Exploring Timings of Popular Sorting Algorithms and Introducing chulevsort

Research Paper · September, 2023

Utrecht University,

University College Roosevelt,

Middelburg, The Netherlands

Author:

Joanikij Chulev

E-mail: j.chulev@ucr.nl

Abstract

This study presents a comprehensive analysis of various sorting algorithms to evaluate their performance in terms of execution time. Three experiments were conducted: the first experiment compared sorting algorithms on randomly generated data, while the second experiment assessed the efficiency of parallel sorting algorithms on large datasets. For the third test we introduced our own sorting algorithm. In the first experiment, a set of random data arrays were sorted multiple times using each algorithm, and the average execution times were recorded. This experiment provided insights into how these algorithms perform on average-case scenarios. The second experiment focused on parallel sorting algorithms, including Parallel Merge Sort, Parallel Quick Sort, and Python's general parallel sorting capabilities. A large dataset was used to assess the algorithms' performance in a parallel computing environment, with the goal of exploiting multiple CPU cores to expedite the sorting process. Finally, we implemented our sorting algorithm called chulevsort which is described in detail and compared to other sorting algorithms in regards of execution times. The results revealed significant differences in execution times among the sorting algorithms, highlighting the trade-offs between simplicity and efficiency.

Contents

Experimental platform

Hardware technology platform:

OS Name: Microsoft Windows 11 Pro

Windows version: 22H2

System Type: x64-based PC

Processor: Intel(R) Core (TM) i7-1065G7 CPU @ 1.30GHz, 1498 Mhz, 4 Core(s), 8 Logical Proc.

Installed Physical Memory (RAM): 16.00 GB

GPU: Integrated Intel GPU (Intel Graphics)

Software technology platform:

Spyder version: 5.4.3 (conda)

Python version: 3.9.17 64-bit

Qt version: 5.15.2

PyQt5 version: 5.15.7

Operating System: Windows 11

1 Experiment 1

In Experiment One, we conducted a comparative analysis of various sorting algorithms on randomly generated data to investigate their performance in terms of execution time. The data was randomly generated via Pythons built-in functions and the data allows for duplicate values. The primary aim of this experiment was to determine whether there are statistically significant differences in the average execution times among the selected sorting algorithms. The hypotheses are formulated as follows:

$H0$ - no significant difference in execution times between the sorting algorithms

Ha – algorithms have a significant difference in execution times and one outperforms

By running each algorithm multiple times and recording their average execution times, we are able to rigorously assess whether any of the algorithms outperformed the rest or if they exhibited similar levels of efficiency in handling random data sets. The sorting algorithms that are going to be compared are Bubble Sort, Selection Sort, Insertion Sort, Merge Sort and Quick Sort. Timing is going to be recorded within Python. Each sorting algorithm has its own time complexity. These time complexities are given in Figure 1 below.

Name of the algorithm	*Average case time complexity*	*Worst case time complexity*	*Stable?*
Bubble sort	$\Theta(n^2)$	$O(n^2)$	Yes
Selection sort	$\Theta(n^2)$	$O(n^2)$	No
Insertion sort	$\Theta(n^2)$	$O(n^2)$	Yes
Merge sort	$\Theta(n \log_2 n)$	$O(n \log_2 n)$	Yes
Quick sort	$\Theta(n \log_2 n)$	$O(n^2)$	No

Figure 1: Time complexity and information on tested algorithms

1.1 Code Analysis

The objective of Experiment 1 is to evaluate the performance of different sorting algorithms. The experiment starts by generating a random array of numbers as the input data for sorting algorithms. This generation is from integer values ranging from -100 to 100. We chose both negative and positive values for diversity and complexity, allowing duplicates. Each algorithm was executed 100 times; thus 100 test runs to have less deviation in time and more standardized results. The array size was varied from 10 to 100 to 1000 to 10000 array items. A mean timing from every test is extracted for each algorithm to get an average time. Then all the average timings were compared. It also identifies and prints the "winner," which is the sorting algorithm with the shortest average execution time.

