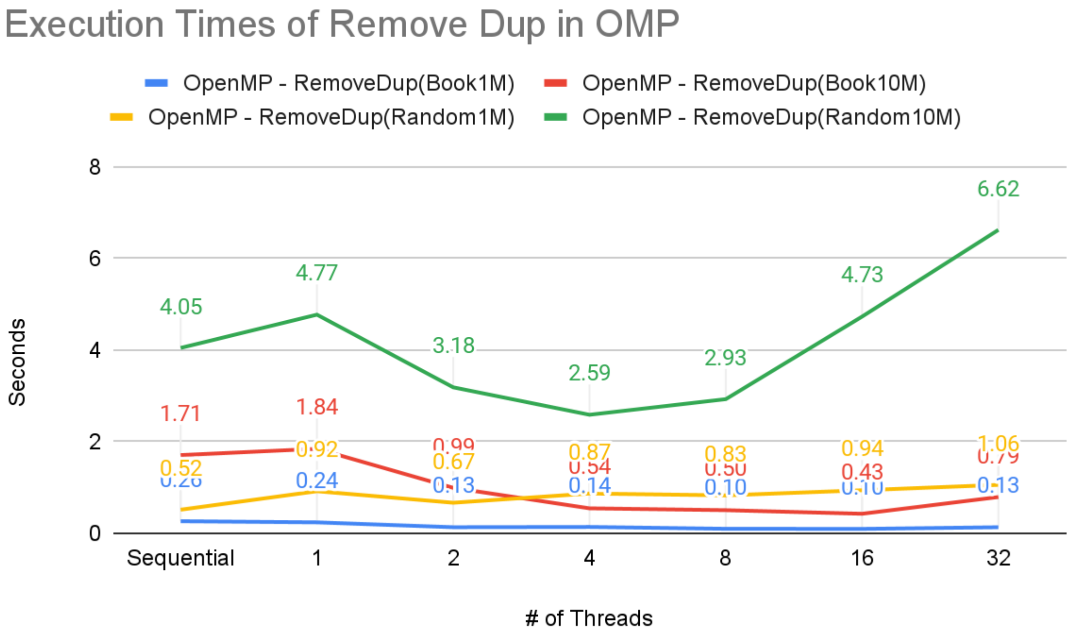
Overall, with carefully tuned scheduling OpenMP outperforms Rust. But this involves some effort, and Rust is able to perform well with no tuning involved. We also observed that it is much faster to code the program in Rust as it had better library support for tasks like saving an image to png (For saving image to PNG in C++ we used code from here, written by Prof. Daniele Panozzo). In terms of Lines of Code(LOC) C++ OpenMP took 142 lines vs 120 for Rust which is very similar.

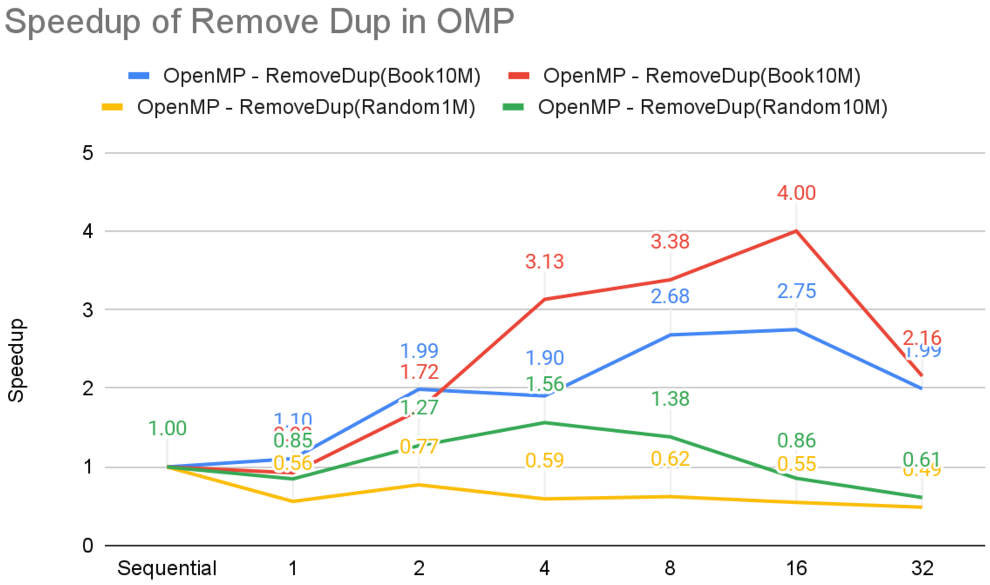
In conclusion for Imbalanced workloads, it is better to use Rust, but if the task is very specific then OpenMP is better with the right tuning of the scheduler.

