Adaptive Parallel Merge Sort in Rust

Quentin Guilloteau
ENSIMAG and MoSIG
Grenoble, France
Quentin.Guilloteau@grenoble-inp.org

Supervised by: Frederic Wagner (LIG)

Abstract

In this document, we give an adaptive parallel mergesort in the Rust programming language with some optimizations to improve time performance as well as memory usage. We also present a 3-way mergesort and benchmark its performance. We will compare each version of the algorithm against the standard parallel sorts.

1 Introduction

Sorting is one of the most fundamental operations in computer science. There are many different sorting algorithms such as Insertion sort, Selection sort, Quick sort or Mergesort. In recent years, the trend among processors manufacturers has been to increase the number of core per chip instead of increasing the processor's speed. However, most of the sorting algorithms are sequential and do not use the multi-core processors of the modern machines. Not using these resources for sorting would be a shame.

The mergesort is a classic example of "divide and conquer" algorithms and is embarrassingly parallel. Thirty years ago, Cole introduced a parallel mergesort running in $O(\log n)$ time using n processors [Cole, 1986]. But parallel algorithms should not depend on the architecture of the machines they run on. That is why there have been techniques developed to make abstraction of the architecture of the machines [Traore *et al.*, 2008]. Adaptive algorithms are algorithms that can change their behaviour based on various parameters. In our case, these parameters could be the size of the array to sort or the number of threads in the system.

With the Rust programming language gaining in popularity [Stack Overflow, 2018] in production, the need for an efficient sorting algorithm is crucial. The main Rust parallel library, Rayon, claimed to have the fastest parallel sort among all parallel libraries such as OpenMP or TBB [Rayon, 2017]. One of the benefits of using Rust is the high level of abstraction available to the programmer. We could imagine writing a unique sort function that will adapt its run depending on the number of threads. If there are n idle threads trying to steal, instead of keeping splitting the work in two, the algorithm could directly split in n.

Useful Definitions

Speedup Given a sequential algorithm A_{seq} and its parallel version A_{par}, let t_{seq} and t_{par} be respectively the execution times for A_{seq} and A_{par} on the same instance. We define the speedup for A_{par}, S as:

$$S = \frac{t_{seq}}{t_{par}} \tag{1}$$

Hence,

- if $S > 1$: A_{par} performs better than A_{seq}
- if $S \leq 1$: A_{par} performs at most as well as A_{seq}

Stable Sort A sort is said to be stable if the order of elements of the same value in the input array is the same as the order of these elements in the sorted/output array.

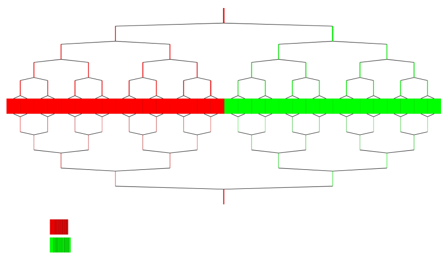
Work stealing Work stealing is the mechanism by which an idle thread of the system will get some work to do. In practice, the thread pool will define a global queue of tasks. Working threads push tasks into this queue and idle threads pop them out. The thread that pushed the task into the queue is able to know if the task has been taken or not. Originally, it was the role of the programmer to tell how and when to create tasks. However, in recent years more adaptive schedulers have been developed. Their main idea is to only create new tasks when idle threads ask for work. In the case of a single-threaded computer, an adaptive scheduler will perform as well as the sequential algorithm as no task-creation-caused overhead will be added.

Outline

Section 2 will give an overview of the algorithm and its mechanisms. In Section 3, we will present our benchmarking setup to measure the performance of the different versions of the algorithm. We will discuss in Section 4 some possible optimizations for the 2-way mergesort, before focusing on the case of the 3-way mergesort in Section 5.

2 Overview of the Algorithm

We will now give a quick overview of the algorithm.


