Team 4 - Final Report

Members:
Naimur Rahman
Emma Ong
Luis Martinez Morales
Anna Huang

Note: All respective graphs made are in the directory as pdfs based on the algorithm.

Report:

Background:

The primary aim of this project was to assess and compare the parallelization efficiency of four distinct sorting algorithms—Bitonic, Radix, Quick, and Merge Sort—leveraging the MPI and CUDA libraries within a C++ framework. The parallelization of Quick Sort presented notable complexities, as its design is not inherently conducive to parallelism. Quick Sort CUDA was parallelized using GPU memory allocation. The array generated is initialized on the host and data is transferred from the host to device. A quicksort kernel is also configured to launch with a grid of blocks and threads. The Quick Sort algorithm is launched on the GPU as a kernel and each thread will execute the Quick Sort algorithm on different parts of the array. Instead of using a recursive approach, an iterative Quick Sort algorithm was used to keep track of segments of each thread that needed sorting. Onec the sorting is done, the sorted array is copied back from device to host. Quick Sort MPI was implemented by dividing the array into smaller sub-arrays. Each MPI process sorts a sub-array using quicksort. Once all the processes have completed independent sorting, the root process gathers all sorted subarrays and merges them into a single array. Bitonic and Radix sort, on the other hand, were able to be parallelized pretty easily for both Cuda and MPI.

In contrast, Bitonic and Radix Sorts exhibited a more straightforward translation to parallelism due to their predictable data flow patterns, which align well with the parallel processing capabilities of CUDA and MPI.

Performance metrics were meticulously gathered using Caliper, focusing on computational segments, data transfer operations, and the main sorting function. The algorithms underwent testing across a spectrum of array sizes—2^16, 2^18, 2^20, and 2^22—and were benchmarked against various data distributions, including completely random, 1% perturbed, already sorted, and reverse sorted datasets. For representational clarity, graphs predominantly displayed results from the random data set, which mirrors the common real-world scenario of unsorted data.

For the MPI implementations, we executed tests on arrays with the aforementioned sizes and data types, using processor counts of 2, 4, 8, 16, 32, and 64. The CUDA tests utilized thread

blocks of sizes 64, 128, 256, 512, and 1024. Baseline sequential performance was also measured for each algorithm by running single-processor MPI versions and single-thread CUDA versions, establishing a reference point for speed-up analysis.

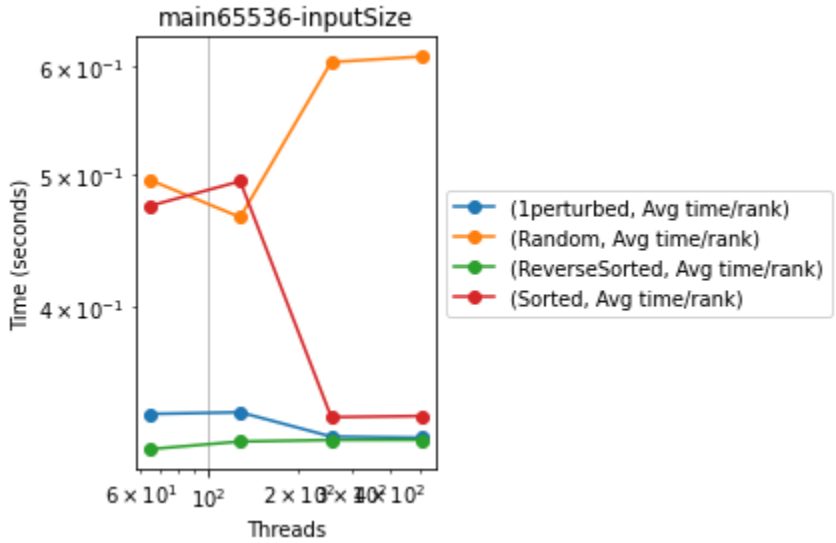
In our analysis, we compared algorithm performance through graphs depicting strong scaling, weak scaling, and speed-up metrics. Our final comparative study focused on evaluating the main runtime performance of CUDA implementations against MPI for Bitonic and Radix Sorts, providing insights into their respective parallelization advantages.
Quick sort was not as efficient as these two algorithms so its main run times were not compared.

Bitonic Sort:

Cuda:

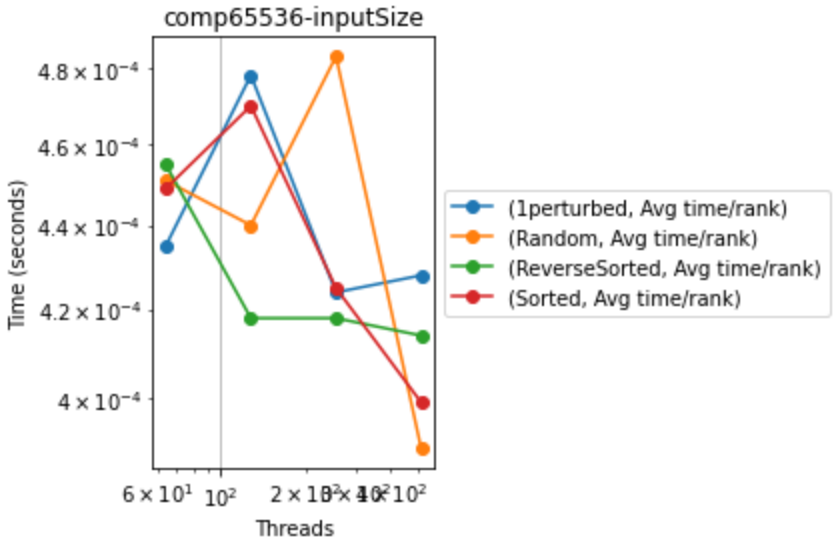
Strong scaling:

Figure 1.1 - **Main** Function Region
inputSize = 65536



Note: From figure 1.1 it is clear that bitonic sort had trouble with random but sorted and reverse sorted inputs performed better. This might be due to the fact that randomness makes the sorting process harder. There are points where the runtime decreases but not all the time.