We implement the measuring time function in the code, the loop for test runs, timing printouts, and finally the algorithms we actually compare. Each algorithm is implemented separately as functions, mainly: def bubble_sort(array), def selection_sort(arr), def insertion_sort(array), def merge_sort(array), def quick_sort(array).

All of these algorithms were taken from Python, R. (2023). and implemented in the code manually and adjusted. The quality of these algorithms seems to be on par as from sourcing from GitHub and anecdotes. All the other code we mentioned, except from the algorithms is written and mine (explained in comments in the py files). Below in Figure 2, you can see an output of the code that was produced.

```
bubble_sort: Average Execution Time = 0.062803 seconds
selection_sort: Average Execution Time = 0.025787 seconds
insertion_sort: Average Execution Time = 0.031297 seconds
merge_sort: Average Execution Time = 0.003291 seconds
quick_sort: Average Execution Time = 0.000933 seconds
The winner is quick_sort with an average execution time of 0.000933 seconds.
```

Figure 2: example output of experiment 1 with 100 test runs

1.2 Table Timing Results

Here we have the executing timing results when varying the data size of the array. Each algorithm is listed as an attribute and time is measured in seconds. The results can be seen in Figure 3.

Array items (n)	Bubble Sort	Selection Sort	Insertion Sort	Merge Sort	Quick Sort
10	0.000009	0.000009	0.000007	0.000021	0.00001
100	0.00059	0.00026	0.00028	0.00025	0.00017
1000	0.060301	0.025112	0.029378	0.003328	0.001089
10000	6.297506	2.592186	3.03399	0.045856	0.008059

Figure 3: Execution timing results table for Experiment 1

1.3 Graph Timing Results

Here we have graphs and details of the performance of the execution times of all algorithms that were tested. Figure 4 represents how the algorithms perform in array sizes 10 and 100. We can see the performance of size 10 is similar. Although even at 100 data points we can see that the lines start to diverge and we can see clear performance differences. Figure 5 does the same for array sizes up to 1000. And figure 6 displays the final results where we can see clear "winners". Consult figures below.

Figure 4: Execution time performance differences graph for array sizes 10 and 100

Figure 5: Execution time performance differences graph for array sizes 10, 100, and 1000

Figure 6: Execution time performance differences graph for all array sizes

1.4 Conclusions

Based on the previous results and mentioned info we can firmly reject the null hypothesis and accept the alternative hypothesis. The results of Experiment 1 demonstrate significant variation in the performance of the tested sorting algorithms when applied to randomly generated data. Each algorithm exhibited different average execution times, highlighting that the choice of sorting algorithm can have a substantial impact on the efficiency of data processing tasks. Among the algorithms evaluated, Bubble Sort consistently proved to be the least efficient in terms of execution time. In contrast, Merge Sort and Quick Sort consistently outperformed the others, showcasing their suitability for sorting tasks involving randomly ordered data. Quick Sort was the overall winner, outperforming every time and every instance of the testing. The findings from Experiment 1 provide valuable guidance for algorithm selection based on timing. Future work could extend this analysis to explore the performance of these algorithms on different data distributions (e.g., sorted, reversed, nearly sorted), larger datasets and different data types. Additionally, experiments in specific application domains could help tailor algorithm selection to specific needs.

2 Experiment 2

In Experiment Two, we extended our investigation by evaluating the performance of parallel sorting algorithms on a large dataset. Parallel algorithms for sorting are of a recent origin and came into existence over the past decade (Lakshmivarahan et al., 1984). The data was randomly generated via Pythons built-in functions and the data allows for duplicate values. The experiment sought to investigate whether parallelism offers significant advantages in terms of execution times when sorting large volumes of data. The hypotheses are formulated as follows:

$H0$ - no significant difference in execution times between the parallel sorting algorithms

Ha - one parallel sorting algorithm would demonstrate significantly reduced execution times

This experiment aimed to shed light on the effectiveness of parallel sorting algorithms, which are not that often discussed. Because of the data amount, we will run these algorithms only 20 times and see each result. The sorting algorithms that are going to be compared are Parallel Merge Sort and Parallel Quick Sort. As in experiment 1, timing again is going to be recorded within Python. Each parallel sorting algorithm also has its own time complexity. Parallel merge sort divides the input into p subarrays, sorts them in parallel using a sequential merge sort, and then merges them in parallel using a binary tree (Programming, 2023). It has a time complexity of $O(n\,logn)$. Parallel quicksort partitions the input around a pivot element, recursively sorts the two subarrays in parallel, and then concatenates them (Programming, 2023). It has a work complexity of $O(n\,logn)$.

