Analyzing Efficiency of Sorting Algorithms in Parallel Utilizing CUDA and MPI

Ilias Soumayah
soumai@rpi.edu
Rensselaer Polytechnic Institute
Troy, NY, USA

Josh Mozes
mozesj@rpi.edu
Rensselaer Polytechnic Institute
Troy, NY, USA

Antonio Martinez
martia20@rpi.edu
Rensselaer Polytechnic Institute
Troy, NY, USA

Reza Malik
malikr3@rpi.edu
Rensselaer Polytechnic Institute
Troy, NY, USA

ABSTRACT

In this paper, the performance dynamics of parallelized sorting algorithms will be explored. The primary focus will be on bubble sort and merge sort, but related works that examined other algorithms will also be discussed. While utilizing CUDA and MPI, we evaluate how these algorithms scale with different data sizes. Our results indicate that the parallel version of bubble sort performs poorly with smaller datasets but shows significant improvement with larger data sets. This suggests the existence of a threshold beyond which parallelization becomes more beneficial than the serial implementation. Merge sort, which is already efficient in serial form, sees a decrease in performance when parallelized with the datasets that we tested. This is primarily due to the overhead associated with additional resources. This study highlights the tradeoffs involved in parallel algorithm design and emphasizes the importance of selecting the best algorithm for a given situation and dataset. By looking at both strong and weak scaling properties and other metrics, this paper allows for a deeper understanding of how to leverage parallel computing to optimize sorting tasks for high-performance computing.

ACM Reference Format:
Ilias Soumayah, Josh Mozes, Antonio Martinez, and Reza Malik. 2024. Analyzing Efficiency of Sorting Algorithms in Parallel Utilizing CUDA and MPI. In *Proceedings of (Parallel Computing).* ACM, New York, NY, USA, 8 pages.

1 CONTRIBUTION SUMMARY

Ilias Soumayah worked on implementing merge sort in both parallel and serial. Antonio Martinez worked on implementing bubble sort in both parallel and serial. Josh Mozes conducted data gathering, analysis of results, and other areas of the report. Reza Malik wrote the report, found and analyzed references, and assisted with data analysis.

Parallel Computing, April 2024, Troy, NY

2 INTRODUCTION

Sorting algorithms are fundamental to computer science. The advent of parallel computing, especially through architectures like CUDA and MPI has allowed for major improvements to algorithmic efficiency and computation speed. This paper will delve into the efficiency of two sorting algorithms when parallelized using MPI and CUDA. The goal is to uncover insights into the performance and scalability of these sorting methods. In the time of big data and high-performance computing (HPC) the ability to quickly and correctly sort vast datasets is far more critical than it may seem. Serial sorting algorithms often fail to meet the demands of modern applications that are data intensive. In these instances, the data can exceed the processing capabilities of a single-threaded environment. However, parallel computing allows for using GPUs and distributed systems. This helps with the previously mentioned bottleneck. By spreading the sorting workload across many processing units, we anticipate reductions in computational time. This leads to improved throughput and overall efficiency. The intersection of MPI and CUDA is critical to this project. MPI provides a strong framework for process communication. CUDA enables direct programming of GPUs. Together these two tools provide a powerful combination for optimizing sorting algorithms in ways that traditional computing does not allow. This paper will explore the use of MPI and CUDA in this manner and confirm the belief that these implementations of sorting algorithms provide superior performance when dealing with large data sets as compared to serial algorithms. This is interesting to explore because it addresses the need for scalable sorting solutions. In today's world, massive amounts of data are handled and processed by people in various fields ranging from biology to financial analysis to search engine optimization. Additionally, this paper offers insights into the advantages and disadvantages of parallel algorithm design. We expect to find that parallel sorting algorithms significantly outperform their serial equivalents. Of course, we expect this performance to vary depending on which algorithm is used, size and type of dataset, and computing architecture used. There have also been ongoing advancements in computing hardware and software. As GPUs become more powerful and distributed computing options become more viable, knowing how to use these technologies to perform tasks like sorting will be critical. All in all, the efficiency of sorting algorithms in parallel is an important area of study with major implications. By doing a comprehensive analysis of the algorithms

mentioned in this paper, we will determine the performance and scalability of these algorithms.