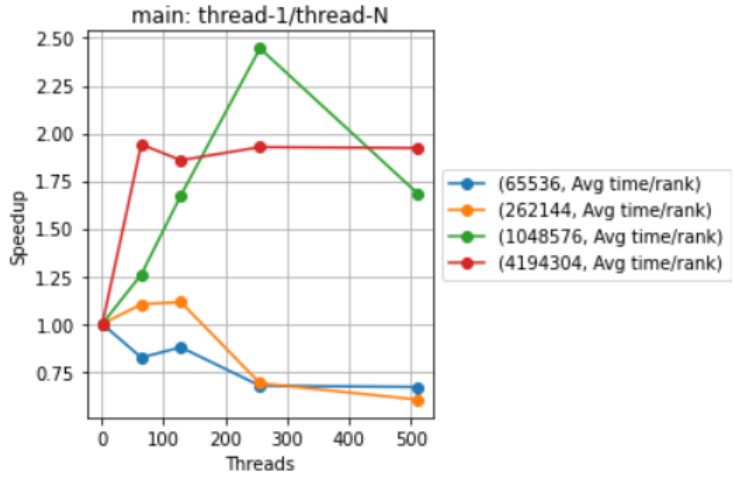
Figure 1.2 - **Comp Large** Function Region
inputSize = 65536



Note: In figure 1.2 the general trend is that the run time decreases with a fixed array size for sorting. This graph follows the expected result.

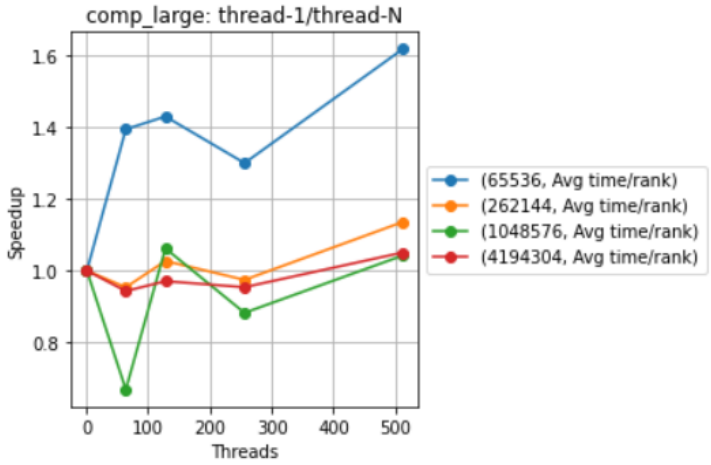
Speedup:

Figure 1.3 - **Main** Function Region
InputType= Random



Note: The speedup graph looks accurate except for array size=10485576. The other speedup lines are flat indicating that this is what we wanted as our outcome.

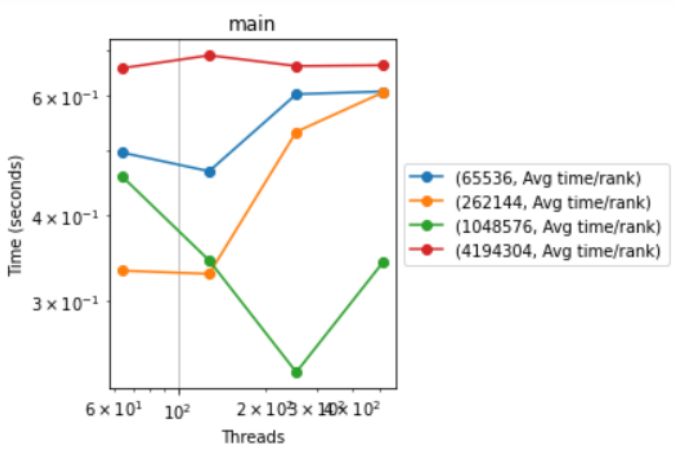
Figure 1.4 - **Comp Large** Function Region
InputType = Random



Note: The speedup graph doesn't look as nice as the previous graph however it is important to note that each line is parallel at higher thread sizes.

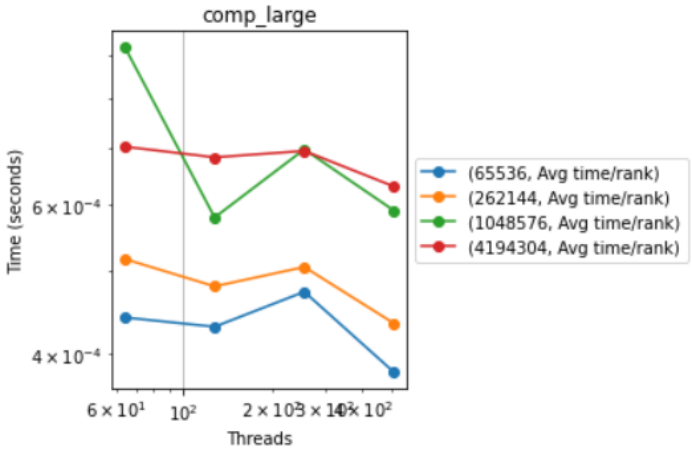
Weak Scaling:

Figure 1.5 - **Main** Function Region
InputType = Random



Note: The weak scaling follows the correct trend at the lower end threads. However, at the later points it increases which shows bad results in terms of a weak scaling graph.