2.1 Code Analysis

The experiment starts by generating a large dataset, which serves as the input data for sorting algorithms. The dataset is intentionally designed to be substantial to evaluate the algorithms' performance under the constraint of processing a significant amount of data. This generation is from integer values

ranging from -10000 to 10000. We chose both negative and positive values for diversity and complexity, allowing duplicates and we also increase the range for more data diversity. Array sizes of 10000, 100000, and 1000000 were chosen. For bigger sizes, computation on my device took too long. Each algorithm was executed only 5 times due to the computational power required. The mean is than calculated and a "winner" is declared, very similar to experiment 1.

As in experiment 1, we implement the measuring time function in the code, the loop for test runs, timing printouts (Note, each timing of each run is printed now) and finally the two algorithms we actually compare. Each algorithm is implemented separately as functions, mainly: parallel_merge_sort and parallel_quick_sort.

These algorithms were upgraded to support and use parallelism from the original sequential algorithms that were used in experiment 1, respectively. This was done with manual coding and with help from a colleague. They were firmly checked if working properly and compared to other GitHub algorithms which seem to do similar things. Thus, through manual analysis, we presume the quality to be at least somewhat good. Below in Figure 7, you can see an output of the code that was produced.

```
parallel_merge_sort: Run 1 - Execution Time = 0.049999 seconds
parallel_merge_sort: Run 2 - Execution Time = 0.050024 seconds
parallel_merge_sort: Run 3 - Execution Time = 0.064862 seconds
parallel_merge_sort: Run 4 - Execution Time = 0.065108 seconds
parallel_merge_sort: Run 5 - Execution Time = 0.069767 seconds
parallel_merge_sort: Average Execution Time = 0.059952 seconds
parallel_quick_sort: Run 1 - Execution Time = 0.030969 seconds
parallel_quick_sort: Run 2 - Execution Time = 0.038984 seconds
parallel_quick_sort: Run 3 - Execution Time = 0.040815 seconds
parallel_quick_sort: Run 4 - Execution Time = 0.032010 seconds
parallel_quick_sort: Run 5 - Execution Time = 0.016986 seconds
parallel_quick_sort: Average Execution Time = 0.031953 seconds
The winner is parallel_quick_sort with an average execution time of 0.031953 seconds
```

Figure 7: example output of experiment 2 with details of each run

2.2 Table Timing Results

Here we have the executing timing results when varying the big data size of the array. Each algorithm is listed as an attribute and time is measured in seconds. The results can be seen in Figure 8.

Array Size (n)	Parallel Merge Sort	Parallel Quick Sort
10000	3.162125	0.021972
100000	28.394007	0.276033
1000000	300.665882	2.872745

Figure 8: Execution timing results table for Experiment 2

2.3 Graph Timing Results

Here we have graphs and details of the performance of the execution times of all algorithms that were tested. Figure 9 represents how the algorithms perform in array sizes 10000 and 100000. We can see that the lines start to diverge at the very start, and at an array size of 100000. We can see clear performance differences. Figure 10 displays the final results where we can see a clear "winner". Consult figures below.

