# Multi-Threaded Merge Sort

Ernani Raffo

March 2023

## 1 Introduction

Merge Sort is a very efficient sorting algorithm with its time complexity being $O(n \log n)$. In this project, we explore the speed of Merge Sort both as a sequential and multithreaded implementation.

## 2 Sequential Merge Sort

The function `MergeSorter::MergeSort(std::vector<uint32_t> &A)` takes in as parameter the array to be sorted, and performs a merge sort as described in *Introduction to Algorithms* by Cormen et. al. In their pseudo-code, arrays are indexed at 1 rather than 0, so inside the function mentioned, it calls the following overloaded method

```
MergeSorter::MergeSort(std::vector<uint32_t> &A, uint32_t p, uint32_t r)
```

with the first parameter being the array to sort, `A`, the second being the lower bound of the array, `1`, and the third being the higher bound, `A.size()`. Since the pseudo-code works with arrays indexed at 1, we access elements inside `A` by decrementing the index we want to access by 1. For example, for an index `i` we would need to write `A[i-1]`.

By default, when running the executable `sort`, the program will only run the sequential merge sort algorithm on `A` with 100 elements. Below is an example output of running `sort`.

```
$ ./sort
Merge Sort, 100 elements, 0.000528167 seconds
    12105075    20544909    58150106    76338300   156513983
   219972873   290319951   471852626   483031418   537655879
   540721923   581869302   640439652   663307952   678852156
   746745227   809094426   893645500   902841100   910208076
   933029415   949333985   988512770  1275731771  1287767370
  1296707006  1323567403  1359573808  1427854500  1515103006
  1530490810  1551745920  1663423246  1668894615  1685003584
  1696117849  1712568902  1736062366  1755486969  1772389185
  1812852786  1817480335  1958646067  1967048444  2039073006
  2094092595  2107063880  2163214728  2265043167  2350294565
  2411870849  2464257528  2495189930  2546159170  2553373352
  2747762695  2765791248  2850164008  2919803768  2926416934
  2986002498  3031277329  3052449900  3117454609  3122246755
  3147346559  3181055693  3271610651  3280281326  3323948758
  3345340191  3408475658  3410096536  3424291161  3427077306
  3427838553  3468319344  3471087299  3477783405  3525484323
  3527224675  3530047642  3586334585  3644141418  3747053250
  3765644424  3772830893  3897101788  3955127111  4037587209
  4101769245  4144499009  4156218106  4161255391  4213834039
  4241106803  4245667946  4264392720  4269491673  4279768804
```

## 3 Multi-Threaded Merge Sort

Merge Sort is nice for multi-threading since we can split an array `A` by the amount of cores we want to use for multi-threading. If we had an array with 1000 elements and we had 4 cores, we can run a merge sort on 4 different sub-arrays each of length 250 in parallel. Once each sub-array is sorted, we must merge them back together. In this project, we do this by $k$-way merging. This algorithm is implemented by performing an iterative 2-way merge, until only one list is left and the merging is complete. Since we are doing this in place, we keep track of the indices of where each thread split the array, and we merge until the indices of the first sub-array contains all elements of `A`.

In the first implementation of a multi-threaded merge sort,

```
MergeSorter::ParallelMergeSort(std::vector<uint32_t> &A, uint32_t cores),
```

the $k$-way merging step happens sequentially, which is not as optimal as we want it to be. The figure below shows the time differences between the first attempt at multi-threading merge sort versus sequentially merge sorting.