(a) `Join`

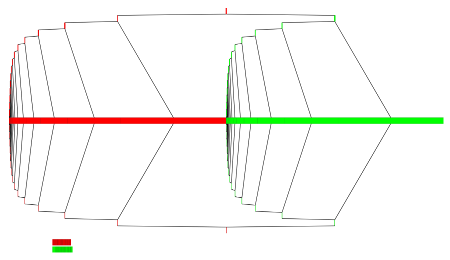

(b) `Join-Context`

Figure 1: Representation the `Join` and the `Join-Context` schedulers with 2 threads

2.1 The Data Structure

The algorithm uses a structure with a vector of 3 arrays, and an index to represent where the meaningful data is in the vector. Having 3 buffers allows us to reduce the number of data copies during the merging phase of the algorithm. Indeed, using a single buffer will force us to use a temporary buffer to store the data and then call `memcpy` to put the data back in the array. Two buffers could be enough in certain situations as we will see later. Using three buffers seems to be the best solution for now.

2.2 Possible Schedulers

There are several schedulers at our disposal to divide and distribute the work among threads. We will here present the two main ones.

Join The `Join` scheduler is the most classic one. It splits the array in two equal parts and start working on the first half, leaving the second half for any thread to take (pushing this half on the queue). This policy has a parameter which is the limit block size. When the array to split has a size that is inferior to this limit block size, it sorts sequentially the array, otherwise it continues splitting recursively.

The `Join` scheduler is fairly simple but creates a lot of tasks, as we can see in Figure 1a [Wagner, 2019], which can affect the performance. The total number of tasks created by this scheduler is $2^{\lceil \log_2 n \rceil + 1} - 1$ with n being the size of the input array.

Join Context The `Join-Context` scheduler is similar to the `Join` policy. Using `Join-Context`, it splits recursively the array and starts working on the first half, also leaving the second half for any thread to take. It stops the recursion when the size of the array is less than a limit block size. In the later case it sorts the array sequentially. The particularity of the `Join-Context` policy is that when the thread is done executing the first half of the work, if the second half has not been stolen, it sorts it sequentially instead of starting the recursive process.

This is interesting because if the second half did not get stolen, this means that there are no idle threads in the system. Thus it is not necessary to create extra tasks that the same thread will end up doing anyway. This mechanism reduces the number of tasks created as we can see in Figure 1b [Wagner, 2019]. Contrary to the `Join` policy, the tree generated by the `Join-Context` scheduler is not deterministic due to the steals. The total number of tasks created by this scheduler is at least $2\lceil \log_2 n \rceil$ and at most as much as for the `Join` scheduler: $2^{\lceil \log_2 n \rceil + 1} - 1$ where n is the size of the input array.

2.3 Merging Mechanism

In order to merge two sub arrays into one, we look first at their position in the buffers. Then we find a buffer that will receive the merged data. We cannot use a buffer where there is some data, as it will require a `memcpy` to move the data. As there are 3 buffers and 2 sub arrays, we are certain to have at least one free buffer to receive the data.

3 Bench-marking

The first step of this work was to benchmark the different mergesorts with various policies.

3.1 Experimental Setup and Tools

To measure and understand the differences between each version of the algorithms we used several tools. We will present here the two main ones.

Grid5000 Grid5000 is a large-scale and versatile testbed for experiment-driven research in all areas of computer science, with a focus on parallel and distributed computing including Cloud, HPC and Big Data. [Grid5000, 2019]

Using Grid5000 allowed us to benchmark the algorithm on a large scale of cores, and thus, helped us understand the assets and the disadvantages of each version of our algorithm depending on various factors.

In our case, we benchmarked the algorithm on diverse input sizes and input generators (random arrays, sorted arrays, reversed array...).