Figure 1.6 - **Comp Large** Function Region
InputType = Random



Note: The figure above somewhat shows the correct trend at the middle threads of what the weak scaling graph should look like. However, at the smaller and larger thread sizes there is a decrease which shouldn't be the case but could be happening because of communication overheads such as memcpy.

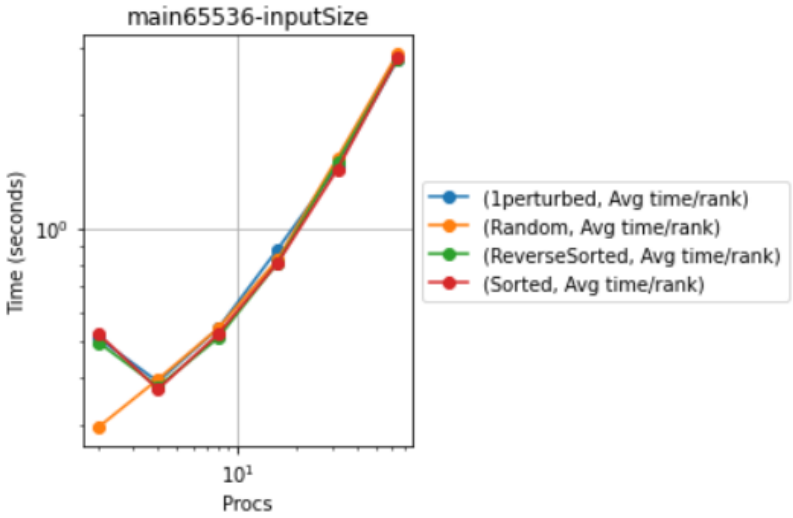
Cuda Observations:
Overall, I am pleased with the results that came from the Cuda Implementation of Bitonic Sort. Speedup and Strong Scaling are easily the most accurate and weak scaling has a slight hiccup due to communication overhead. It is true that the main method didn't always output the intended results and that sometimes the comp function had the best results. This indicates that the main as a whole sometimes gives a better representation of the algorithm as a whole and it takes in the smaller details. Comp lets us know that increasing the number of threads does not always result in a better performance.

Like mentioned before, I really liked how accurate the strong scaling graph was for comp. It demonstrates that by keeping the array size fixed and increasing the number of threads, the runtime will get smaller.

MPI:

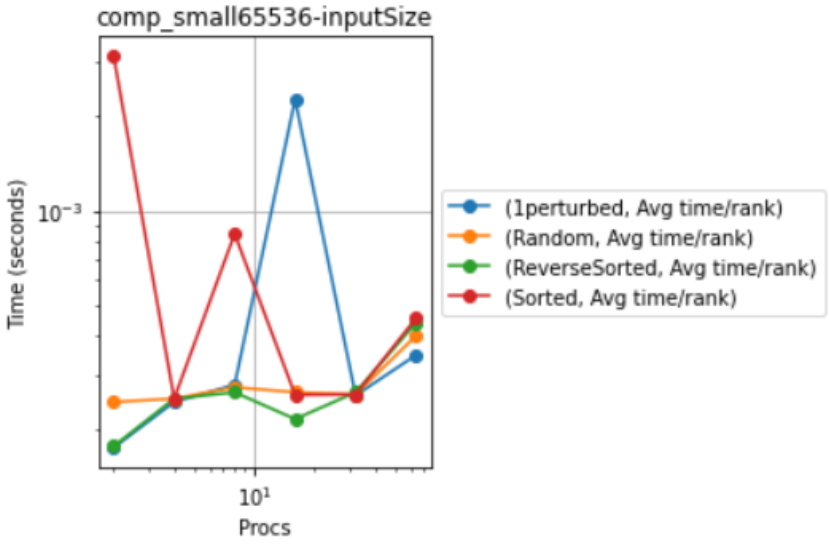
Strong scaling:

Figure 1.7 - **Main** Function Region
InputSize = 65536



Note: The figure above does not show a very accurate result of what should be going on behind the scenes. The ideal result should be that the time decreases when keeping a fixed array size.

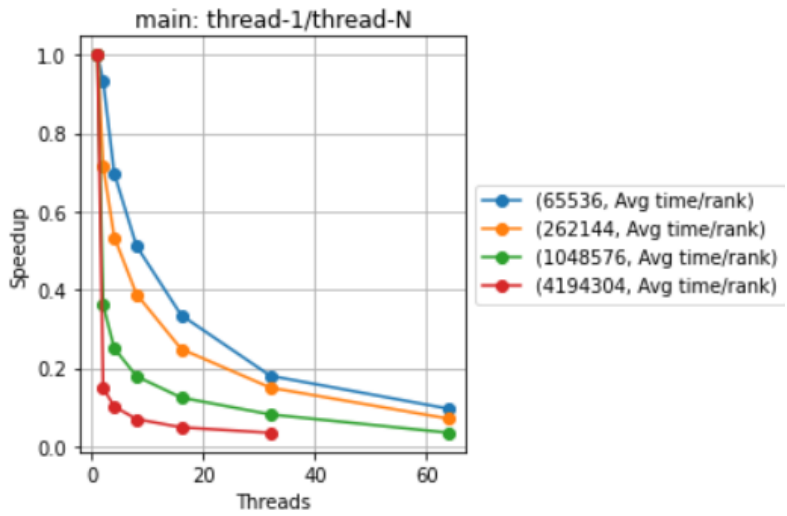
Figure 1.8 - **Comp Small** Function Region
InputSize = 65536



Note: The figure above shows better promise than the main function graph. While at some points increasing the number of procs decreases the run time, there are spikes that show increasing the number of procs actually increases run time. This could be due to the nature of varying the input types or increased communication.

