Performance evaluation of parallel sorting algorithm in the industry simulation environment

Jian Wang
Computer and Data Sciences
Case Western Reserve University
Cleveland, Ohio
jxw1640@case.edu

Frank Liu
Computer and Data Sciences
Case Western Reserve University
Cleveland, Ohio
wxl713@case.edu

Sixu Li
Computer and Data Sciences
Case Western Reserve University
Cleveland, Ohio
sxl2199@case.edu

Ruicheng Zhu
Computer and Data Sciences
Case Western Reserve University
Cleveland, Ohio
rxz540@case.edu

Chaoda Song
Computer and Data Sciences
Case Western Reserve University
Cleveland, Ohio
cxs965@case.edu

Xinxin Wang
Computer and Data Sciences
Case Western Reserve University
Cleveland, Ohio
xxw909@case.edu

***Abstract*— Sorting algorithms are ubiquitous and are a type of algorithm that large programs cannot avoid, especially in the field of parallel computing. In the industry, affected by program complexity and programmer level, different types of sorting algorithms will be widely used in various business systems. It is not difficult to find that many programmers are using algorithm code templates, and the performance of these templates directly determines the overall performance of the program. We have selected the following six most commonly used algorithms from a large set of sorting algorithms that support parallel operations: Enumeration Sort, Odd-Even Transposition Sort, Parallel Merge Sort, Hyper Quick Sort, Parallel Shell Sort, and Parallel Heap Sort. Horizontally, we will compare the time and space efficiency of the same algorithm between code templates of multiple groups of websites. Vertically, we will select the code template with the highest overall efficiency in each algorithm for comparison. We believe that our comparative experiments will truly reflect the many problems that junior and even intermediate programmers in the industry face when writing projects and provide constructive suggestions for this.**

***Keywords*— algorithms, templates, programmers, sorting, code, parallel, computing**

I. Introduction

Sorting algorithms [1], as the name suggests, are a series of algorithms used to sort a sequence. As the amount of data increases, sorting algorithms become more and more complex. When programmers have large amounts of data but also have requirements for program operations, they begin to consider sorting algorithms that support parallel operations. This type of sorting algorithm requires that the computer itself supports parallel operations at the hardware level and that the program has been processed accordingly during compilation. With the support of hardware, because some steps can be executed in parallel at the hardware level, the execution time of the algorithm is significantly shortened. With the advent of the era of cloud computing and big data, more and more data are being centrally calculated, which gives parallel sorting algorithms ample room for use. Through testing, we also realized the power of these algorithms when the amount of data significantly increases.

However, through testing, we also clearly realized that some algorithms are not suitable for parallel processing. The essence of parallelizing a program is to reasonably divide the steps of the program into some fragments. Some of these fragments can be run at the same time without interfering with each other. We let the parts that can be run at the same time be allocated to multiple threads. and have them executed simultaneously. However, no matter how finely we split the program, the data after the execution of these fragments will eventually need to be summarized. At this time, the algorithm problem of key nodes arises. In particular, for some algorithms, during serial execution, when the steps are executed halfway or even less, this part of the code can be terminated early due to some trigger conditions. However, for parallel execution, resource allocation often requires queuing, and it is not particularly easy to terminate all threads in real-time midway. On the contrary, it will take longer and occupy more computing resources.

II. Choice of Parallel Sorting Algorithm

Since there are many kinds of parallel sorting algorithms, we cannot test every one of them, which would be too time-consuming, but we have selected some of the most typical ones for comprehensive testing. Considering that most programmers in the industry are basic coders, we choose to follow their programming logic to select appropriate algorithms as samples. Considering that most programmers will turn to mainstream blogs and technology-related forums when they encounter any problems after graduating from schools or training institutions, we placed the sample collection locations in these places. Considering that ChatGPT [14] has come out now, many people will choose to go to ChatGPT for help and obtain codes. We also included the samples generated by ChatGPT in the scope of the investigation. Since different programming languages and operating environments will bring differences, we unified the code into C language and ran it on an HPE MicroServer Gen10 [16]. The server has an Intel Xeon E-2246G processor [17] and is equipped with 64GB of memory and a full SSD RAID 5 array. We compile and run all algorithm source codes on this server to ensure the uniformity of the experimental environment.