Remove Duplicates


Execution Times of Remove Dup in Rust


Speedup of Remove Dup in Rust





We ran two different types of text file inputs: one with random words (which has a minimal duplicate rate) and another from books (more duplicate words). The remove-duplicates algorithm uses a reduction approach instead of critical regions: we implemented the same reduction logic for both OpenMP and Rust.

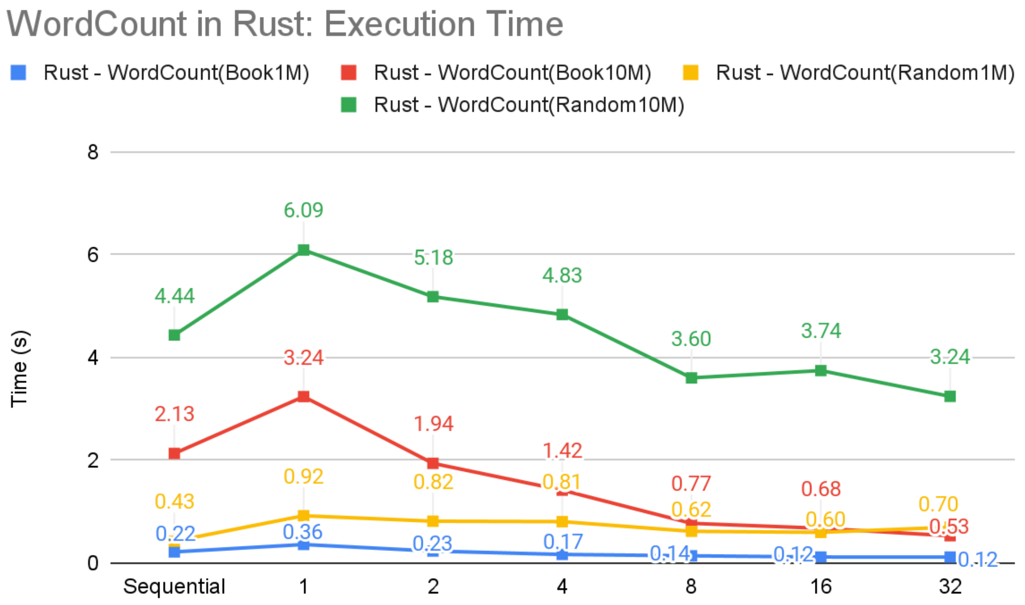
When running test cases with input texts that have a higher proportion of duplicate words–resulting in a lower ratio of hash set keys to number of words–the execution time drops as we increase #threads to 8 but starts increasing from 16 threads onwards.

One possible reason behind the turning point in the graph is the synchronization overhead occurring in the reduction operation. Each thread works on its private hash set, and the reduction merges all the local hash sets (private to each thread) into one hashset. Since this final ‘hash set’ is a shared resource, with the increase of threads trying to access the shared memory simultaneously, there is a higher likelihood of a thread trying to access a locked resource. Thus, the race conditions drastically increase the time of the ‘reduction step’.