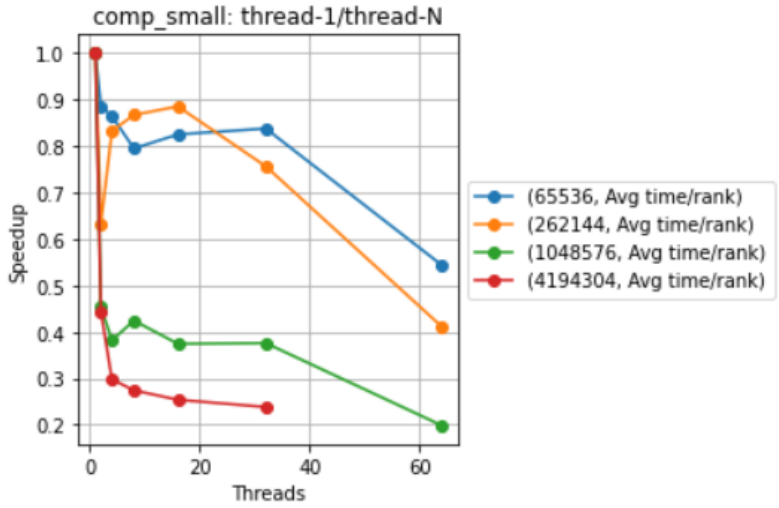
Speedup:

Figure 1.9 - **Main** Function Region
InputType = Random



Note: The speedup for the MPI main function region shows an interesting result. While the speedup keeps decreasing as the number of procs decreases. Something good that comes out of this graph is the fact that the large input sizes show consistently less speedup. That makes a lot of these line segments almost parallel.

Figure 1.10 - **Comp Small** Function Region
InputType = Random

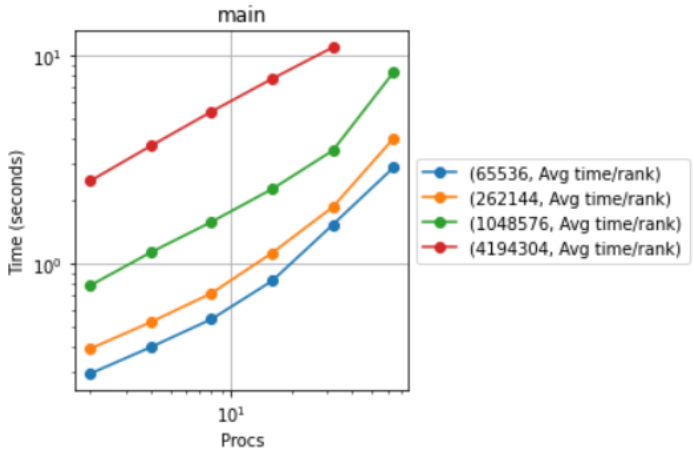


Note: The speedup graph above for comp small function region looks a little more promising than the previous graph. It appears that speedup again is good at the middle number of procs. However the speedup is a little wrong at the smaller and larger end of the procs. There is a distinct plateau at the center. Something to note is that when I tried running 64 procs on arraysize 4194303 the job was never completed.

Weak Scaling:

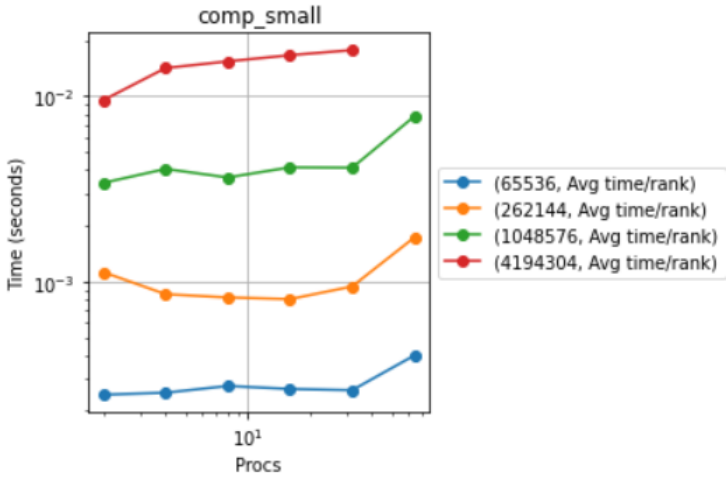
Figure 1.11 - **Main** Function Region

InputType = Random



Note: The main function region plot looks great. It is clear to see that increasing the number of procs along with the arraysize shows a linear relationship. This graph clearly shows an accurate trend.

Figure 1.12 - **Comp Small** Function Region
InputType = Random



Note: The figure above somewhat shows the correct trend at the middle threads of what the weak scaling graph should look like. However, at the smaller and larger thread sizes there is a decrease which shouldn't be the case but could be happening because of communication overheads such as memcpy.

MPI Observations:
I would like to mention that strong scaling was not the best indicator of this algorithm's performance. I believe that the time increased with more procs because of overhead due to communication. Maybe a way to counteract this would be to make the algorithm a bit more efficient. I did have to use a version of merge sort to be able to sort with bitonic sort.

Speedup also did not give the best results. This was especially seen at the smallest and largest procs. However, it is good to see that larger arrays sizes demonstrated less of a speedup. This goes to show that at least there is something consistent. Additionally, there was a plateau at the middle num procs.