A. *Enumeration Sort*

Enumeration Sort [2] is a sorting algorithm that operates by counting the number of objects with distinct key values and arranging them in a particular order. The basic idea behind Enumeration Sort is to determine, for each element in the input list, the number of elements that are less than or equal to it. Based on these counts, the algorithm then places the elements in their correct sorted positions. Parallelizing Enumeration Sort involves distributing the counting and sorting tasks across multiple processors or threads to improve efficiency, especially for large datasets. However, it's important to note that not all sorting algorithms can be easily parallelized due to dependencies between elements. Enumeration Sort, being a counting-based algorithm, has some characteristics that make it amenable to parallelization. It's important to manage synchronization between processors or threads during the global count and cumulative sum steps.

Here we chose the code from Geeksforgeeks, a famous blog website for beginners. Also, we generated one by ChatGPT. As a control group, we also generated a common Enumeration Sort from ChatGPT for comparison.

B. *Parallel Shell Sort*

Parallel Shell Sort is a parallelized version of the Shell Sort algorithm [3], which is an extension of the insertion sort algorithm. Shell Sort improves on the efficiency of the insertion sort by allowing the comparison and exchange of elements that are far apart. It works by breaking the original list into smaller sublists, sorting each sublist independently, and then combining the sublists into a single sorted list.

The parallel version of Shell Sort takes advantage of multiple processors or threads to perform the sorting process more efficiently. In a parallel algorithm, the main idea is to divide the data into smaller chunks that can be sorted independently, and then merge or combine the sorted chunks.

Parallel Shell Sort provides an opportunity for improved performance, especially when dealing with large datasets. However, the efficiency gains depend on factors such as the number of processors, the size of the dataset, and the specific implementation of the algorithm.

C. *Parallel Merge Sort*

Parallel Merge Sort [5] is a parallelized version of the traditional Merge Sort algorithm [4], which is a comparison-based sorting algorithm. The goal of Merge Sort is to divide an array or list into two halves, recursively sort each half, and then merge the sorted halves to produce a fully sorted result.

In the context of parallel computing, the idea is to leverage multiple processors or cores to perform the sorting in parallel [6], thus potentially reducing the overall time complexity and improving the efficiency of the sorting process.

By parallelizing the sorting and merging steps, Parallel Merge Sort can potentially achieve better performance compared to the sequential version, especially when dealing with large datasets and a parallel computing environment.

D. *Parallel Heap Sort*

Heap Sort [7] is typically considered an in-place, comparison-based sorting algorithm with a time complexity of O(n log n). It is based on the binary heap data structure, where the elements of the array are treated as a binary tree.

Parallelizing Heap Sort involves utilizing multiple processors or cores to perform the sorting in parallel, potentially reducing the overall time complexity and improving efficiency.

It's important to note that designing an efficient parallel algorithm for Heap Sort involves addressing challenges related to load balancing, synchronization, and communication among parallel processes. Additionally, the degree of parallelism and the performance of the parallel algorithm can depend on factors such as the underlying hardware architecture and the size and nature of the data being sorted.

E. *Hyper Quick Sort*

Parallel quicksort [9] is a parallelized version of the traditional quicksort algorithm. Quicksort [8] is a comparison sort and is known for its efficiency and average-case performance. In parallel quicksort, the idea is to exploit parallel processing capabilities to speed up the sorting process [11].

The basic idea of quicksort involves selecting a pivot element from the array and partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot. The process is then applied recursively to the sub-arrays.

In parallel quicksort, the partitioning and sorting steps are performed concurrently on different processors or threads, allowing for better utilization of resources and potentially reducing the overall time complexity of the algorithm.

There are various ways to parallelize quicksort, and the specific implementation may depend on the parallel computing architecture and programming model being used. Some common parallelization strategies include parallelizing the partitioning step, parallelizing the recursive calls to sort sub-arrays, or a combination of both.