3 METHODS

3.1 Merge Sort Implementation

Merge sort is an algorithm that has the following phases. Firstly, an unsorted array is split using recursion. Then, when it is of size 2, the 2 elements are compared and swapped as needed. Finally, sublists are merged together until the list is of the original size again. Our implementation of this began by generating a random list of the size specified - this serves as input. Then, an MPI environment is initialized to facilitate parallel processing. Sublists are derived from the main list to each MPI process (with each process handling a certain subsection of the list). MPI I/O is used to compute the starting offset for each sublist based on process rank and list length. The data is then loaded from the file into an integer array which represents the sublist for each respective MPI process. Then, a CUDA Kernel is launched to sort the sublist to each MPI process. The mergeCuda() function contains a while loop which progressively sorts and merges small segments of the array. It begins by sorting with pairs of elements, then merges these pairs into sorted segments of size 4, and continues doubling segment size until the whole list is sorted. The kernel dynamically adapts the number of threads to make sure that resource utilization and overhead is optimized. Process role assignment is done by dividing the MPI processes into receivers and senders. Senders transmit their sorted sublists to the associated receivers. After receiving a sublist, a receiver process merges that with its own sorted sublist. Then these processes are gradually phased out after sending data. When the number of processes is odd, excess data is handled by sending it to the process that has rank 0. This then merges with the other data. The process of sending and merging is continued in a loop until only two sublists are left. The final two sublists are then merged which produces the final product - a sorted list that is the same size as the original input. This fully sorted list is then outputted.

3.2 Bubble Sort Implementation

This is a parallelized version of bubble sort using MPI for distributed processing and CUDA. The algorithm serves to sort a large array that is initially spread across MPI processes. First, an array generator function is used to create a random array of size N. MPI I/O is used to read a segment of the file that is assigned to it. The raw data is converted into an integer array in order to make sorting more simple. The CUDA implementation is done by each MPI rank initializing a CUDA kernel to sort the subsection of the array that is loaded. For efficiency, the kernel is configured to launch n/2 threads in a single block. The bubble sort algorithm is executed on the GPU, where each thread has a responsibility for some portion of the sorting process. Threads are synchronized to ensure that no data is lost. Once this sorting is completed, a sorted sub-array is sent back to the host process. MPI is used to gather the individual sorted sub-arrays from the different processes. An N-merge sort technique is then used to merge these into a single array. The algorithm uses n indices, each of which point to the start of one of the sub-arrays. It finds the smallest element among current indices (iteratively), places that element into the final array, and increments the index that contained this element. This process repeats until all indices have been visited and the final array is completely sorted. The sorted array is then outputted. The time taken for the entire sorting and merging process is recorded and outputted for analysis.

4 REVIEW OF RELATED WORKS

In this section, ten pertinent research papers will be discussed. These studies also explore the utilization of CUDA and MPI for enhancing the efficiency of sorting algorithms in parallel. The analysis in this section will provide crucial context and information regarding the broader advancements in parallel computing. Each work referenced in this section was carefully chosen and was reviewed for sources and relevancy. Through evaluating these studies, this section seeks to highlight key methodologies, performance benchmarks, and scalability of similar algorithms. This analysis serves to recognize the critical past work of previous researchers and further define the scope and direction of this study.
The studies are cited fully in the references section.

4.1 Study 1: A Study of Parallel Sorting Algorithms Using CUDA and OpenMP

This is a study with a similar goal to ours, but it uses CUDA and OpenMP - while we are using CUDA and MPI. The key difference between MPI and OpenMP is that in MPI, processes have their own memory space and execute independently from other processes. Whereas in OpenMP, threads share the same set of resources and access shared memory.
This study describes two main types of parallelism. Firstly, there is Task-level Parallelism which involves distributing execution processes across multiple computational nodes (ex. CPUs or CPU cores). This is done by using threads that share workload for executing functions. However, communication overhead can be costly in this case. Secondly, there is Data-level parallelism which focuses on executing multiple data items simultaneously by distributing data across several threads or processes. These threads share data but not functions. This form of parallelism is more precise and is often used in operating systems where multiple threads are able to run concurrently when enough CPUs or cores are available. This study critiques some other existing literature for conducting poor comparisons and failing to consider some important metrics. Initial chapters of the study provide background on the two languages used. The study then goes into a massive review of existing literature to find gaps in testing and data. Chapter 6 introduces suitable algorithms for comparison and chapter 7 discusses the results. It is noted that CUDA offers higher throughput while OpenMP algorithms tend to run faster due to CPU's higher clock frequencies. The study also mentions CUDA's complexity in coding and debugging. The main conclusion of the study is that data level parallelism is significantly more advantageous when compared to task-level parallelism. This is seen in terms of efficiency and the potential for future improvements. The study discusses the future and calls for advancements in GPU hardware to enhance computational capabilities, improved testing, and better analysis of data.