Finally, I am very pleased with the results of weak scaling. There is clearly a positive correlation between the number of procs and the arraysize. Overall I really did like the results of CUDA Bitonic sort but MPI still shows great promise.

Radix Sort:

Cuda:

Strong Scaling for size of 65536:

Figure 2.1 - Large Computation Region

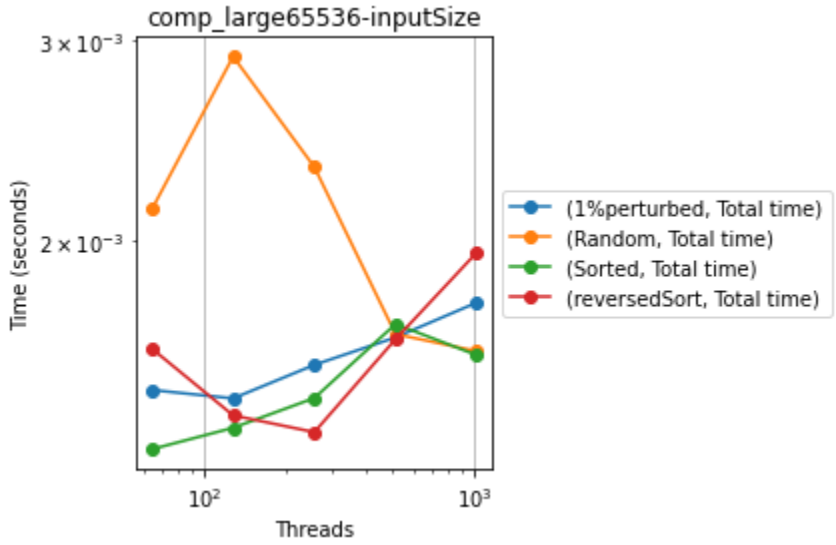
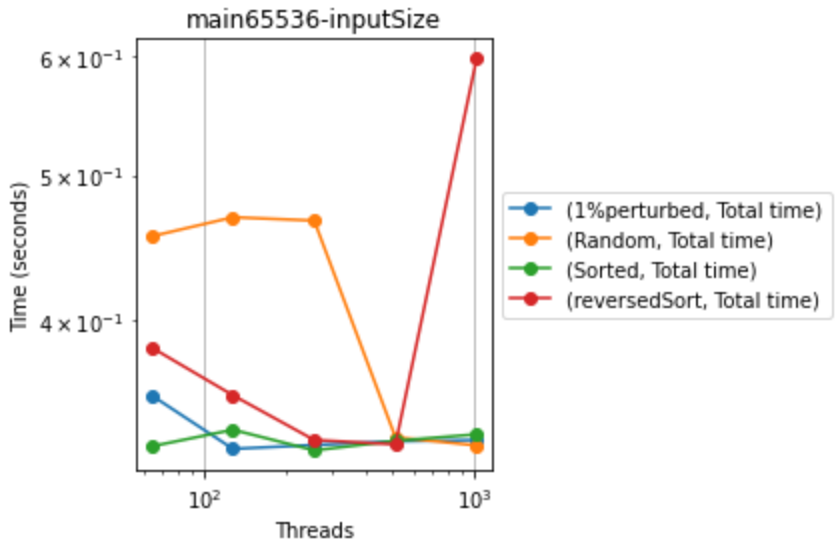


Figure 2.2 - Main Function Region



Weak Scaling for size of 65536:

Figure 2.3 - Large Computation Weak Scaling

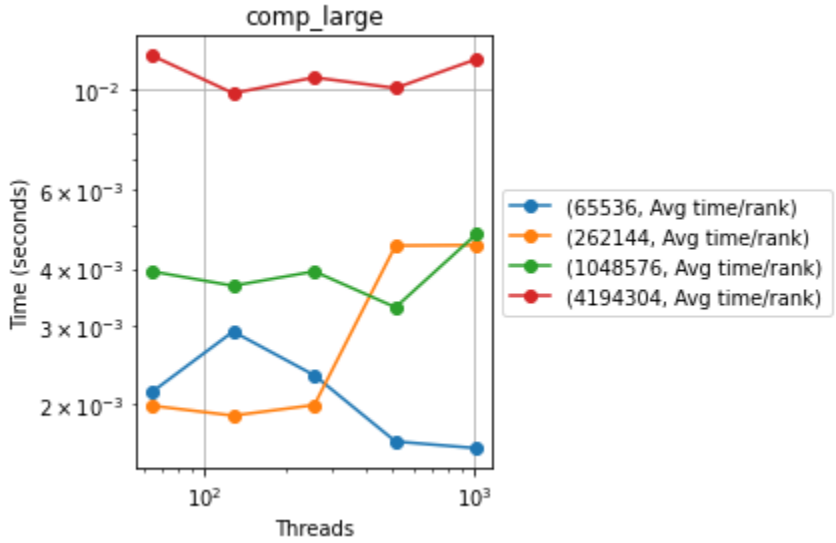
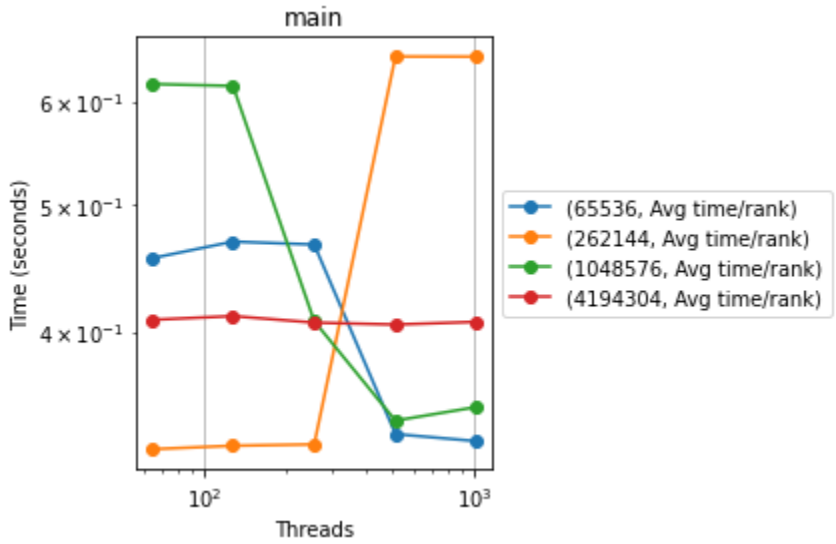