It's important to note that parallel algorithms often introduce additional complexities, such as the need for synchronization and load balancing [10], to ensure efficient parallel execution.

F. *Odd-Even Transposition Sort*

Odd-Even Transposition Sort [13] is a parallel sorting algorithm that extends the concept of the odd-even sort [12] algorithm to parallel computing. It is designed to be implemented on parallel architectures, where multiple processors or parallel units can work simultaneously.

Odd-Even Transposition Sort is a simple sorting algorithm that can be easily parallelized, making it suitable for parallel computing architectures. However, it's important to note that the efficiency of this algorithm can be affected by factors such as load balancing (ensuring that processors have roughly equal amounts of work) and communication overhead between processors.

III. Comparison

For comparison, I generated four sets of data. The size of the generated random number is between 0 and 1000000. The data volumes of these three sets of data are 10,000, 100,000, and 1,000,000 respectively. We originally prepared a set of data with a data volume of 10,000,000, but after testing, we found that in the third set of data, the running speed of the program had slowed down significantly, and it took a long time to complete, so later We eliminated this set of data. Subsequently, we brought these three sets of data into all programs for testing. These four sets of data cover data volumes from small to large and can fully reflect the capabilities of the algorithm. At the same time, the time consumed in calculating these four sets of data is still within the acceptable range, and it can also avoid the problem of being unable to draw on the final chart due to too time-consuming calculations.

The first thing we conducted was a horizontal test of each group of programs, that is, a mutual comparison between three samples of a single algorithm because this helps us select the best one and use this best sample for subsequent longitudinal comparisons.

Before testing, we limited the number of threads our algorithm could use to 10 because leaving 2 threads redundant would help prevent the operating system and other programs running in the background from interfering with our test results. At the same time, limiting the number of threads in this way can also cause some applications to be queued to obtain computing resources when the amount of data is large. This can also reflect the time-consuming growth that occurs when resources are bottlenecked.

Next, we compile the programs one by one and bring them into the programs for testing and use the running time. We summarized the running time of each program and obtained the following tables. The programs we use are all described in C language, because this can avoid the difference in runtime time and decoding time caused by different program development languages. At the same time, since the C language will be converted into a binary file after being compiled. The time loss caused by it during the decoding process can be better eliminated.

A. *Horizontal Contrast*

1) *Parallel Enumeration Sort*

The following are the time-consuming statistics obtained after bringing in three sets of data. The tool used is the "time" command that comes with Linux because this command is very intuitive; it is also directly provided by the system, and the accuracy meets our requirements.

TABLE 1. "TIME" RESULT OF ENUMERATION SORT SAMPLE

Sample			
	Data 10000	***Data 100000***	***Data 1000000***
real	0m10.286s	17m6.019s	(Crushed)
user	0m10.270s	17m5.903s	(Crushed)
sys	0m0.000s	0m0.020s	(Crushed)

TABLE 2. "TIME" RESULT OF ENUMERATION SORT FROM CHATGPT CODE

ChatGPT			
	Data 10000	***Data 100000***	***Data 1000000***
real	0m0.135s	0m12.209s	22m5.758s
user	0m1.567s	2m20.558s	254m17.978s
sys	0m0.000s	0m0.052s	0m1.008s

TABLE 3. "TIME" RESULT OF SERIAL ENUMERATION SORT

Serial			
	Data 10000	***Data 100000***	***Data 1000000***
real	0m0.656s	1m6.374s	138m33.311s
user	0m0.656s	1m6.314s	138m8.995s
sys	0m0.000s	0m0.000s	0m0.058s

We focus on the "real" times and compare them.