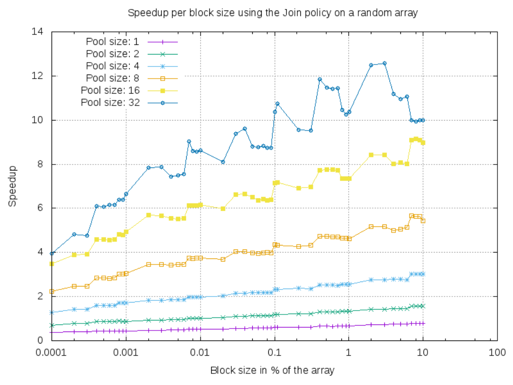
We used the `dahu` (Grenoble) cluster with the following configuration: 2 $\times$ Intel Xeon Gold 6130 with 16 cores/CPU, 32 kb L1i and L1d caches, 1 Mb L2 cache and 23 Mb L3 cache.

Rayon-Logs Rayon-Logs is a Rust library developed by my supervisor [Wagner, 2019].

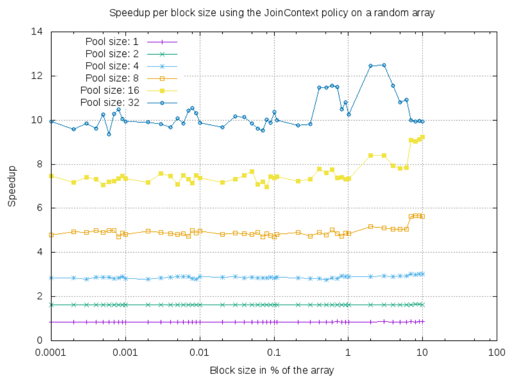
It allows the user to visualize the execution of a parallel algorithm in Rust using Rayon.

Figures 1a, 1b and 3 have been generated using this tool.

The information displayed are:



(a) `Join`



(b) `Join-Context`

Figure 2: Speedup per block size for the `Join` and the `Join-Context` policies on a random array

- the execution time of each task
- the work of each task (sub-array size)
- the thread executing each task
- the relative speed of the task compared to the other tasks of the same type

During this internship, we also had to integrate performance counters to Rayon-Logs in order to measure various metrics, like cache misses, page faults or CPU cycles.

3.2 Experimental Methodology

Every experimentation done in this study has been carried out on the Grid5000's machines (see Section 3.1) on arrays of size between 10 000 000 and 100 000 000. The elements were unsigned integers on 32 bits. Each experiment was done on 3 types on inputs: (i) Random arrays, (ii) Sorted arrays and (iii) Reversed arrays.

The random array is the most common use case for a sorting algorithm. The sorted and reversed instances were chosen to compare our algorithm with the Rayon's sort. Rayon uses a variation of the Tim sort which sorts a sorted or reversed array in $O(n)$ time [Peters, 2002].

3.3 Experimental Optimal Block Size

In this experiment we are eager to know what should be the optimal block size for each policy.

Setup We measured the speedup of the mergesort with the `Join` and `Join-Context` policy while giving different block sizes as parameter of the scheduler. We reproduced this experiment on various thread pool sizes.

Results Figure 2a represents the speedup of the mergesort using the `Join` policy with a variable block size. We can observe that there are different stages of range of block sizes for which there is not much impact on performance. Each stage represents a different depth of recursion. The parity of the depth of recursion seems to play an important role, as one stage over two has a drop in performance. Moreover, it seems that for the `Join` policy, a block size of about 2-3% of the total size is nearly optimal. There is also a drop in speedup around the 10% mark for the 32 threads large pool. This could be explained by the fact that a block size of 10% does not requires 32 working threads, thus having more than $2^{\lceil \log_2(10) \rceil}$ threads in the system has no impact.

Figure 2b shows the speedup of the mergesort using the `Join-Context` policy with a variable block size. In this case, the block size does not affect the performance significantly. Indeed, putting aside the thread pool of 32 threads, the speedup of every other thread pool is almost constant no matter the block size. In the case of the 32 threads, we see a bit more fluctuations depending on the parity of the depth of the tree. The highest speedup is reached for a block size of about 2-3% of the input array size. We also observe the drop in performance for a block size of 10%, which shares the same origin as for the `Join` policy.