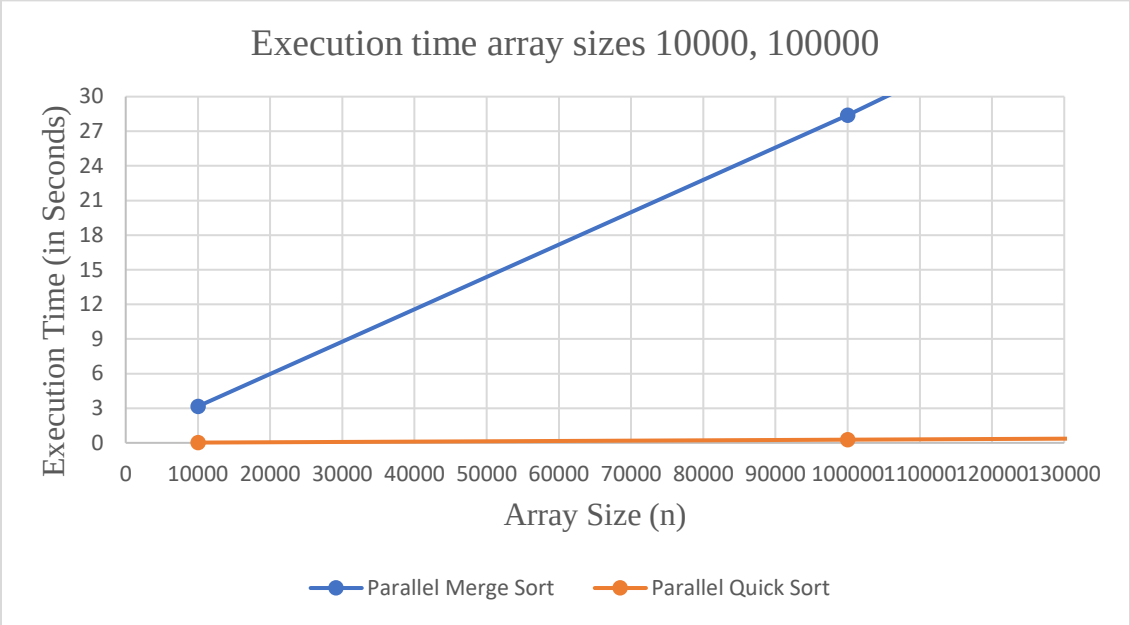

Figure 9: Execution time performance differences graph for array sizes of 10000 and 100000