4.2 Study 2: Parallel Computing for Sorting Algorithms

This study correctly points out that over the past 25 to 40 years, major advancements have been made in the field of microprocessors, memory, and networks. Parallel computing is now regarded as a viable high-end computing solution. It has applications across many fields such as environmental science, engineering, biology, medicine, and more. The study explores different methods of parallelizing quicksort - including a shared address pace and hypercube-based model which optimizes the sorting process by dividing tasks to separate processors. The paper ultimately concludes that while traditional serial algorithms for sorting and finding medians are effective, parallel sorting algorithms offer superior performance when implemented properly with modern parallel computing techniques and standards.

4.3 Study 3: Fast Parallel Sorting Algorithms on GPUs

This study mentions that Multi-core CPUs are now very common due to advances in microprocessor architecture which has led to improved high performance computing. This study is different from ours in that the main focus of this study was looking at the differences in performance depending on GPUs. Additionally, these researchers looked at many more algorithms over a longer period of time. GPU manages threads at a hardware level which minimizes times for context switches. Tasks are executed in parallel using an addressing scheme to manage data across dimensions and work groups. The conclusion of this study offers several key takeaways. Firstly, the study proves that the performance of s'orting algorithms is greatly influenced by the inherent characteristics and architecture of the hardware used to execute it. The study also showed that certain sorting algorithms are better suited for certain types of hardware. For example, bitonic sort, which is highly parallelizable, achieves a major speed increase on a powerful GPU. On the other hand, rank sort shows better performance on CPUs because it depends on data arrangement which is handled better by the architecture of a CPU. Among the tested algorithms, butterfly sort demonstrated the best performance overall. The document suggests that future work will look at refining the algorithm and further exploring various other sorting algorithms.

4.4 Study 4: The Comparison of Parallel Sorting Algorithms Implemented on Different Hardware Platforms

The study highlights the fact that sorting is a fundamental problem that is crucial for efficient data access, especially when it comes to data-driven algorithms. Like some of the other studies mentioned above, this study emphasizes the fact that certain sorting algorithms perform better with certain hardware architectures. Examples of sorting algorithms that can be implemented in parallel that were looked at in this study are merge sort, quick sort, odd-even sort, and some hybrid sorting algorithms which were designed for multi-core systems. This study utilized CUDA and OpenMP with CUDA being used for programming NVIDIA GPUs and OpenMP allowing for multi-threaded programming with shared memory. The study concluded that hybrid algorithms are generally faster than algorithms that run solely on CPUs, but it is dependent on how optimized the algorithms are the quality of the hardware. The authors also found that multi-core algorithms outperform single sore and GPU-based algorithms when it comes to scalability and efficiency.

4.5 Study 5: Fast in-place, Comparison-Based Sorting With CUDA: A Study With Bitonic Sort

This study mentions that modern GPUs are very programmable and work well with frameworks like CUDA. This makes them suitable for both general computing and intensive tasks like sorting. The introduction of GPUs as computing platforms has led to development of sorting algorithms that are specifically designed for GPU architectures. This study found that the optimized Bitonic sort implementation that they created outperforms other GPU sorting algorithms. Despite the actual time complexity of Bitonic sort, it remains competitive with other GPU sorting algorithms due to its comparison based nature and strategic use of memory. The study concludes that the optimized Bitonic sort is a major advancement. It is effective and maintains this through various key sizes. It performs especially well with very large key sizes where it outperforms many alternatives.

4.6 Study 6: Performance Evaluation of Merge and Quick Sort using GPU Computing with CUDA

This article mentions two main methods of sorting: sequential and parallel. The focus is on parallel sorting due to its relevance for handling massive data sets and applications to various fields in science. This study agrees with many of the other studies discussed that NVIDIA's CUDA is the most efficient way to parallelize sorting algorithms like merge sort and quick sort. The experiments in this paper were done using an NVIDIA GeForce GTX GPU and showed that parallel sorting algorithms outperform sequential equivalents by a large amount. The study concludes that GPU based merge sort and quick sort are highly efficient, enhance speed, and also optimize space usage compared to CPU based sorting. The study also states that the CUDA framework allows for us to leverage GPUs for this purpose.

4.7 Study 7: An Evaluation of Fast Segmented Sorting Implementations on GPUs

This study compares the fastest segmented sorting implementations across seven different types of GPUs. Two main algorithms were evaluated: merge sort and radix sort. Heat maps and S-curve analysis were utilized to visualize and evaluate performance metric and drawbacks. The performance of each sorting strategy was also analyzed based on segment sizes. The study concluded that a strategy should be chosen based on segment size and count in order to optimize performance. The authors suggest future work for evaluating strategies for sorting arrays with various data types. They also suggest a new radix sort variant that considers both the values and their segment indicies during sorting.