Conclusion Taking a block size of about 2-3% of the input array size seems to give the best results with the `Join` and the `Join-Context` policies. It is also useless to take a block size greater than $\frac{n}{p}$, as it will leave some threads idle.

4 Optimizations

Global optimizations are difficult to find because they need to improve the algorithm on every instance of the input, or propose an interesting trade-off in performance between the types of instances. Optimizations often focus on one type of instance, e.g. already sorted arrays. We will here present some optimizations that we implemented, and compare them with other versions of our mergesort.

4.1 Extra `memcpy`

Problem As the input array points to the first buffer of our structure, we should put the sorted data back into the first buffer once sorted. However, it is not guaranteed that the data will be in this first buffer at the end of the execution. Hence, a `memcpy` of the size of the input array will be needed to move the data into the right buffer.

We can actually adjust the minimum block size to ensure that data will be in the correct buffer at the end.

The idea is to try to use only two buffers, and only rely on the third one when necessary. We have to take the depth of the task into account to know in which buffer we should merge data. As we want the last task ($\texttt{depth} = 0$) to merge into the first buffer, we will adopt the following rule:

- If the `depth` is even: then merge into buffer 1. If not possible then merge into buffer 3.
- If the `depth` is odd: then merge into buffer 2. If not possible then merge into buffer 3.

Finding the new block size Let $\mathcal{A}$ be a k-way mergesort with $k \geq 2$. Let n be the size of the input array and B be the block size associated with the scheduler. Under these conditions, the number of recursions levels (R) is:

$$R = \left\lceil \log_k \left(\frac{n}{B} \right) \right\rceil \tag{2}$$

We want to alternate between two buffers and only need the third one if necessary. Thus, the number of recursions has to be even. Let R' be the closest even number to R.

$$R' = \left\lceil \frac{R}{2} \right\rceil \times 2 \tag{3}$$

The new block size B' for this number of recursions is thus

$$B' = \left\lceil \frac{n}{k^{R'}} \right\rceil \tag{4}$$

Performance This optimization achieves a gain in performance compared to the original version of about 10% for random arrays, 20% for reversed arrays and no real gain for sorted arrays.

4.2 From 3 Buffers to 2 Buffers

Problem Having 3 buffers can have a real memory cost. Indeed, if we are sorting an array of size 100 millions integers on 32 bits, one buffer takes $100 \times 10^6 \times 4 = 400$ Mbytes. Thus three of these will take 1.2 Gbytes. In this section, we will try to reduce the number of buffers to only 2 by modifying the value of the block size.

During the benchmarks, we spotted a strange behaviour from the algorithm thanks to a Rayon-Logs (see Section 3.1) log. If the policy is `Join-Context`, then the merging time of the sub arrays of the thread takes more time than if it has been stolen. The fact is that the buffers are so large, that loading one causes page faults, and thus impacts the performance. We can see this impact on the log in Figure 3. The red and green threads are actually slower (larger rectangle) than the blue and yellow ones (shorter rectangle) during the merging part. Indeed, the first two merges of the red and green threads require to load the 3 buffers, where the two first merges from the blue and yellow threads only need to load two.

The third buffer is needed when we are merging data from different depth parity. This could happen when at least one sub array has a size greater than the limit block size and at least one does not. The second way this could happen is when we are using the `Join-Context` scheduler and there is no steal. The deepest merge will put the data in the second buffer, but the data from the un-stolen sequential part is in the first buffer, hence the merge operation will need to use the third buffer. Figure 4 resumes this situation.