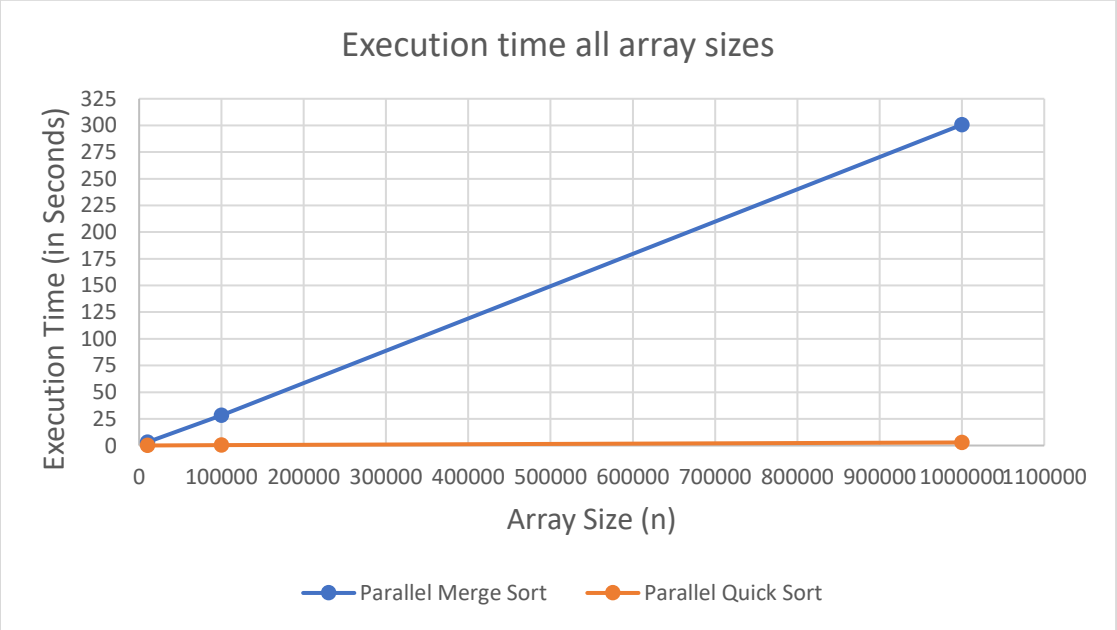

Figure 10: Execution time performance differences graph for all array sizes

2.4 Conclusions

After multiple runs, we can reject the null hypothesis and accept the alternative hypothesis. Similar to Experiment 1, the results of Experiment 2 demonstrate significant variation in the performance of the tested sorting algorithms when applied to randomly generated big data. Quick Parallel sort was always far ahead of Merge Parallel sort in all cases. The results of Experiment 2 provide compelling evidence that leveraging parallel sorting algorithms can significantly improve the efficiency of sorting large datasets. Parallel sorting algorithms, such as Parallel Merge Sort and Parallel Quick Sort were used although many others exist. The implications of Experiment 2 extend to real-world scenarios where efficient data processing is crucial. Applications dealing with vast volumes of data, such as big data analytics and scientific simulations, can benefit significantly from the adoption of parallel sorting algorithms to expedite data preparation and analysis.

Different parallel algorithms may excel in different scenarios, and understanding their strengths and weaknesses is vital for optimal performance. Future research could delve deeper into exploring additional parallel sorting algorithms and optimizing parallelization strategies, as that was not discussed in this paper. Additionally, investigating the scalability of parallel sorting algorithms on distributed computing clusters could provide insights into their performance.

3 Experiment 3

In Experiment Number 3, we aimed to assess and compare the performance of the chulevsort algorithm against three widely used sorting algorithms: Bubble Sort, Insertion Sort, and Selection Sort. It will be compared the same way Experiment 1 was conducted with array sizes of 10, 100, 1000, 10000 and randomly generated integer values ranging from -100 to -100.

We avoid comparing it with Quick Sort or Merge Sort due to them being industry standards with most optimal timings. Our primary goal was to determine if the execution time of chulevsort significantly differs from that of the traditional sorting algorithms. Thus, we formulate the following hypotheses:

*H*0 - *no significant difference in execution times between chulevsort and Bubble Sort, Insertion Sort, and Selection Sort*

Ha - chulevsort algorithm has a significantly different execution time compared to at least one of the other three sorting algorithms

To achieve this, we created a controlled experiment where we generated a random list of integers, conducted sorting operations on identical copies of this list using each of the four sorting algorithms, and measured the execution time. The results of this experiment will provide valuable insights into the relative efficiency of " chulevsort " compared to well-established sorting techniques, shedding light on its potential as an alternative sorting method.

3.1 Method Analysis

Chulevsort is a sorting algorithm that operates on a list of numerical values, such as integers. Its primary goal is to sort the input list into ascending order. We provide a detailed explanation of how the algorithm works:

The algorithm begins by calculating the mean (average) value of all the elements in the input list. This mean value acts as a pivot point for dividing the list into two sublists. The algorithm iterates through the input list and separates the elements into two sublists:

Lower Values: Elements that are smaller than the mean value are placed in the "Lower Values" sublist.

Higher Values: Elements that are greater than or equal to the mean value are placed in the "Higher Values" sublist.

To improve efficiency, chulevsort uses multithreading to sort the two sublists concurrently. This means it sorts the "Lower Values" and "Higher Values" in parallel. Two separate threads are created to perform the sorting. The first thread sorts the "Lower Values" sublist. It identifies the minimum values in this sublist iteratively, removes them, and appends them to a new list. The second thread sorts the "Higher Values" sublist. Similarly, it identifies the maximum values in this sublist iteratively, removes them, and appends them to another new list. The use of threads allows chulevsort to take advantage of multi-core processors and potentially speed up the sorting process. Once both threads have completed their sorting tasks, the algorithm reverses the order of the sorted "Higher Values" sublist. This is necessary because the "Higher Values" were identified and removed from the end of the sublist but need to be in ascending order. After reversing the "Higher Values" sublist, the algorithm concatenates the "Lower Values" and "Higher Values" sublists to create a single sorted list. The algorithm returns the sorted list, which now contains all the elements from the input list arranged in ascending order.

3.2 Code Analysis

The code is comprised of several functions that get called (Note: we make use of the threading library):

Firstly the “find_min” Function. This function is responsible for finding the minimum value in a given list and removing it. It uses the min() function to find the minimum value, which is provided by python.