Figure 2.4 - Main Function Weak Scaling



Speedup for Random Datasets:

Figure 2.5 - Large Computation Speedup

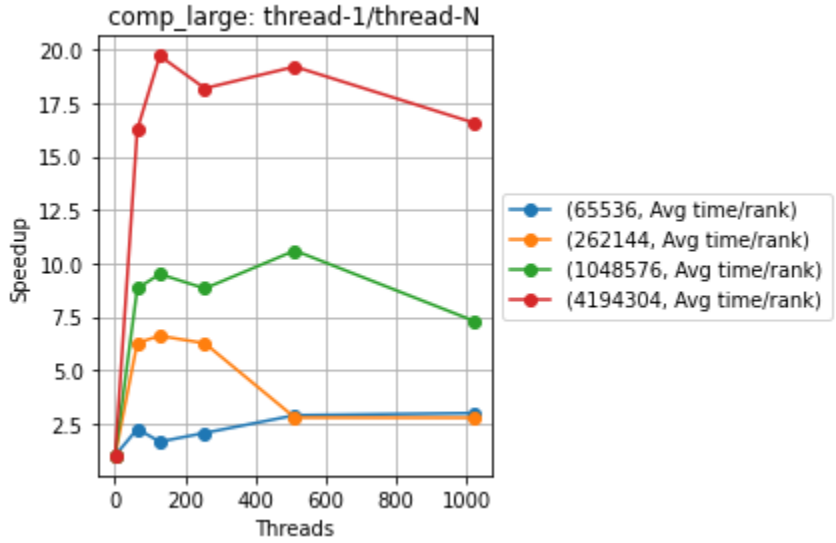
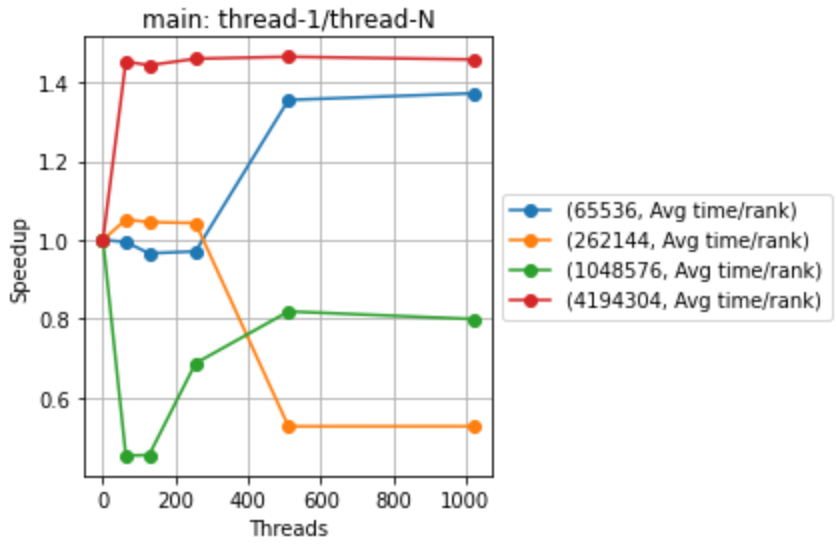


Figure 2.6 - Main Function Speedup



MPI Graphs:

Cuda Graphs:

Strong Scaling for size of 65536:

Figure 2.7 Large Computation

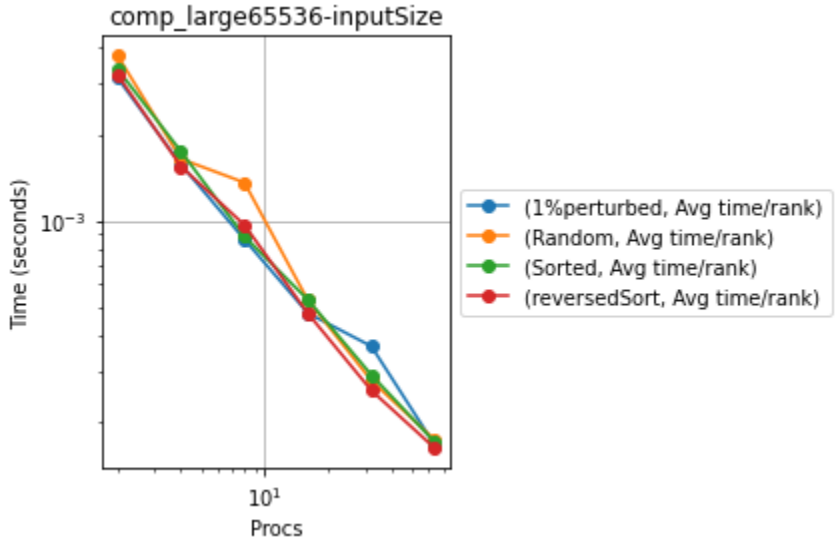
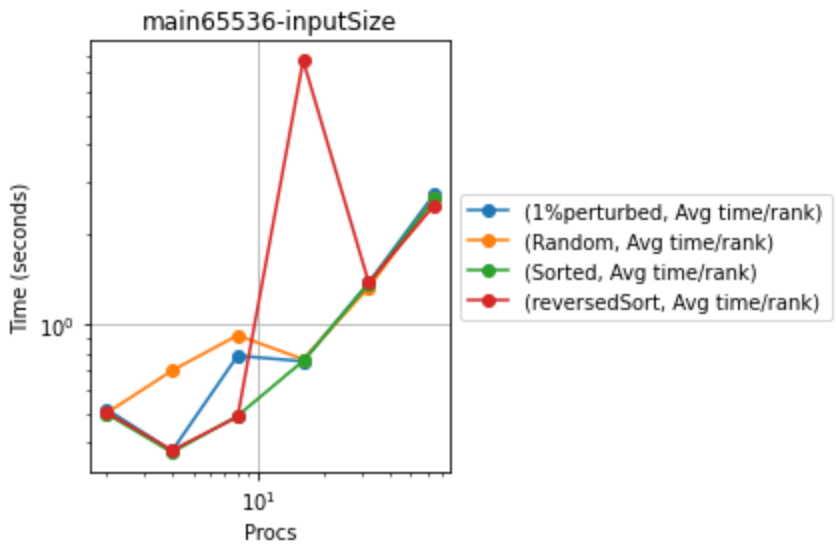


Figure 2.8 Main Function



Weak Scaling for Random Datasets:

Figure 2.9 Large Computation

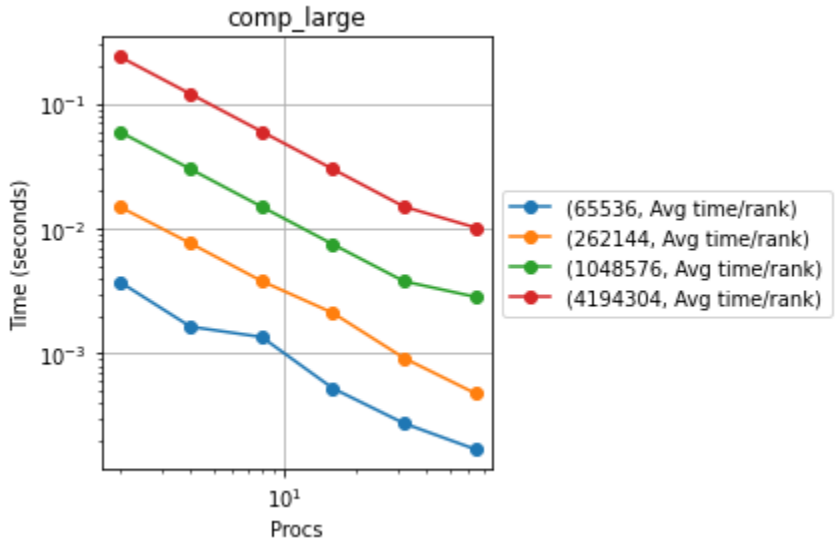
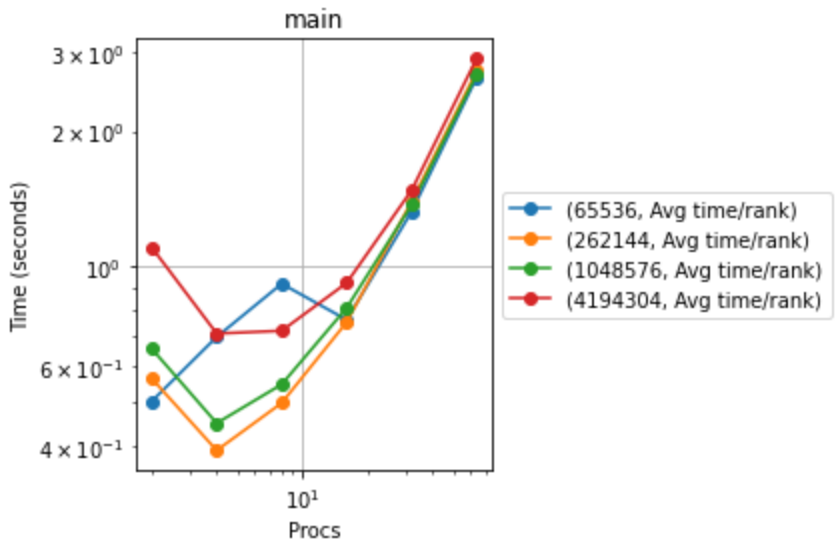


Figure 2.10 Main Function



Speedup for Random Datasets:

Note: While it says threads for the x-axis for the speed-up graphs, it really means the number of processors