4.8 Study 8: Performance Evaluation of Parallel Count Sort using GPU Computing with CUDA

This study looks at parallelizing count sort using GPU computing in order to make it more capable of handling large sets of data. This is done by using CUDA to manage computation tasks that are otherwise not managed efficiently. The paper tests both parallel and sequential versions of count sort on various types of data and data distributions. The parallel count sort outperforms the sequential version in all cases with up to 66 times better efficiency. The main takeaways of this study are that GPU computing improves performance of count sort, can handle large data sets, and is applicable to data of various types.

4.9 Study 9: Practical Massively Parallel Sorting

This study presents generalizations of sample sort and multiway merge sort that lead to major improvements to scalability. This allows for data movement and startup overhead to be optimized. This research presents several valuable tools including a simple sorting algorithm for inputs that are very small. It also presents an algorithm for data distribution that minimizes message startups. The study addresses the scalability limitations of certain sorting algorithms by introducing multi-level sorting techniques. These reduce communication volume. This creates a new benchmark for parallel sorting performance due to how well optimized these algorithms are.

4.10 Study 10: Parallel Sorting with Minimal Data

The study tackles the problem of parallel sorting when each process contributes to only one data item. Previous solutions to this problem require O(p) memory and O(p log p) time. This leads to inefficiency when there is many processes. The authors present some novel algorithms. One of these is a basic linear algorithm which is optimal for up to 100,000 processes. They also present a reduced memory algorithm which only requires O(1). This decreases the memory usage but increases the runtime. The paper succeeds in addressing the scalability and efficiency problems with parallel sorting with minimal data. It correctly suggests use of a scalable algorithm that combines minimal memory usage with good time complexity, leading to a very efficient algorithm.

5 PERFORMANCE RESULTS

Fig 1.

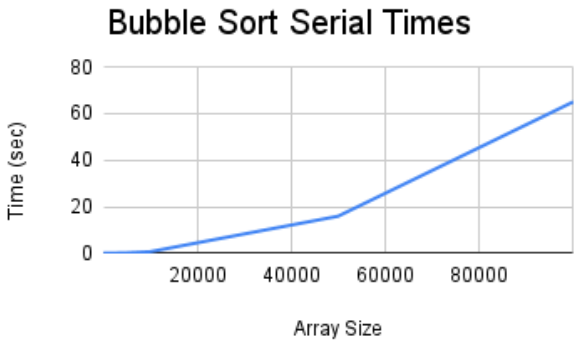
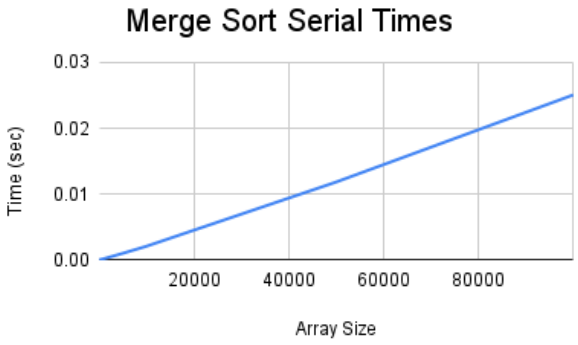


Fig 2.



The above graphs display times of serial runs among a vast range of array sizes for bubble sort (Fig 1) and merge sort (Fig 2).

Fig 3.

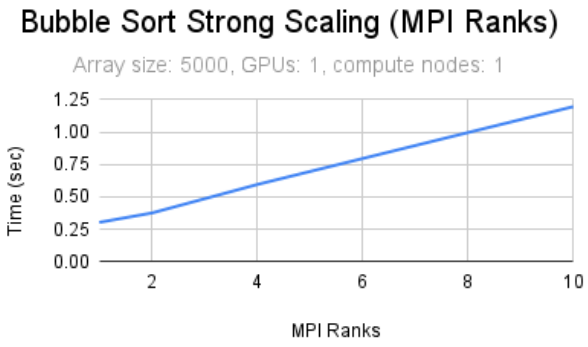


Fig 4.