Secondly, the “find_max” Function. This function is responsible for finding the maximum value in a given list and removing it. It uses the max() function to find the maximum value, also provided by python.

Next, the “split_list” Function. This function calculates the mean (average) value of the input list and splits it into two sublists: "Lower Values" and "Higher Values." It iterates through the input list, placing values smaller than the mean in the "Lower Values" sublist and values greater than or equal to the mean in the "Higher Values" sublist.

And finally, the “chulevsort” Function. It calls split_list to divide the input list into "Lower Values" and "Higher Values" sublists. It creates two separate threads to sort these sublists concurrently using find_min and find_max functions. It reverses the sorted "Higher Values" sublist. Finally, it concatenates the sorted "Lower Values" and "Higher Values" sublists to produce the sorted result. We can see the full code outline and all functions described below in Figure 11.

```python
import threading

def find_min(input_list, min_values):
    while input_list:
        min_value = min(input_list)
        min_values.append(min_value)
        input_list.remove(min_value)

def find_max(input_list, max_values):
    while input_list:
        max_value = max(input_list)
        max_values.append(max_value)
        input_list.remove(max_value)

def split_list(input_list):
    mean_value = sum(input_list) / len(input_list)
    lower_values = [x for x in input_list if x < mean_value]
    higher_values = [x for x in input_list if x >= mean_value]
    return lower_values, higher_values

def chulevsort(input_list):
    lower_values, higher_values = split_list(input_list)

    min_values = []
    max_values = []


    min_thread = threading.Thread(target=find_min, args=(lower_values, min_values))
    max_thread = threading.Thread(target=find_max, args=(higher_values, max_values))

    min_thread.start()
    max_thread.start()

    min_thread.join()
    max_thread.join()

    max_values.reverse()

    sorted_list = min_values + max_values

    return sorted_list
```

Figure 11: full code outline of the chulevsort algorithm

3.3 Timing Results

Here we have the executing timing results when varying the data size of the array. Each algorithm is listed as an attribute and time is measured in seconds. Note: the same timings were taken from Experiment 1 for the sorting algorithms, except for chulevsort. The results can be seen in Figure 12.

Array items (n)	Bubble Sort	Selection Sort	Insertion Sort	chulevsort
10	0.000009	0.000009	0.000007	0.000996
100	0.00059	0.00026	0.00028	0.001034
1000	0.060301	0.025112	0.029378	0.004477
10000	6.297506	2.592186	3.03399	0.33711

Figure 12: Execution timing results table for Experiment 3

Graphs are also provided for varying array sizes and all array sizes to see the performance. We can see in Figure 13 that once array size hits 100 data items, all the sorting algorithms start to diverge upwards, except for chulevsort which remains somewhat constant with timing. Figure 14 displays the final results where we can see a clear "winner". Consult figures below.

Figure 13: Execution time performance differences graph for array sizes 10 and 100