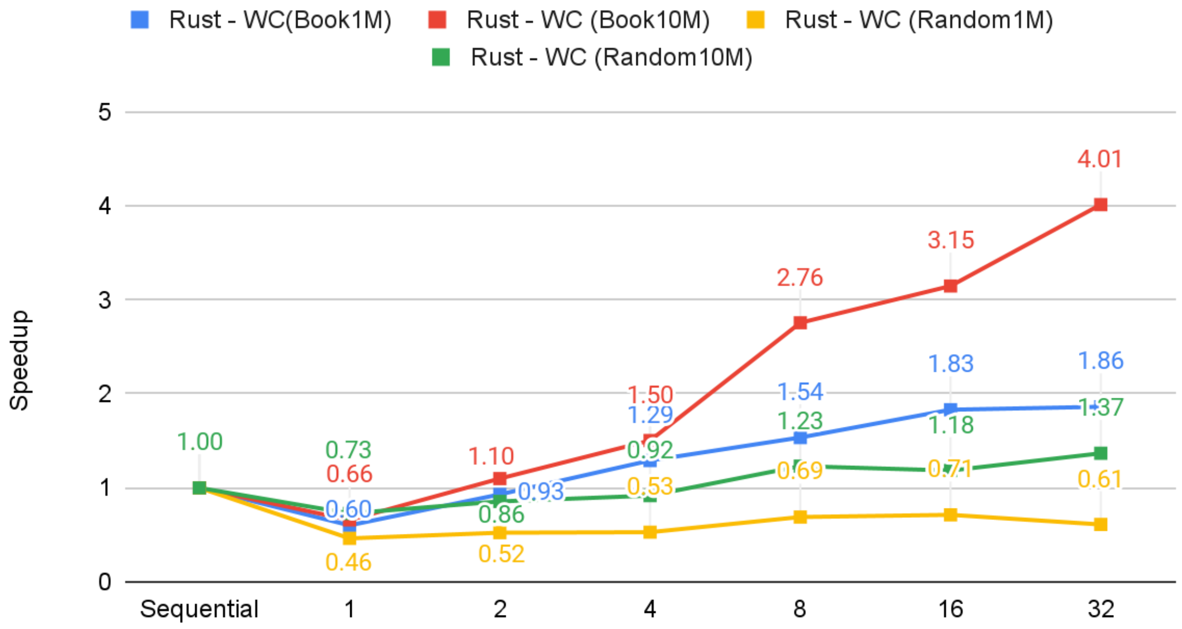
This idea is validated when we run the algorithms on an input with less repetitions. This time, the increase in execution time begins at 8 threads rather than the 16 threads of the previous program. This is most probably due to the increase in memory required to store the local hash sets and the resulting final ‘hash set’ of the reduction operation. With a proportionally higher number of unique words, there are a lot of inserts into the hashset during the reduction step leading to the increased latency. This is seen clearly from the fact that the speedup of the parallel versions is even slower than that of the sequential version when the input file has too many unique words.

The anomaly could have been due to either aspect of the reduction step: either the raw size of the data structures being reduced or the number of keys in the data structures. To test this, we benchmarked the word count algorithm. This algorithm is widely used in real-world applications, and it lets us test a larger memory data structure for the same number of keys (due to the bytes occupied by the value). For instance, a hashset of strings occupying x bytes for the removeduplicates, would now be a hashmap of ‘x + 8m’ bytes in the word count problem.