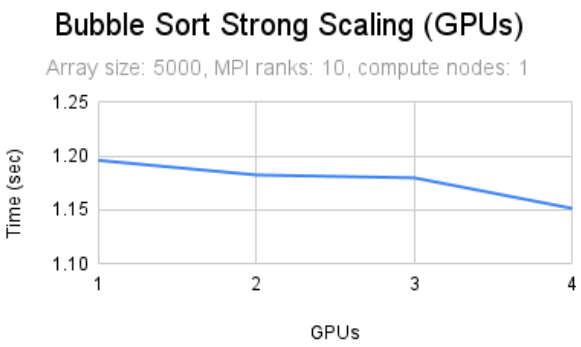
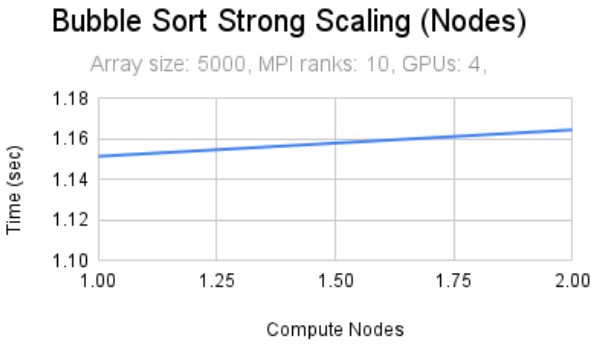


Fig 5.



The above graphs display strong scaling for our parallel implementation of bubble sort. These graphs show scaling according to MPI ranks (Fig 3), GPUs (Fig 4), and compute nodes (Fig 5) with a fixed array size of five thousand integers.

Fig 6.

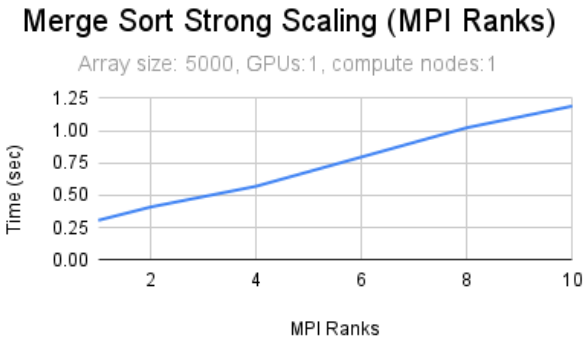


Fig 7.

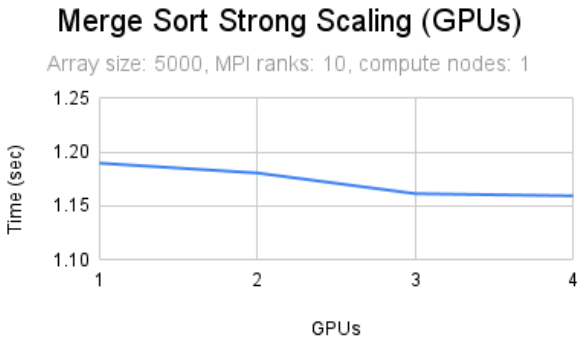
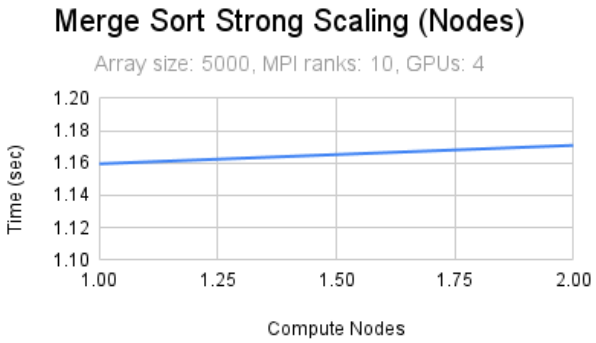


Fig 8.



The above graphs display strong scaling for our parallel implementation of merge sort. These graphs show scaling according to MPI ranks (Fig 3), GPUs (Fig 4), and compute nodes (Fig 5) with a fixed array size of five thousand integers.

Fig 9.

Mergesort Weak Scaling

Array Size	MPI Ranks ▲	GPUs	Compute Nodes	Total Time (sec)
1000	1	1	1	0.306591
10000	8	2	1	0.975835
50000	10	4	1	1.331311
100000	10	4	2	1.506099

Fig 10.

Bubblesort Weak Scaling

Array Size	MPI Ranks	GPUs	Compute Nodes	Total Time (sec)
1000	1	1	1	0.29497
10000	8	2	1	0.970796
50000	10	4	1	1.182803
100000	10	4	2	1.782646

The above graphs display weak scaling for both merge sort and bubble sort.

Fig 11.