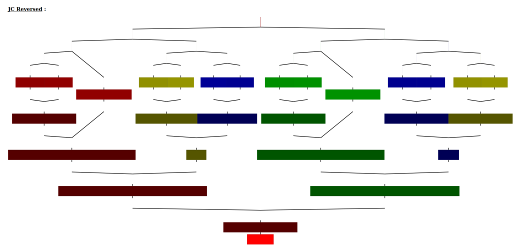


Figure 3: Example of the impact of using 3 buffers

Solution `Join`: adapting the block size We want to ensure that we will not have to merge data from different depth parity. Let $\mathcal{A}$ be a k-way mergesort with $k \geq 2$ using the `Join` policy. Let the input array be of size n. As we are splitting in k, there are $(k-1)$ sub arrays of size $n_1 = \lfloor \frac{n}{k} \rfloor$ and one sub array of size n_2 where $(k-1) \times n_1 + n_2 = n$.

At every level of recursion the maximum difference between the sizes of the sub arrays is $k-1$. Indeed, suppose it is greater than k, then we could distribute the difference among the k sub arrays. Thus:

$$|n_1 - n_2| \leq k - 1 \tag{5}$$

As $n_1 \leq n_2$, we have several cases:

- Case 1: $n_2 > B, n_1 \leq B$: there is a different level of recursion between the sub arrays, and thus we need the third buffer.
- Case 2: $n_2 \leq B, n_1 \leq B$: we do not need a third buffer.
- Case 3: $n_2 > B, n_1 > B$: we continue the recursion.

Hence, the limiting case is when exactly one of the sizes is greater than the limit block size B. Let us take the case 1
We have $n_2 > B \geq n_1$
We want to increase B to B' such that $B' \geq n_2 > n_1$.
As $n_2 > n_1$, we have $1 \leq |n_1 - n_2| \leq k - 1$.
Thus, a valid candidate for B' would be:

$$B' = B + (k-1) \tag{6}$$

We can use the result from Section 4.1 to find a block size avoiding the extra `memcpy` and necessitating only two buffers.

Let R' be as defined in Equation (3).
Thus, the new block size B is:

$$B = \left\lceil \frac{n}{k^{R'}} \right\rceil + (k-1) \tag{7}$$

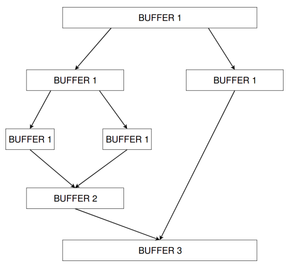


Figure 4: Situation where 3 buffers are needed with the `Join-Context` policy

Solution `Join-Context`: Alternate the Schedulers To avoid the use of the third buffer, we will need to alternate the `Join-Context` scheduler with the `Join` scheduler. We also need to ensure that the last level of recursion is a call to the `Join-Context` scheduler so that we do not end up in the situation depicted in Figure 4. By alternating the calls to the two schedulers, we are sure that the data will be in the same buffers when merging sub arrays.

Let $\mathcal{A}$ be a k-way mergesort with $k \geq 2$ using the `Join-Context` policy. We know the block size B and the size of the array n, hence we can compute the height R of the recursion tree. The last deepest call to the scheduler ($\texttt{depth} = R$) should be using the `Join-Context` policy to avoid the situation presented in Figure 4. Then the rule is:

- If the `depth` is odd: call the `Join` scheduler.
- If the `depth` is even: call the `Join-Context` scheduler.

As the `Join-Context` scheduler could give the same tree as the `Join` scheduler if there are enough steals, we also have to adjust the block size B in the same way as seen in Section 4.2.

Let R' be as defined in Equation (3) (R' is even by definition). The new block size B is also: $B = \left\lceil \frac{n}{k^{R'}} \right\rceil + (k-1)$

4.3 Sorted and Almost Sorted Arrays

Problem When sorting an already sorted array, our algorithm will perform the merge operation even if the two sub arrays are in order. This greatly impacts the performance of our algorithm in this case. Indeed, the sequential sort used by Rust is the Tim Sort. The Tim Sort is extremely good at sorting already sorted arrays in linear time. Therefore the bottleneck of the algorithm in this case is the merging operation.