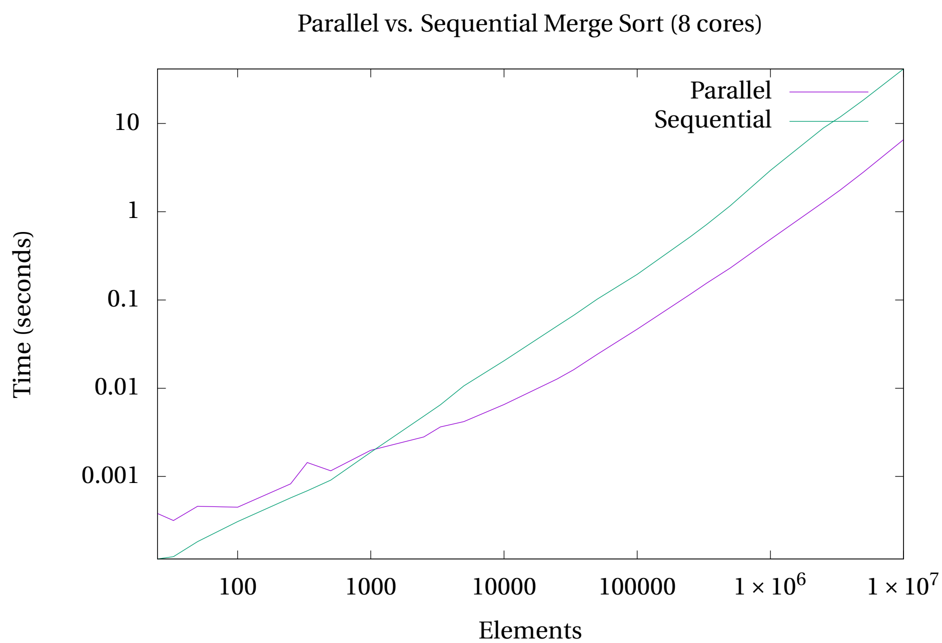


Figure 1: Sorting `A` up to 10,000,000 elements

We can see that with less than 1000 elements, the parallel implementation is actually worse than the sequential version since it needs to merge the lists back and it does so in a sequential manner. To optimize our parallel implementation, we must implement a threaded version of $k$-way merging. Note that $k$-way merging contains $\log k$ steps and in each step, we merge $k/step$ lists together to get $\lceil (k/step)/2 \rceil$ remaining lists for the next step. Therefore, in each step, we can start threads for each merges we need to do. In this manner, our implementation is truly parallel, and much more optimized than before. This approach is implemented in

```
MergeSorter::OptimizedParallelMergeSort(std::vector<uint32_t> &A, uint32_t cores).
```

But can it get even better? Yes. So far, dividing the work between threads is not evenly distributed. For example, let's say we had 96 cores available to us and we needed to sort an array with 1,000,000 elements. This would mean that each thread would be sorting an array with 10,416 elements, and the last sub-array would contain $1,000,000$

mod 96 = 64 more elements to sort. What we would like to have instead, and what is implemented in

```
MergeSorter::OptimizedParallelMergeSortV2(std::vector<uint32_t> &A, uint32_t cores)
```

is for the first 64 threads to contain only 1 more element to sort, and the remaining 32 threads to have exactly 10,416 elements to sort. In this manner, the program does not have to wait longer for the last thread to finish sorting more elements than the others, as they will finish at approximately the same time.

When the remainder is small, however, there is not much difference in the time it takes to finish sorting. This project was run on a MacBook Pro with an M1 Pro chip and 8 cores, meaning that the worst possible remainder in terms of dividing the work is 7. This remainder is so small that both optimized versions of parallel merge sorting take about the same time, with either being better than each other.

Below are figures showing benchmarks between all merge sorting techniques implemented. `Optimized Parallel Merge Sort` refers to the first optimization and `Optimized Parallel V2` to the second optimization which builds off the first by adding the evenly divided work-load between threads aspect.