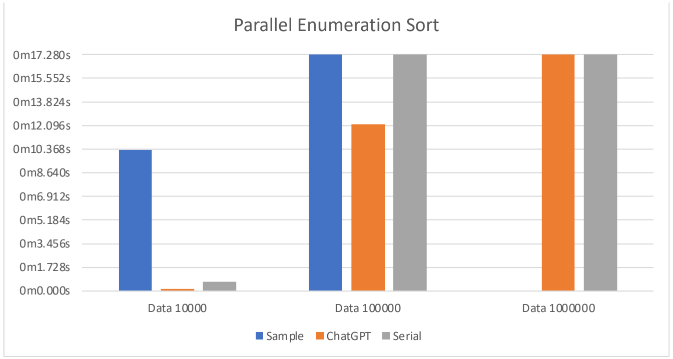

Fig. 1. Comparison between Enumeration Sort Algorithm Samples

We can notice that there is an issue with Geeksforgeeks' code that causes it to run slowly, even slower than the code running serially. Running the Geeksforgeeks code alone can reveal that the program code has serious problems with online multi-thread synchronization, which causes the program to wait for a long time during synchronization. But comparing the code of ChatGPT and the code running serially, the speedup brought by OpenMP [15] multi-threading is still obvious. Here we choose the code of ChatGPT for subsequent vertical comparison.

2) *Parallel Shell Sort*

The following are the time measurement results obtained after bringing in three sets of data. We still use exactly the same testing method as the previous algorithm.

TABLE 4. "TIME" RESULT OF SHELL SORT SAMPLE

Sample			
	Data 10000	***Data 100000***	***Data 1000000***
real	0m0.581s	0m1.609s	0m20.238s

user	0m0.112s	0m1.527s	1m47.003s
sys	0m0.013s	0m0.016s	0m0.036s

TABLE 5. "TIME" RESULT OF SHELL SORT FROM CHATGPT CODE

ChatGPT			
	Data 10000	***Data 100000***	***Data 1000000***
real	0m0.562s	0m9.755s	13m36.280s
user	0m0.975s	0m53.556s	80m54.553s
sys	0m0.012s	0m0.015s	0m0.026s

TABLE 6. "TIME" RESULT OF SERIAL SHELL SORT

Serial			
	Data 10000	***Data 100000***	***Data 1000000***
real	0m0.656s	1m6.374s	138m33.311s
user	0m0.656s	1m6.314s	138m8.995s
sys	0m0.000s	0m0.000s	0m0.058s

Also, focus on the "real" times that are spent and compare them.

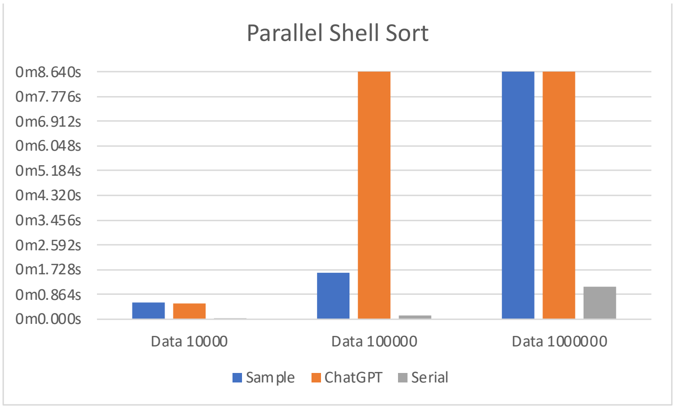

Fig. 2. Comparison between Shell Sort Algorithm Samples

It is easy to tell that for this algorithm, serial execution is faster than parallel execution. At the same time, the code templates provided by third-party websites run faster than the code generated by ChatGPT. Here we select code samples from third-party websites for subsequent longitudinal comparison.

3) Parallel Merge Sort

The following are the time statistical results obtained after bringing in three sets of data. The calculation method remains the same as above.