Mergesort I/O Times

Array Size	MPI Ranks	GPUs	Compute Nodes ▲	I/O Time (sec)	% Time
1000	1	1	1	0.00095	0.3098590631
10000	8	2	1	0.00115	0.1178477919
50000	10	4	1	0.00135	0.1014038042
100000	10	4	2	0.00105	0.06971653258

Fig 12.

Bubblesort I/O Times

Array Size	MPI Ranks	GPUs	Compute Nodes ▲	I/O Time (sec)	% Time
1000	1	1	1	0.001702	0.5770078313
10000	8	2	1	0.003976	0.409560814
50000	10	4	1	0.00444	0.3753795011
100000	10	4	2	0.003691	0.3120967728

The above tables display total I/O time and percent of total runtime spent on I/O based on the weak scaling runtimes previously shown.

Fig 13.

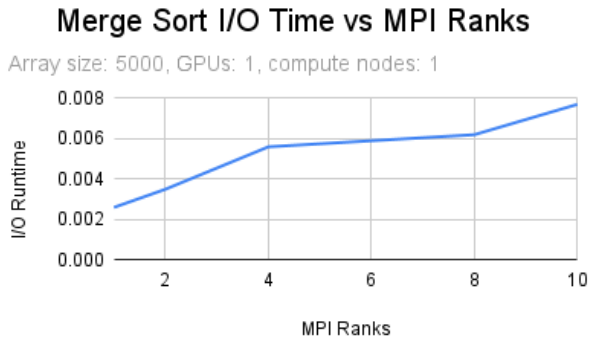
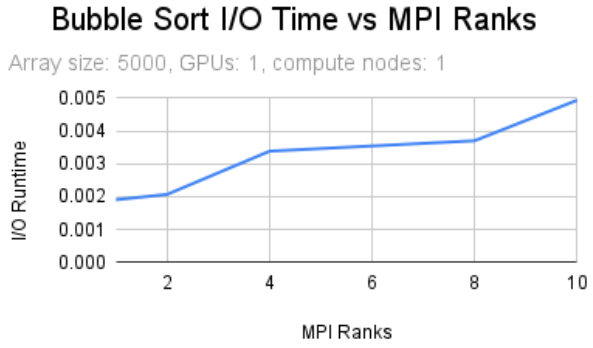


Fig 14.



The above graphs display total I/O time for different numbers of MPI ranks with fixed array size, GPUs, and compute nodes from the strong scaling runs for both merge sort and bubble sort.

Fig 15.

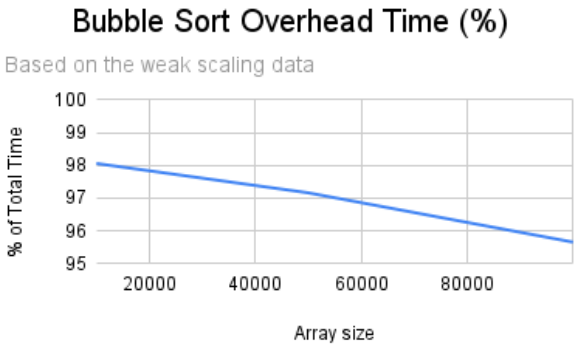
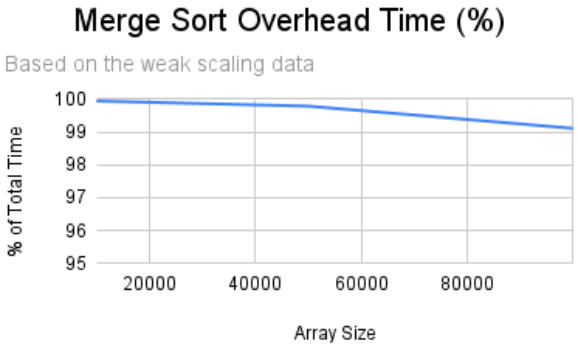


Fig 16.



The above graphs display the percentage of the total runtime that overhead took for both bubble sort and merge sort during our weak scaling tests.

6 ANALYSIS OF RESULTS

6.1 Analysis of Sequential Results

In this section, we will briefly analyze the sequential results of both bubble sort and merge sort.

Bubble sort (Fig 1) and merge sort (Fig 2) in serial show a positive correlation between runtime and input array size. This is typical for sequential algorithms that cannot split up the work like parallel computation. An important note is to see the difference between the two algorithms at large array sizes. Merge sort displays a O(nlogn), whereas bubble sort has O(n^2) causing this difference. We strongly recommend using merge sort over bubble sort if used in serial due to the data shown.