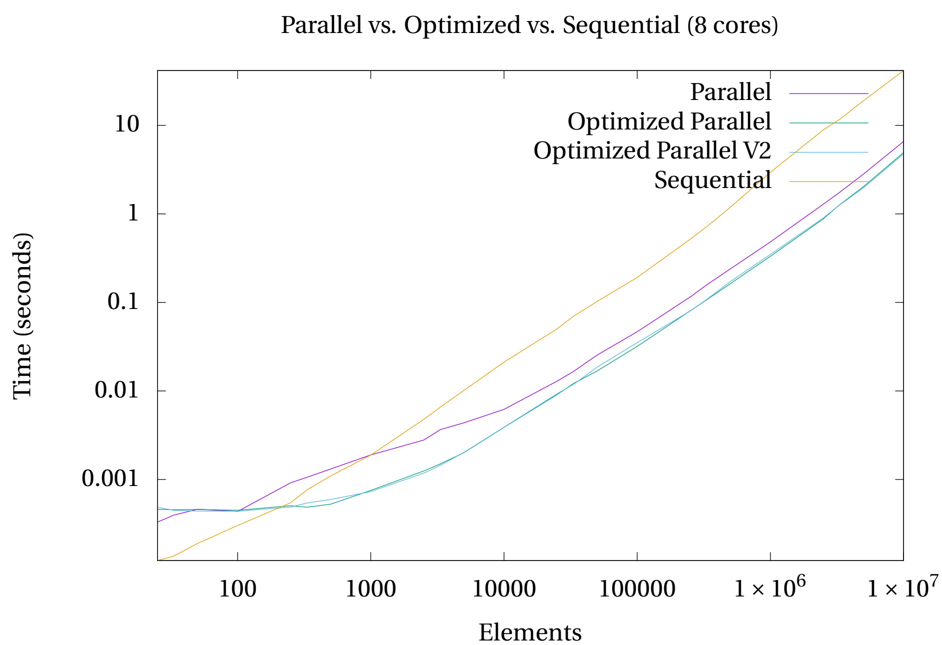


Figure 2: Sorting `A` up to 10,000,000 elements with all implementations

In the above figure, we can see how the optimized versions of multi-threading merge sort are much better than the first parallel method. Though, with `A` containing less elements, they are actually slightly worse since we are creating threads for sub-arrays with barely any elements to sort. It can also be seen that the optimized versions of parallel merge sorting become better than the sequential version much quicker than the first parallel implementation.

In figure 3 and figure 4, we take a closer look at more points for the time it takes to sort `A` with 10,000 to 1,000,000 elements and `A` with 1,000,000 to 10,000,000 elements. From both graphs, we can see how with 8 cores, the optimized versions of the parallel implementation fight with each other to be faster. If we had more cores, the second version would definitely be better when the remainder is higher, and would be about the same as the remainder is closer to 0. The biggest change in optimization is when the $k$-way merging method was implemented in a multi-threaded manner. We can see that both optimized versions perform *much* better than the first parallel version with very large arrays.