TABLE 7. "TIME" RESULT OF MERGE SORT

Sample			
	Data 10000	***Data 100000***	***Data 1000000***
real	0m0.052s	0m0.125s	0m0.844s
user	0m0.066s	0m0.205s	0m0.766s
sys	0m0.012s	0m0.029s	0m0.167s

TABLE 8. "TIME" RESULT OF MERGE SORT FROM CHATGPT CODE

ChatGPT			
	Data 10000	***Data 100000***	***Data 1000000***
real	0m0.007s	0m0.056s	0m0.416s
user	0m0.040s	0m0.249s	0m0.859s
sys	0m0.000s	0m0.000s	0m0.004s

TABLE 9. "TIME" RESULT OF SERIAL MERGE SORT

Serial			
	Data 10000	***Data 100000***	***Data 1000000***
real	0m0.007s	0m0.064s	0m0.663s
user	0m0.003s	0m0.063s	0m0.659s
sys	0m0.004s	0m0.000s	0m0.004s

Also, focus on the "real" times that are spent and compare them.

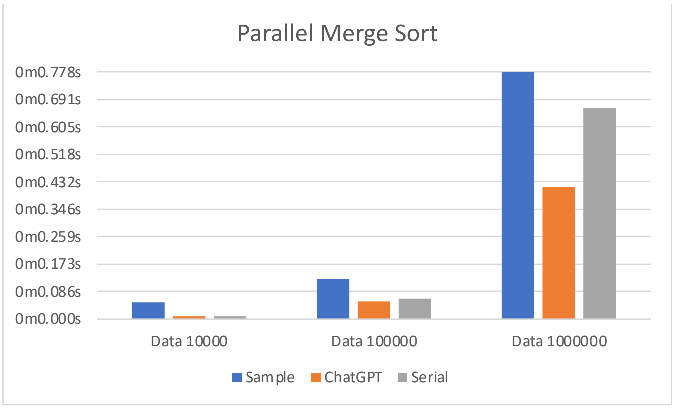

Fig. 3. Comparison between Merge Sort Algorithm Samples

It can be seen that among the samples of this algorithm, the version generated by ChatGPT is more dominant and shows an advantage in running speed. This advantage becomes more significant as the amount of data increases. Here we choose the code generated by ChatGPT for subsequent longitudinal comparison.

4) Parallel Heap Sort

The same testing method is applied to this algorithm. We got the following results.

TABLE 10. "TIME" RESULT OF HEAP SORT SAMPLE

Sample			
	Data 10000	***Data 100000***	***Data 1000000***
real	0m0.004s	0m0.028s	0m0.262s
user	0m0.016s	0m0.116s	0m0.681s
sys	0m0.000s	0m0.004s	0m0.008s

TABLE 11. "TIME" RESULT OF HEAP SORT FROM CHATGPT CODE

	ChatGPT		
	Data 10000	***Data 100000***	***Data 1000000***
real	0m0.007s	0m0.056s	0m0.416s
user	0m0.040s	0m0.249s	0m0.859s
sys	0m0.000s	0m0.000s	0m0.004s

TABLE 12. "TIME" RESULT OF SERIAL HEAP SORT

	Serial		
	Data 10000	***Data 100000***	***Data 1000000***
real	0m0.007s	0m0.064s	0m0.663s
user	0m0.003s	0m0.063s	0m0.659s
sys	0m0.004s	0m0.000s	0m0.004s

Also, focus on the "real" times that are spent and compare them.

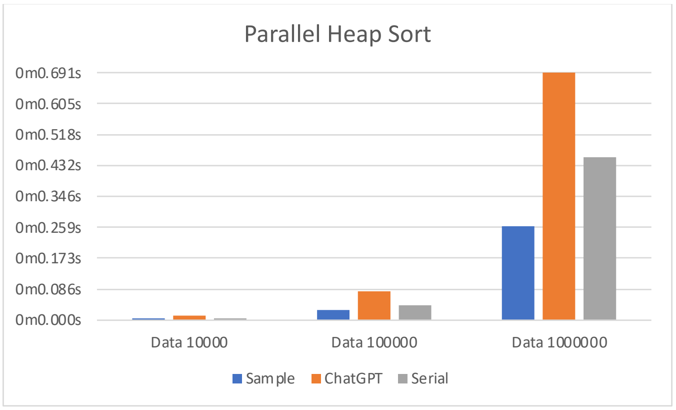

Fig. 4. Comparison between Heap Sort Algorithm Samples

It is easy to tell that in this algorithm, code samples from third-party websites have an advantage. Here we select code samples from third-party websites for subsequent longitudinal comparison.