Checking if the sub arrays are in order before merging When merging two sub arrays, we can check whether or not they are already in order. If both sub arrays have their data in the same buffer, we only have to look at the last element (l_1) of the first array and the first element (f_2) of the second array. If $f_2 < l_1$ then the arrays are already ordered, and we thus do not have to merge them. This allows us to not have to move all the elements in another buffer (i.e. avoid `memcpys`).

If the input array is already totally sorted, then we only use one buffer as there will be no merge. However, if the input array is "almost" sorted (i.e. sorted with a given percentage of the elements swapped together) then this solution ruins the optimizations done in Sections 4.1 and 4.2.

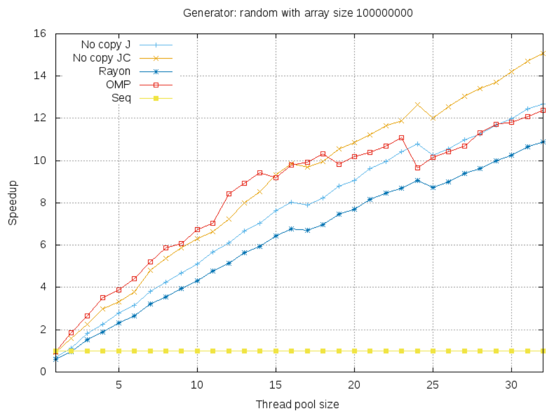


Figure 5: Speedup comparison of the different versions of the algorithm for a random array

Performance For testing this optimization, we can not use a random nor a sorted array. Indeed, the random array has a small probability to have two sub-arrays in order when sorting. On the other hand, the sorted array will only have ordered sub-arrays. Therefore, we will benchmark this version on *almost* sorted arrays. The results show that for a percentage of elements swapped between 1% and 10%, the performance of this version are within 5% of the performance for the version seen in sections 4.1 and 4.2. We have then integrated this optimization in the core of the algorithm.

4.4 Performance

Figure 5 compares the different versions of the algorithm on a random array and compare them to the parallel sorts from Rayon (Rust) and OpenMP (C/C++). We observe that the version with no copy using the `Join-Context` policy out performs every other version of our algorithm. It is also better than the Rayon's sort but worse than the OpenMP sort until a thread pool size of about 20. Concerning sorted arrays, the versions using the `Join-Context` policy perform better than the others (Rayon and OpenMP included). This could be explained by the fact that the `Join-Context` policy spends more time using the sequential Tim sort (which is highly efficient in the case of sorted arrays). Regarding reversed arrays, every version that we tried performed badly, with a speedup inferior to 1.

5 3-way Mergesort

The classical mergesort splits the array in two. However, when the number of available threads is for example 3, it does not really make sense to split in half. Indeed, 3 threads will require two cuts, leaving one thread with half of the original work and the other two threads with a quarter of it. This approach will unbalance the work between the threads. Creating tasks has a non negligible cost, this is why giving the

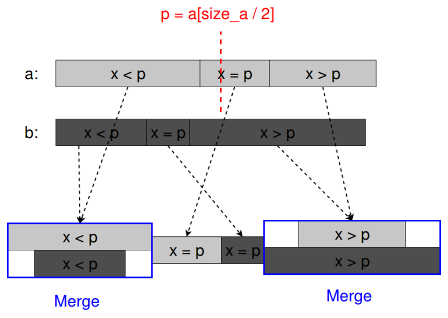


Figure 6: Recursive Merge Mechanism

maximum possible workload to each thread is important for performance.

We decided to **focus on a 3-way mergesort** to see if we could gain any performance compared to the classic 2-way mergesort.

5.1 Choice of the parallel merge

[Cormen *et al.*, 2011] presents a parallelizable recursive stable 2-way merge algorithm. We remind its mechanism in Figure 6.

3-way Parallel Merge We can perform a 3-way parallel merge by splitting the sub arrays in 3 parts. There are two pivots in this case. The mechanism is the same as for a 2-way recursive merge.