6.2 Analysis of Strong Scaling Results

In this section, we will analyze the strong scaling results of both bubble sort and merge sort based on the number of MPI ranks, GPUs, and compute nodes with a constant input array size of five thousand integers.

We see similar trends for all results of strong scaling in both the merge sort and bubble sort parallel algorithms. When applying strong scaling with respect to MPI ranks, we see that there is a positive correlation between the number of MPI ranks and total

runtime (Fig 3, Fig 6). Due to our implementation of the parallel algorithms described above, we can see that at this input size, the added overhead from having more MPI ranks outweighs the benefits, creating increasing runtimes. For both algorithms, they run at about 0.25 seconds for 1 MPI rank and linearly increase to about 1.25 seconds at 10 MPI ranks.

On the other hand, when applying strong scaling with respect to the number of GPUs, we see that runtimes decrease as the number of GPUs increases for both merge sort and bubble sort (Fig 4, Fig 7). For both algorithms, they start around 1.2 seconds for 1 GPU and linearly decrease to around 1.15 seconds for 4 GPUs. This is expected behavior of parallel algorithms in general and shows that the additional overhead corresponding to more GPUs is outweighed by the benefits in our test cases.

Finally, when applying strong scaling with respect to the number of compute nodes, we again saw a slight positive correlation between number of compute nodes and runtime (Fig 5, Fig 8). This correlation tells us that, with our test cases of five thousand integer array data, the added overhead of having more compute nodes outweighed the benefits.

In this section, we showed that the benefit of having additional GPUs outweighed the extra overhead time, however this was not the case with MPI ranks or compute nodes. With larger datasets, we expect that the benefits of having additional MPI ranks and compute nodes will outweigh the negative effect of increased overhead. For our tests, the benefit of just using CPUs via MPI was outweighed by the additional overhead, whereas the benefit of hybrid CPU/GPU with both MPI and CUDA made the performance better.

6.3 Analysis of Weak Scaling Results

In this section, we will analyze the weak scaling results of both bubble sort and merge sort.

For both merge sort and bubble sort, we see similar times when incrementing the number of CPU/GPU/nodes along with input size (Fig 9, Fig 10). This is vastly different than what we saw when comparing the serial times with the same input values. In serial, the bubble sort algorithm performed much worse than that of merge sort with large datasets. However, in parallel, it was able to stay much closer to that of the merge sort. This shows how being in parallel can greatly reduce the runtime of algorithms with large datasets as it greatly reduces the computation time at a relatively small cost of additional overhead. On the other hand, for merge sort, we see that the runtimes are not better than the serial version at the higher datasets tested. This is because merge sort in serial is a more efficient sorting algorithm than bubble sort (O(nlogn) vs O(n^2)). So the serial runs are much faster at higher datasets than that of bubble sort, and therefore, with the datasets we tested, the additional overhead of being in parallel outweighed the benefits of faster computation.

If tested with larger datasets, we can expect that the benefits of being in parallel will eventually outweigh the overhead, and the parallel runs will become faster than the runs in serial not only for bubble sort but for merge sort as well.

6.4 Analysis of I/O Runtime Results

In this section, we will analyze the effects of MPI I/O on the total runtime.

With our implementation of MPI I/O mentioned earlier, we see that the total I/O time is relatively small compared to total runtime (Fig 11, Fig 12) and increases with an increase in MPI ranks (Fig 13, Fig 14). As more MPI ranks are added, we have more communication, and thus our I/O time increases at a nearly linear rate. Another important note is that we saw similar I/O times with a changing number of GPUs and compute nodes. These two variables have little affect on the I/O runtime, unlike the MPI rank number.

6.5 Analysis of Overhead Time

In this section, we will analyze the effects of overhead time on the total runtime of our algorithms in parallel.

For both merge sort and bubble sort, we see that time spent on overhead takes up most of the runtime (Fig 15, Fig 16). For bubble sort, at smaller input sizes, we see the overhead taking around 98% of the total runtime, and it drops below 96% for our largest datasets. For merge sort, at smaller input sizes, we see the overhead taking nearly 100% of the total runtime, and it drops to 99% for our largest datasets. This discrepancy is likely due to the time complexity difference between the two algorithms. They will have a similar overhead time, but the computation time is slightly longer for bubble sort, driving down the percentage that is overhead time. In general, with the datasets that we tested, the majority of time that is overhead is much greater than the time spent computing, which is why the overall time is longer in parallel for merge sort and for small datasets of bubble sort as compared to the serial versions.