5) Hyper Quick Sort

Also, the same testing method is applied to this algorithm. We got the following results.

TABLE 13. "TIME" RESULT OF QUICK SORT SAMPLE

	Sample		
	Data 10000	***Data 100000***	***Data 1000000***
real	0m0.011s	0m0.089s	0m0.970s
user	0m0.058s	0m0.284s	0m1.043s
sys	0m0.005s	0m0.051s	0m0.218s

TABLE 14. "TIME" RESULT OF QUICK SORT FROM CHATGPT CODE

	ChatGPT		
	Data 10000	***Data 100000***	***Data 1000000***
real	0m0.012s	0m0.896s	1m31.442s
user	0m0.054s	0m2.250s	5m57.433s
sys	0m0.035s	0m3.576s	4m3.927s

TABLE 15. "TIME" RESULT OF SERIAL QUICK SORT

	Serial		
	Data 10000	***Data 100000***	***Data 1000000***
real	0m0.004s	0m0.034s	0m0.376s
user	0m0.004s	0m0.030s	0m0.355s
sys	0m0.000s	0m0.004s	0m0.008s

Also, focus on the "real" times that are spent and compare them.

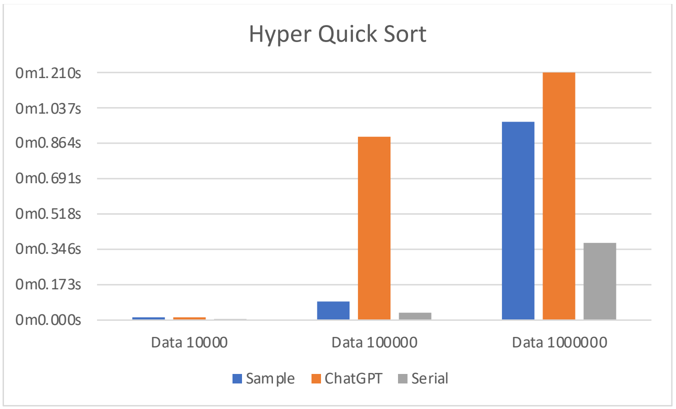

Fig. 5. Comparison between Quick Sort Algorithm Samples

It is easy to tell that in this algorithm, code samples from third-party websites have an advantage. Here we select code samples from third-party websites for subsequent longitudinal comparison.

6) Odd-Even Transposition Sort

Also same as the algorithms mentioned before, the same testing method is applied to this algorithm. We got the following results.

TABLE 16. "TIME" RESULT OF ODD-EVEN TRANSPOSITION SORT SAMPLE

	Sample		
	Data 10000	***Data 100000***	***Data 1000000***
real	0m0.092s	0m8.719s	15m0.513s
user	0m0.878s	1m27.038s	150m0.508s
sys	0m0.004s	0m0.020s	0m0.476s

TABLE 17. "TIME" RESULT OF ODD-EVEN TRANSPOSITION SORT FROM CHATGPT CODE

	ChatGPT		
	Data 10000	***Data 100000***	***Data 1000000***
real	0m0.062s	(Crushed)	(Crushed)

user	0m0.210s	(Crushed)	(Crushed)
sys	0m0.012s	(Crushed)	(Crushed)

TABLE 18. "TIME" RESULT OF SERIAL ODD-EVEN SORT

Serial			
	Data 10000	***Data 100000***	***Data 1000000***
real	0m0.152s	0m16.391s	27m16.153s
user	0m0.148s	0m16.386s	27m16.008s
sys	0m0.004s	0m0.004s	0m0.028s

Also, focus on the "real" times that are spent and compare them.