2-way Parallel Merge As the merge operation is less time consuming than the sort operation for the same input sizes, can we gain in performance by only using a 2-way merge with a 3-way split ? We could leave only 2 threads merging in parallel, while one thread continues the sorting part. Then, when all the sorting is done, there will be however, more steals and the work distribution will be unbalanced between the threads.

5.2 Optimizations

If we do not use the optimizations discussed in Section 4.3 for sorted arrays, we could use the following ones:

- Avoid the extra `memcpy`: as seen in Section 4.1
- Only use 2 buffers: as seen in Section 4.2, we can adjust the block size and the calls to the schedulers to only use 2 buffers

5.3 Performance

Figure 7 presents the comparison of the different versions of the 3-way mergesort on a random array. The 3-way mergesort (Join 3/3) performs better than the classic 2-way mergesort (2/2 Join). It seems that the 2-way merge on a 3-way split do not improve the performance of the algorithm (3/2 Join). By avoiding the last `memcpy` (3/3 Join no copy), we get a significant gain of speedup compared to the classic 3-way mergesort (3/3 Join) of about +1 for 32 threads. However, only using two buffers (3/3 Join no copy + 2 buffers) does not increase the speedup. It is nice to see that the two later versions are competitive with Rayon's sort up to a dozen threads.

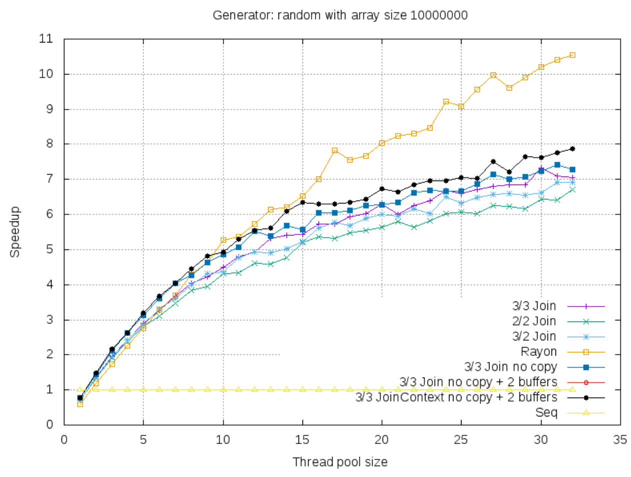


Figure 7: Comparison of the different versions of the 3-way mergesort

Unfortunately, all of these versions are extremely bad (worse than the sequential algorithm) in the cases of sorted and reversed arrays, as we did not implement any optimization for this case.

6 Conclusion

We presented an adaptive parallel 2-way mergesort. We highlighted several optimizations to improve speed and memory usage. We also introduced a 3-way parallel mergesort to see if any gain of performance was possible. We compared the different versions of the algorithm to the parallel sorts from libraries such as OpenMP or Rayon. Even if our algorithm is not as efficient in the cases of reversed and sorted arrays, we managed to compete, and in certain cases, to beat parallel sorts from standard libraries.

References

[Cole, 1986] Richard Cole *Parallel Mergesort*

[Grid5000, 2019] Grid5000 *https://www.grid5000.fr/*

[Wagner, 2019] Rayon-Logs *https://github.com/wagnerf42/rayon-logs*

[Wagner, 2019] Frederic Wagner's homepage *http://www-id.imag.fr/Laboratoire/Membres/Wagner_Frederic/*

[Peters, 2002] Timsort *https://bugs.python.org/file4451/timsort.txt*

[Rayon, 2017] Rayon Team *https://github.com/rayon-rs/rayon/pull/379*

[Cormen *et al.*, 2011] Introduction to Algorithms (Third Edition), Chapter 27, Section 3 *MIT Press*

[Traore *et al.*, 2008] Algorithmes Adaptatifs de tri parallele *MIT Press*

[Stack Overflow, 2018] StackOverflow survey *link*