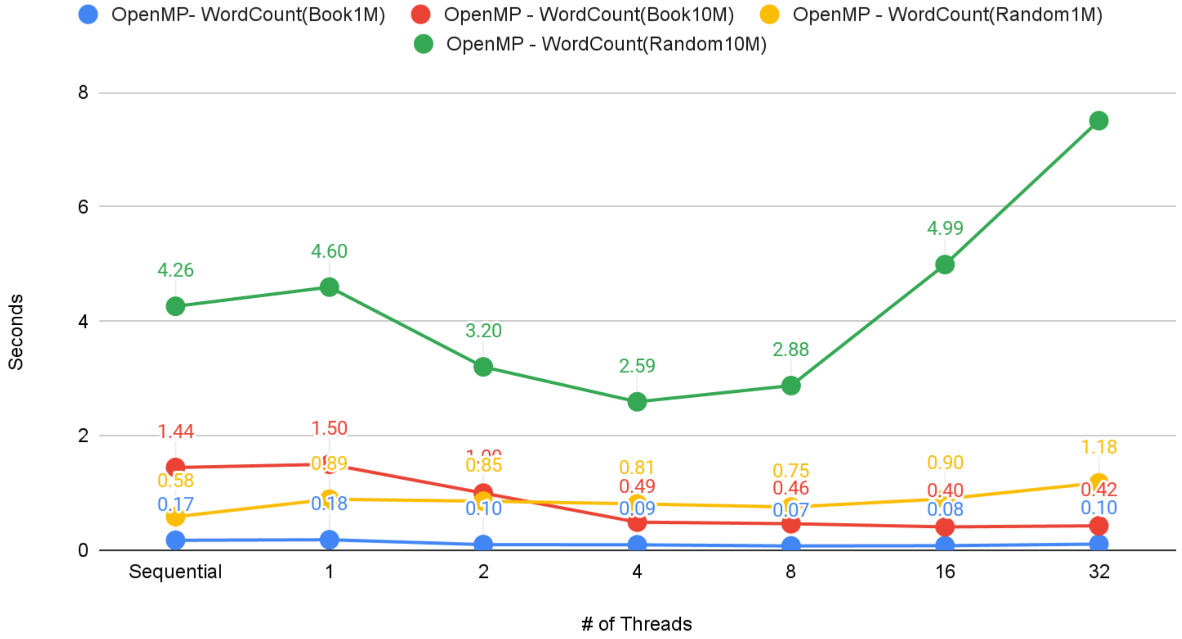
WordCount

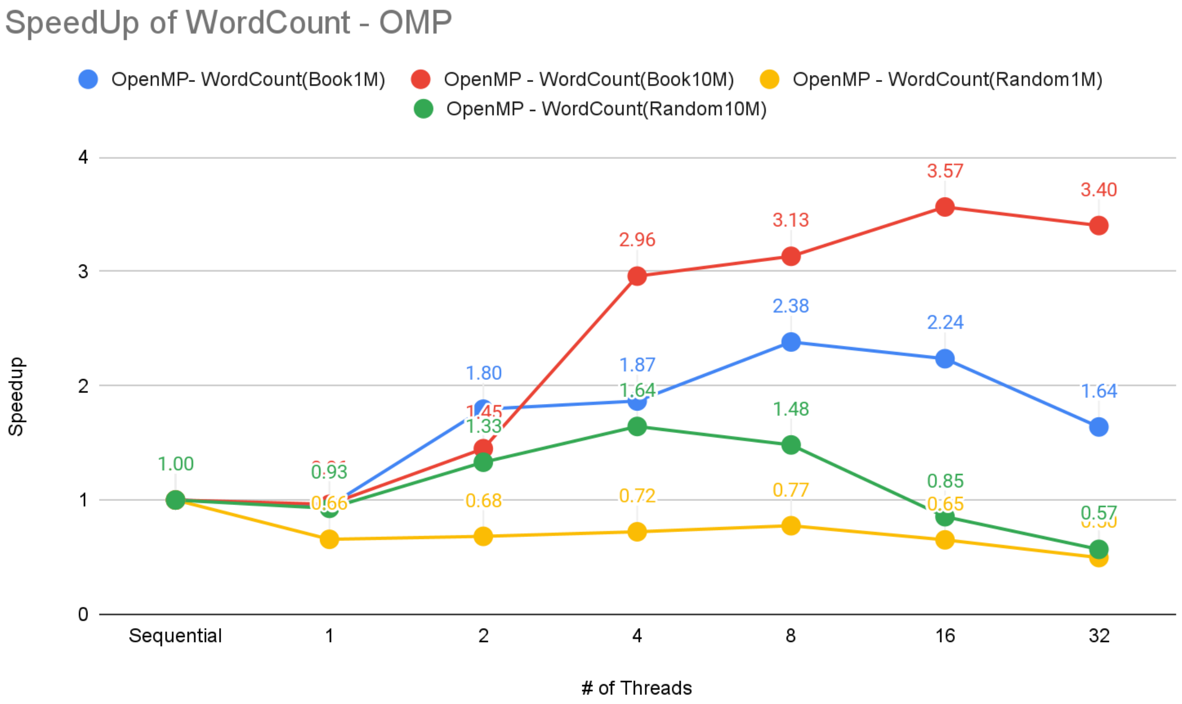


WordCount in Rust: Speedup



Execution Time of WordCount - OMP



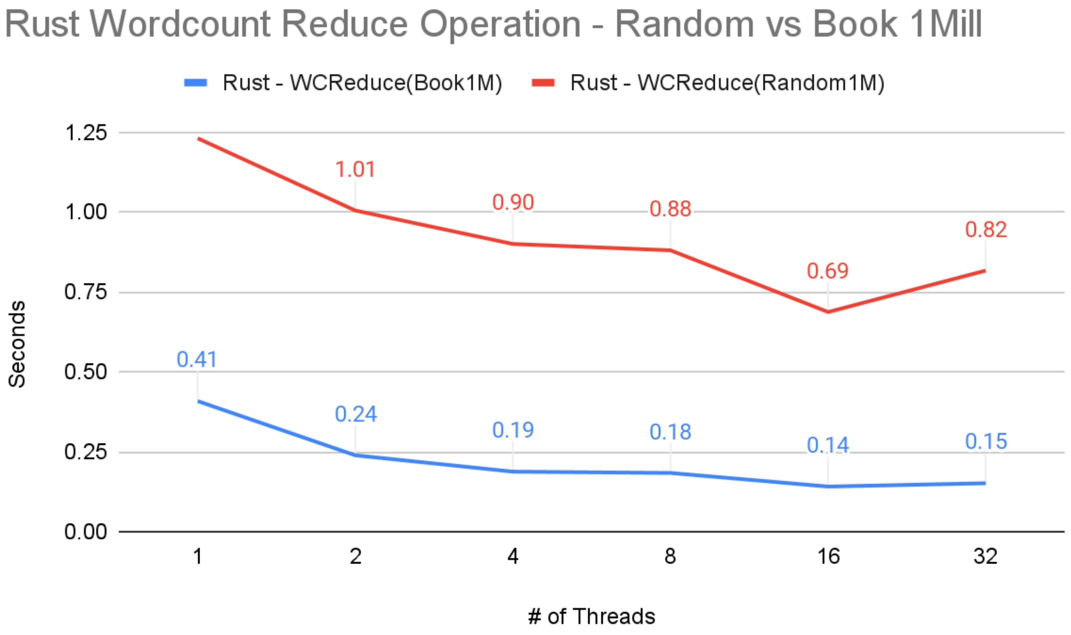


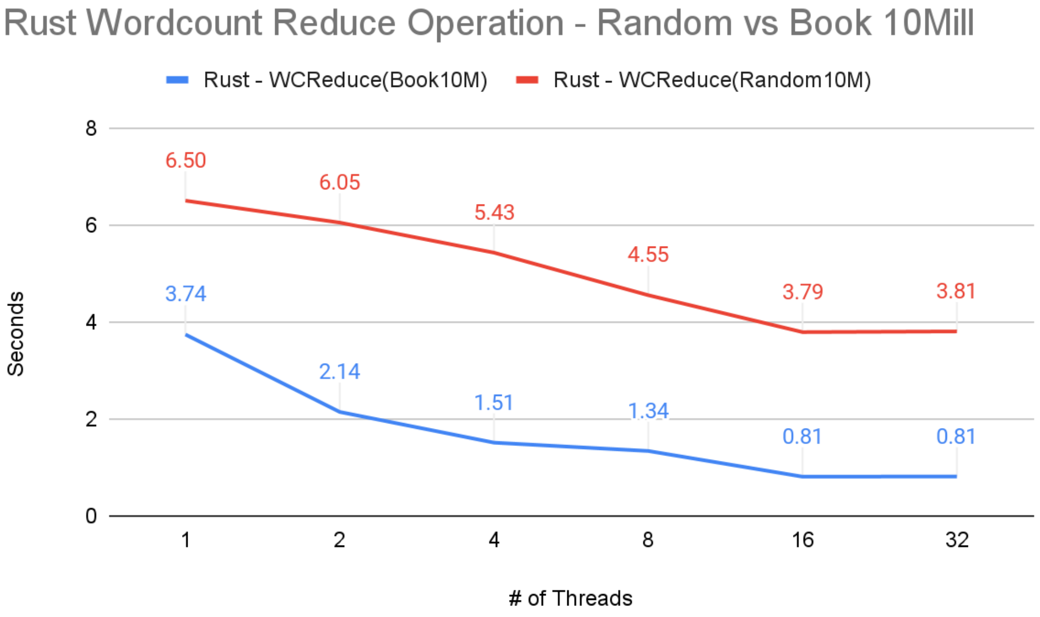
We see that the speedups of wordcount are very similar to those of the removeduplicates problem in each respective language for the same problem size and number of threads. This means that the bottleneck of the reduce step is more likely due to the number of unique keys in the data structure rather than the size of it.

Between OpenMP and Rust, Rust seems to show more consistent speedup as we increase the number of words (and uniqueness) in our document. For instance, we see that OpenMP's parallel versions are worse than sequential for both the "more unique" documents whereas Rust has that problem with only one of them. Moreover, OpenMP's speedup shows a drastic dip when moving towards 32 threads. There are two possible reasons for this:

- Rust's reduce step is on demand and does not wait for all the threads to finish whereas OpenMP's reduce step has an implicit barrier that waits for all the threads. In the case that there is load imbalance, the local hashsets in each thread are of different sizes due to the different words each thread got, the implicit barrier adds a lot of synchronization overhead that could be hurting OpenMP whereas this challenge is overcome by Rust's work stealing algorithm.
- OpenMP's reduction step involving hashset/hashmap operations does not scale well with higher number of threads, especially when the size of these data structures is very large.

We isolated the ‘reduce’ step’s time in Rust wordcount to depict the contrast between a document with few duplicate words and one with many.