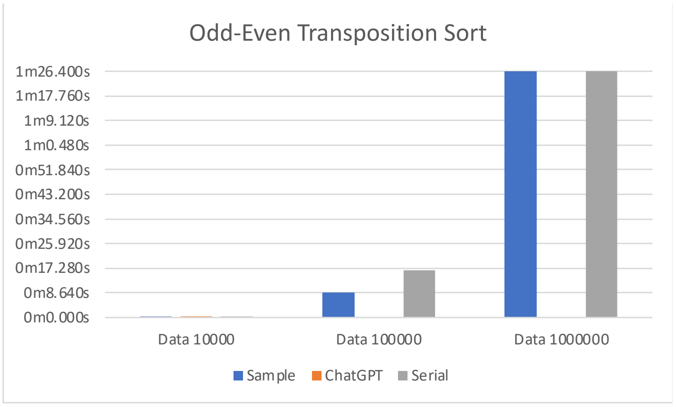

Fig. 6. Comparison between Odd-Even Transposition Sort Algorithm Samples

It is easy to tell whether it is the code generated by ChatGPT or the serial code, when the amount of data increases, the program crashes. Only code samples from third-party websites passed the test and got timing data. Here we select code samples from third-party websites for subsequent longitudinal comparison.

B. *Vertical Contrast*

Now, let's put the above data together and chart them in a way that is easier to observe.

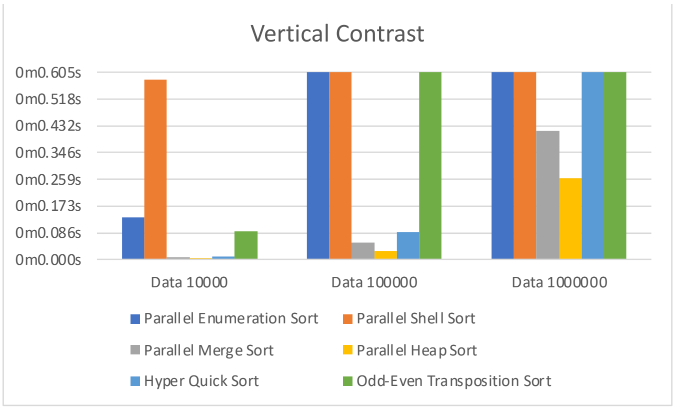

Fig. 7. Vertical Contract between algorithms

It can be clearly observed that with the support of parallel computing OpenMP, the fastest algorithm is Parallel Heap Sort, followed by Parallel Merge Sort, and then Hyper Quick Sort. In comparison, Parallel Shell Sort ranks at the bottom in terms of performance. Parallel Enumeration Sort and Odd-Even Transposition Sort also have poor performance compared to other algorithms.

IV. CONCLUSION

It is easy to tell that although the six algorithms we selected are widely used by programmers in daily programming, they show very significant performance differences when facing increasingly larger amounts of data. Although everyone has studied data structures and knows how to calculate time complexity and space complexity, in actual industrial scenarios, very few people really pay such careful attention to algorithm selection to achieve time and space complexity when designing software architecture. degree assessment. The greatest significance of this survey is to provide a visual reference for programmers in the industry. The results of our algorithm evaluation are integrated into charts, which are very intuitive. Especially for people who are not very advanced but need to engage in related programming work, this kind of information is very useful for them when citing code templates online. After all, most programmers are not highly skilled. Not every programmer gets an A in data structures and algorithms before graduation. Quoting code templates online is nothing to be ashamed of because it can significantly reduce development time. Just like we use Algorithm header files when writing C++ programs, these online templates also play the same role and are an extension of the header files. As long as programmers abide by the open-source agreement, respect copyright, mark the source, and use it as required, this is an initiative to promote the spread of excellent code and encourage algorithm engineers to contribute more templates and libraries.

We also hope that more people will enrich this type of testing in the future, be able to test more algorithms in more environments, and provide visual evaluation results, which will also help junior and even intermediate programmers in the industry. The growth also helps enterprise-level software to select the most appropriate algorithm during research and development. Because sorting algorithms are just one of the commonly used categories of algorithms. There are a large number of open source templates for search algorithms, graph algorithms, and various encryption algorithms. Some of them also have large amounts of code, many contributors, and many versions, and have been directly applied in a large number of projects. These algorithms also require humans to make similar assessments. At the same time, we also realized that ChatGPT cannot generate excellent code. The code quality is very unstable and sometimes even slower than the serial algorithm. Therefore, this test also intuitively demonstrated to programmers through data that ChatGPT cannot replace the work of a programmer. A programmer with the ability to think can write code that runs much more efficiently than ChatGPT.