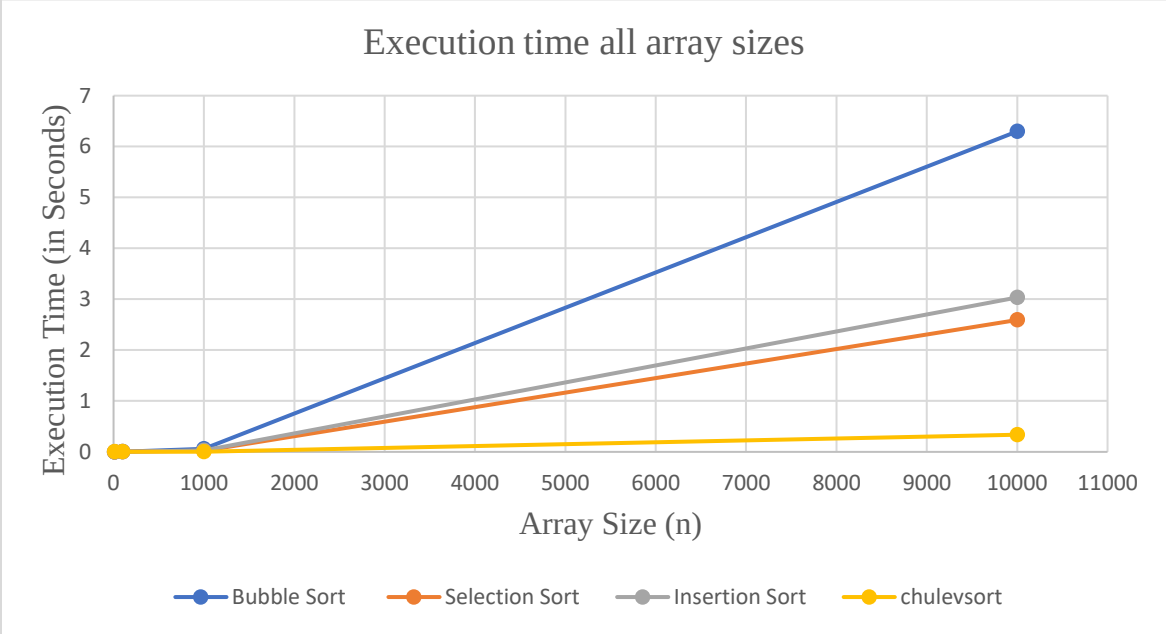

Figure 14: Execution time performance differences graph for all array sizes

3.4 Conclusions

After testing, we can firmly reject the null hypothesis and accept the alternative hypothesis. Chulevsort did outperform not one but all algorithms based on the array data of 1000 and 10000. However, it is worth mentioning it did underperform, being quite slower than the other algorithms in the array data of 10 and 100. This is likely due to the use of parallelism and threads, as well as using built in functions in these threads. To enhance performance, it leverages multithreading, but this becomes only apparent after a sufficient data size for sorting. Through logical analysis, it turned out to be an algorithm with a time complexity of $O(n)$. This sorting process is linear, $O(n)$, as each element is processed once. ChulevSort's parallel sorting approach and the subsequent merging of the sublists are all performed in linear time. Therefore, the overall time complexity remains $O(n)$. This analysis assumes that the multithreading and extremal value sorting operations maintain linear time complexity, and for practical purposes, this is a reasonable assumption. The algorithms' combination of divide-and-conquer, multithreading, and extremal value sorting makes it a potentially efficient sorting algorithm even when compared to staples in computer science. Future work may compare this algorithm to faster algorithms and other parallel algorithms on much bigger data sets. The code may also be adjusted to include other types of arrays and divide them based on some type of heuristic.

4 General Conclusions and Summary

In the conducted experiments, we compared the performance of various sorting algorithms, both sequential and parallel, to gain insights into their efficiency and suitability for different scenarios. Quick Sort and Merge Sort consistently demonstrated the best performance with significantly lower average execution times compared to other algorithms. This was also true for their parallel counterparts. However, Quick Sort (sequential and parallel) was always outperforming all the other sorting methods. We also included and defined our own algorithm chulevsort that used multithreading and actually outperformed Bubble Sort, Selection Sort and Insertion Sort on larger data sets. To summarize, the choice of a sorting algorithm should be based on the nature of the data, the size of the dataset, and the available hardware resources, although Quick Sort always seems to be the best option and the industry standard. Understanding the strengths and weaknesses of different algorithms is crucial for optimizing sorting tasks in various applications.

Acknowledgements to Professor Dr. Tamas Vinko for assistance with explanations and elaborations of the inner working of algorithms and time complexities.

References

Lakshmivarahan, S., Dhall, S. K., & Miller, L. L. (1984). Parallel sorting algorithms. In *Elsevier eBooks* (pp. 295–354). https://doi.org/10.1016/s0065-2458(08)60467-2

Python, R. (2023). Sorting algorithms in Python. *realpython.com*. https://realpython.com/sorting-algorithms-python/

Programming, P. (2023). How do you measure and compare the complexity of parallel sorting algorithms? *www.linkedin.com*. https://www.linkedin.com/advice/3/how-do-you-measure-compare-complexity-parallel#:~:text=It%20has%20a%20work%20complexity,parallel%2C%20and%20then%20concatenates%20them.