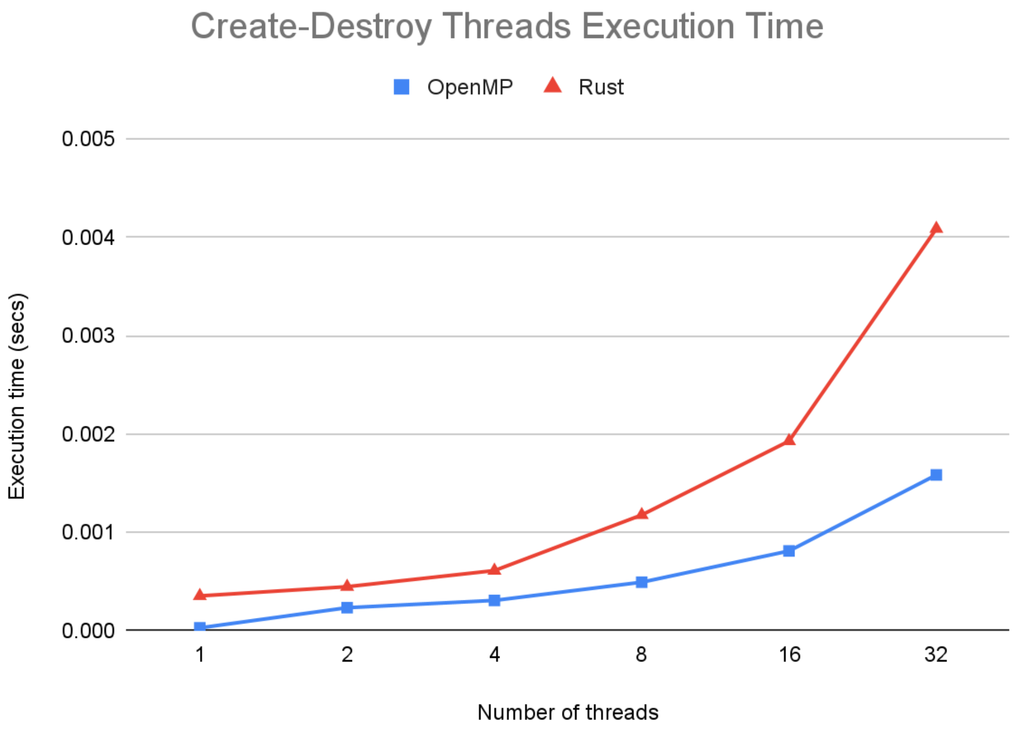


We observe that for the same size of document, the book (with fewer unique words) is consistently much faster in the reduce step than the random file from the dictionary (with many unique words). We have displayed the results for both 1 Million words and 10 Million words. The gap is much more pronounced for more unique characters.

This explains why the speedup is still increasing with an increase in number of words from the 1million book to 10 million book, unlike the plateau we see in speedup for Random 1 million and 10 million words.

Create/Destroy threads

We tested the execution time of creating and destroying threads in multiples of 2 from 1 to 32 threads. Thread creation time increases with the number of threads, this could be because even though there is a single system call, the time taken by each system call would be higher with a higher number of threads since thread creation by the OS is sequential.



OpenMP is faster than Rust, this could be attributed to the difference in design and implementation of the two languages and the fact that OpenMP has been around for many decades and is more mature and efficient compared to Rust which is relatively new. Rust also depends on an external library Rayon for creating threads which is an additional overhead as compared to OpenMP which is well-integrated with the g++ compiler. LOC is similar in both languages (14-15 lines) and the ease of creating threads is similar.

Programmability

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 like iterators and closures which makes it more expressive as compared to C++. The additional features however come with overhead and need to be evaluated for the particular use case.

Conclusions

- For purely computational tasks where we can split the task into equal compute portions, using OpenMP with the static scheduler gives better performance
- For tasks with unbalanced workloads, Rust gives better performance. OpenMP however can perform better with the right tuning, but Rust seems more robust to unknown and unbalanced workloads.
- Rust shows more consistent speedup 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 imbalance hashset reductions.

References

[1] A. Bychkov and V. Nikolskiy. Rust Language for Supercomputing Applications. Part of the Communications in Computer and Information Science book series (CCIS, volume 1510)
[2] S. Nanz, S. West, K. Silveira, and B. Meyer. Benchmarking Usability and Performance of Multicore Languages. 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement.
[3] G. Kalderen and A. From. A comparative analysis between parallel models in C/C++ and C#/Java. Link
[4] *ParallelSliceMut - Rust*. ParallelSliceMut in rayon::slice - Rust. (n.d.). Retrieved April 1, 2023, from https://docs.rs/rayon/latest/rayon/slice/trait.ParallelSliceMut.html#method.par_sort
[5] Rust documentation - https://doc.rust-lang.org/std/index.html
[6] Rayon documentation - https://docs.rs/rayon/latest/rayon/
[7] G. V. Wilson and R. B. Irvin, “Assessing and comparing the usability of parallel programming systems,” University of Toronto, Tech. Rep.CSRI-321, 1995.