ACKNOWLEDGMENT

This paper is a group work in the HPC course. Thanks to the entire team members for their cooperation and dedication during this semester. At the same time, most members of this group are also scientific research members of the OnePolicy Research Foundation. These members are also doing relevant algorithm analysis research within the foundation. Thanks to them for opening up and contributing their own foundation and some codes to this research work. Finally, we also sincerely thank the HPC course teaching team for all their efforts in the course this semester.

References

[1] GeeksforGeeks. 2017. "Sorting Algorithms - GeeksforGeeks." GeeksforGeeks. 2017. https://www.geeksforgeeks.org/sorting-algorithms/.

[2] "Counting Sort." 2013. GeeksforGeeks. March 18, 2013. https://www.geeksforgeeks.org/counting-sort/.

[3] GeeksforGeeks. 2014. "ShellSort." GeeksforGeeks. June 16, 2014. https://www.geeksforgeeks.org/shellsort/.

[4] GeeksforGeeks. 2018. "Merge Sort - GeeksforGeeks." GeeksforGeeks. October 31, 2018. https://www.geeksforgeeks.org/merge-sort/.

[5] "Merge Sort Using Multi-Threading." 2017. GeeksforGeeks. December 28, 2017. https://www.geeksforgeeks.org/merge-sort-using-multi-threading/.

[6] "Concurrent Merge Sort in Shared Memory." 2016. GeeksforGeeks. July 25, 2016. https://www.geeksforgeeks.org/concurrent-merge-sort-in-shared-memory/.

[7] "Heap Sort - Data Structures and Algorithms Tutorials." 2013. GeeksforGeeks. March 16, 2013. https://www.geeksforgeeks.org/heap-sort.

[8] GeeksforGeeks. 2014. "QuickSort - GeeksforGeeks." GeeksforGeeks. GeeksforGeeks. January 7, 2014. https://www.geeksforgeeks.org/quick-sort/.

[9] "Advanced Quick Sort (Hybrid Algorithm)." 2020. GeeksforGeeks. April 11, 2020. https://www.geeksforgeeks.org/advanced-quick-sort-hybrid-algorithm/.

[10] "Implementation of Quick Sort Using MPI, OMP and Posix Thread." 2021. GeeksforGeeks. July 1, 2021. https://www.geeksforgeeks.org/implementation-of-quick-sort-using-mpi-omp-and-posix-thread/.

[11] "Quick Sort Using Multi-Threading." 2020. GeeksforGeeks. July 8, 2020. https://www.geeksforgeeks.org/quick-sort-using-multi-threading/.

[12] "Odd-Even Sort / Brick Sort." 2016. GeeksforGeeks. June 28, 2016. https://www.geeksforgeeks.org/odd-even-sort-brick-sort/.

[13] "Odd Even Transposition Sort / Brick Sort Using Pthreads." 2019. GeeksforGeeks. May 6, 2019. https://www.geeksforgeeks.org/odd-even-transposition-sort-brick-sort-using-pthreads/.

[14] OpenAI. 2023. "ChatGPT." Chat.openai.com. OpenAI. 2023. https://chat.openai.com/.

[15] tim.lewis. n.d. "Home." OpenMP. https://www.openmp.org/.

[16] PSNow. "HPE ProLiant MicroServer Gen10 Plus," n.d. https://www.hpe.com/psnow/doc/a00073554enw.

[17] "Intel® Xeon® E-2246G Processor (12M Cache, 3.60 GHz) - Product Specifications." n.d. Intel. Accessed December 18, 2023. https://www.intel.com/content/www/us/en/products/sku/191043/intel-xeon-e2246g-processor-12m-cache-3-60-ghz/specifications.html.