7 CONCLUSION

In this paper, we found that the parallel version of bubble sort was somewhat slow with low input sizes but fast with high input sizes compared to the serial version. However, for merge sort, the parallelized version was consistently slower than the serial version. This added time is due to the overhead required for additional resources. With large data sets, the benefits of parallelizing the algorithms begin to be seen and outweigh the negative effects of overhead. Strong and weak scaling of each algorithm based on MPI ranks, GPUs, and compute nodes was also considered. For bubble sort, parallel was faster than serial for large sets of data. For merge sort, this was not seen. This is due to the differences in the time complexity of the algorithms (O(nlogn) vs O(n^2)). As a result, the benefit of parallelizing merge sort was not fully seen from the datasets tested in this paper. Bubble sort struggles with large data sets in serial, so parallelizing it led to major improvements. The key takeaways from the analysis in this paper highlight the importance of selecting the appropriate algorithm for parallelization based on data characteristics and overhead. The contrast between bubble sort and merge sort in their parallel implementations shows that not all algorithms benefit equally from parallelization given the same input size. Dealing with large data sets is becoming increasingly important for various fields which makes the analysis done in this study very relevant.

8 FUTURE WORK

There is a considerable amount of potential future work to build off of this paper. Firstly, many more sorting algorithms can be implemented in parallel. Then, these algorithms along with the ones mentioned in this paper can be analyzed together. This will allow for determining which algorithm is the most efficient in parallel and which functions have the biggest improvement from serial to parallel versions. Additionally, future work might entail creating a new algorithm that combines elements of existing sorting techniques to create a more optimized algorithm. This is similar to the hybrid algorithms discussed in some of the related works. Furthermore, another direction could be the examination of energy efficiency across different algorithms that deal with massive data sets. This is becoming an increasingly important factor when it comes to sustainable computing. Finally, implementing different sorting algorithms in a variety of hardware architectures would also be a valuable direction that would provide information about the adaptability and efficiency of these algorithms.

9 REFERENCES

Alisa, Zainab. (2018). Parallel Computing for Sorting Algorithms.

Axtmann, Michael, Timo Bingmann, Peter Sanders, and Christian Schulz. "Practical Massively Parallel Sorting." *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '15. Portland, Oregon, USA: Association for Computing Machinery, 2015. 13-23. Web. <https://doi.org/10.1145/2755573.2755595>.

Faujdar, Neetu and Ghrera, Satya. (2015). Performance Evaluation of Merge and Quick Sort using GPU Computing with CUDA. The International journal of applied engineering(IJAER) ISSN 0973-4562. 10. 3979-3982.

Faujdar, Neetu, and SatyaPrakash Ghrera. "Performance Evaluation of Parallel Count Sort Using GPU Computing With CUDA". Indian Journal of Science and Technology, vol. 9, no. 15, May 2016, doi:10.17485/ijst/2016/v9i15/132480.

Gökahmetoğlu, Hakan. "A Study of Parallel Sorting Algorithms Using CUDA and OpenMP." Aug. 2018. Web. https://acikbilim.yok.gov.tr/handle/20.500.12812/58622.

Jan, Bilal. (2012). Fast Parallel Sorting Algorithms on GPUs. International Journal of Distributed and Parallel systems. 3. 107-118. 10.5121/ijdps.2012.3609.

Peters, Hagen, Ole Schulz-Hildebrandt, and Norbert Luttenberger. "Fast In-place, Comparison-based Sorting with CUDA: A Study with Bitonic Sort." Concurrency and Computation: Practice and Experience 23.7 (2011): 681-693. Web. https://doi.org/10.1002/cpe.1686.

Schmid, Rafael F., Flávia Pisani, Edson N. Cáceres, and Edson Borin. "An Evaluation of Fast Segmented Sorting Implementations on GPUs." Parallel Computing 110 (2022): Article 102889. ScienceDirect. Elsevier. Web. https://doi.org/10.1016/j.parco.2021.102889.

Siebert, C., Wolf, F. (2011). Parallel Sorting with Minimal Data. In: Cotronis, Y., Danalis, A., Nikolopoulos, D.S., Dongarra, J. (eds) Recent Advances in the Message Passing Interface. EuroMPI 2011. Lecture Notes in Computer Science, vol 6960. Springer, Berlin, Heidelberg.

Żurek, Dominik and Marcin, Pietron and Wielgosz, Maciej and Kazimierz, Wiatr. (2013). Comparison Of Hybrid Sorting Algorithms Implemented On Different Parallel Hardware Platforms. Computer Science. 14. 679. 10.7494/csci.2013.14.4.679.