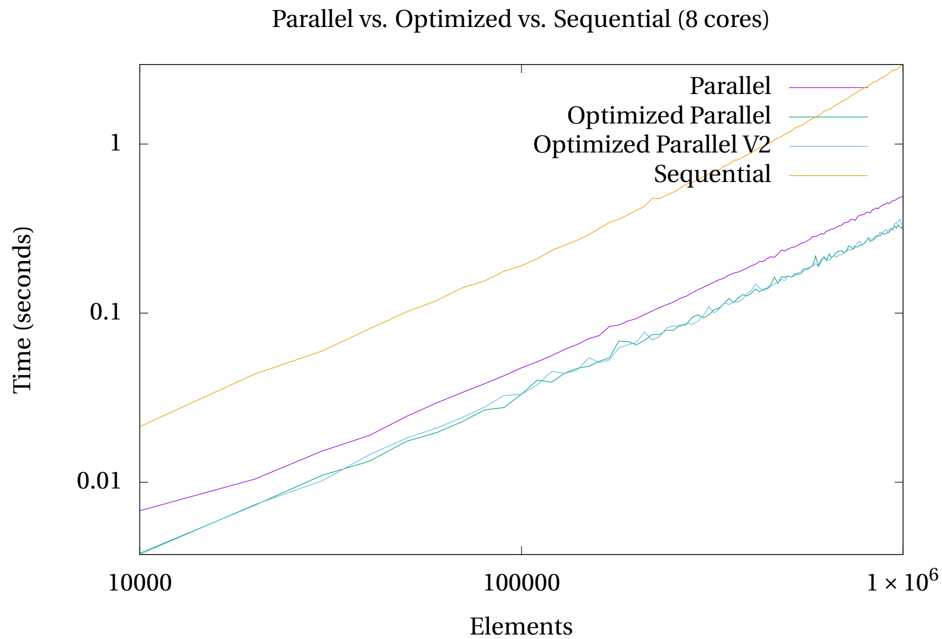


Figure 3: Large arrays

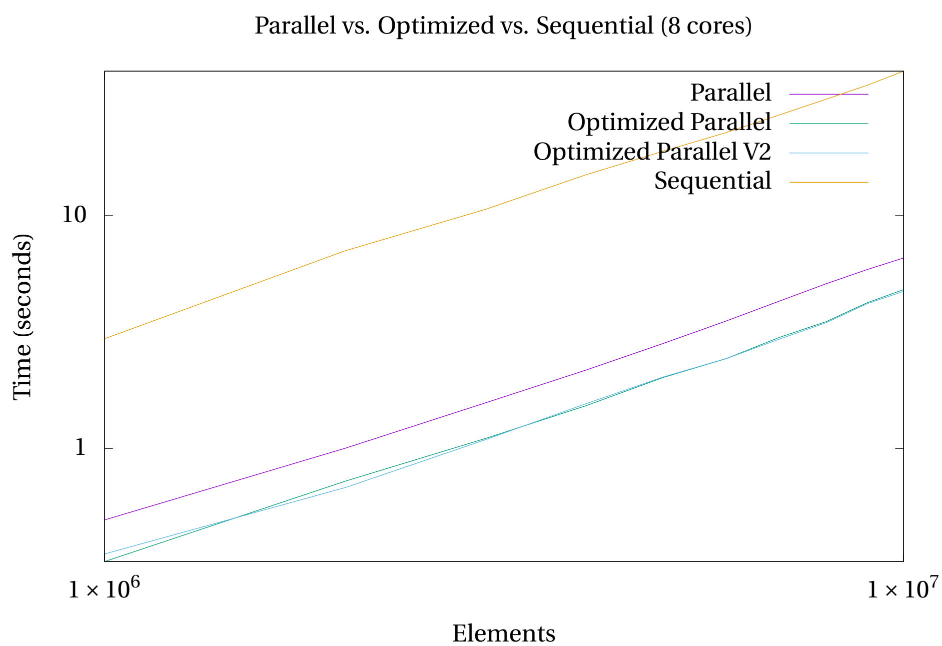


Figure 4: Very large arrays

Lastly, we take a look at the worst possible remainder for a multi-threaded merge sort on 8 cores. Each point corresponds to the $x$ position minus 1. Therefore, the data points are 99, 999, 9999, 99999, and 999999. On 8 cores, each have a remainder of 7, and we can see how the second version of the optimized implementations performs slightly better than the first, but later on, the time between the two are about the same, with the first sometimes being better. Again, if we had more cores and the remainder was closer to the amount of elements to sort, the second version would perform much better, as it would not need to wait for the last thread to finish sorting its elements plus the remainder.

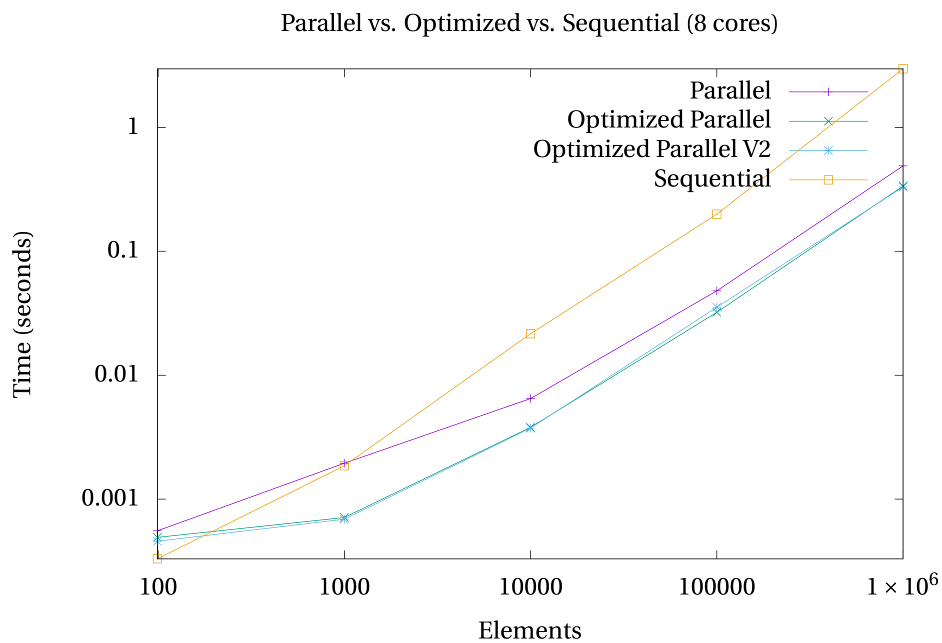


Figure 5: Arrays with worst possible work distribution

# 4 Conclusion

Overall, a multi-threaded approach to merge sorting is very efficient, and with more cores, a greater speedup can be achieved for arrays with a numerous amount of elements. A drawback of multi-threading merge sort is that sub-arrays need to be merged again when coming back from their original fork. If this is multi-threaded as well, the problem is greatly remedied, but it would be better if the array, `A`, was already sorted in place when coming back from the original fork. Such a method can be done by a MSD (Most Significant Digit) radix sort. Nevertheless, a parallel merge sort is *very* fast compared to its sequential form.