Figure 2.11 Large Computation

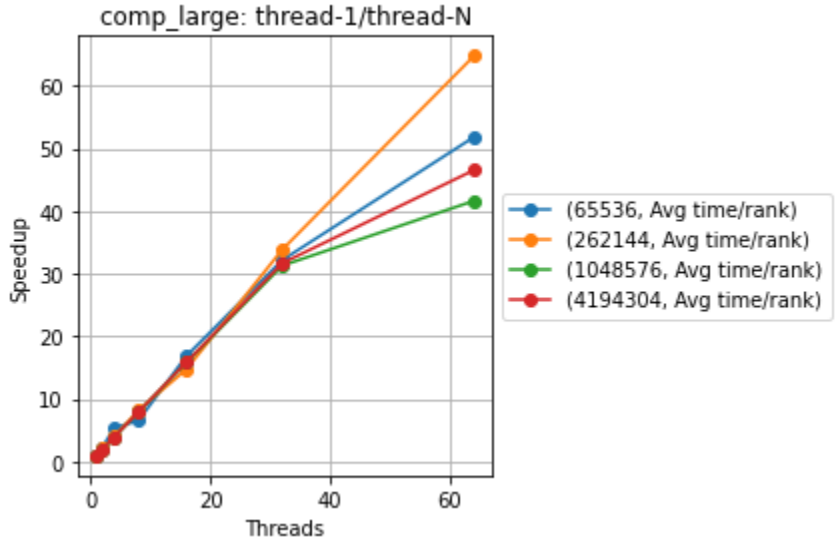
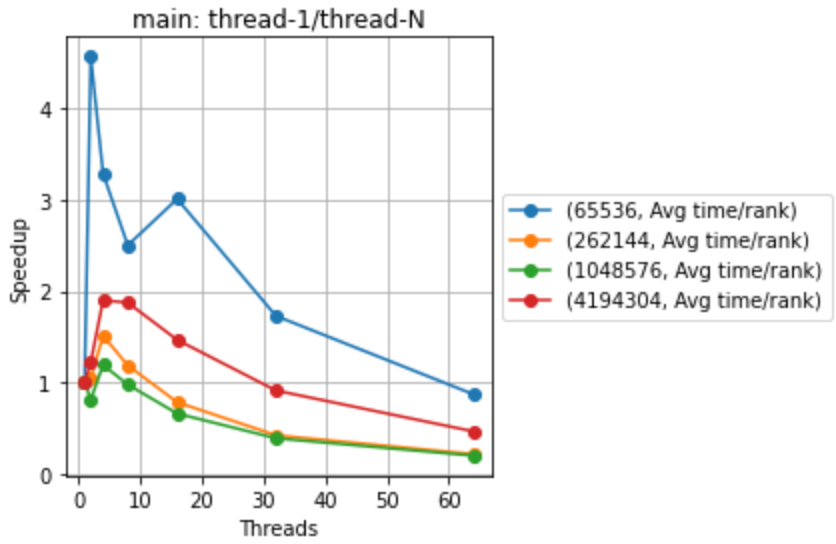


Figure 2.12 Main Function



Cuda Observations:

Regarding Radix sort in Cuda, the algorithm performed quite well both in terms of strong scaling, and weak scaling. In strong scaling, looking at figures 2.1 and 2.2, the randomly sorted datasets received the most over reduction in time as thread count increased whereas the reversed sorted dataset actually increased as thread count increased. Particularly, in the large computation region (figure 2.1), all the other datasets increased in time but the random sorting decreased on average. The other three sets started at low times and then greatly increased as thread counts did while the randomly placed dataset started at a much higher time for computation and greatly decreased as the threads increased. This most likely could be due to the nature of radix sorting as it does not really account for if the data is greatly sorted and as such, does a lot of unnecessary work.

The main function showed much better evidence of efficiency in terms of strong scaling as in figure 2.2, all datasets runtime decreased, except for reverse sorted. The random dataset again received the most reduction in run time due to increased threads as the perturbed and sorted datasets had diminished reductions. The reverse sorted datasets tended to decrease but increased at the maximum number of threads. As such, it seems more like an outlier than a clear trend of increase.

Weak scaling in figures 2.3 and 2.4 show some decreases but not a clear idea of the relative improvements in run time for large computations and the program as a whole. It's when looking at figures 2.5 and 2.6 that show the speedups of the large computation and main regions in the random datasets respectively where it is evident that parallelization and increasing thread counts decreases run time for the sorting algorithm. The higher speed up magnitudes tended to come with higher array sizes. The speedups tended to stay constant in the main function but topped out and slightly decreased or stagnated after a certain amount of threads.

MPI Observations:

For MPI Radix sort, there were definitely benefits to parallelization as shown by the decreasing run time per rank in terms of large computation in figure 2.7 whereas there was a trend of increase in the main function for all the datasets in figure 2.8. Weak scaling had a similar effect where all the array sizes tended to share the same trends as processor amounts increased where the large computations decreased linearly in figure 2.9 and the main function time per rank increased in parabolic increase as seen in figure 2.10. When looking at the speed ups for the random datasets, again it shows that large computation benefitted from MPI but the main threads speed up increased and then decreased after about 8 processors in the algorithm (figures 2.11 and 2.12). For figure 2.11, the speed up seemed to be linear and continually increasing unlike with Cuda. The magnitudes were definitely higher as well.

When looking at the figures for MPI radix as a whole, it is clear that for the main function there is a certain number of processors that is the most optimal, both for strong and weak scaling. Large computations times though tended to decrease linearly as processors increased. The lack of benefits in terms of main function average run time per rank could be due to the way the algorithm was set up. Additionally, a main contributor could be due to MPI's use of distributed memory as the communication overhead to send and receive data could lead to the increase as there are more processors to communicate. Computations seem to perform better in this program on a relative level compared to cuda but it is at the cost of relative run time.

Radix Conclusion:

Overall, the Cuda implementation of radix seemed to run better when comparing speed ups and the scaling, making it seem like the better overall algorithm between the two. More testing with greater array sizes and even more types of datasets could paint a more definitive picture. Additionally, most of the analysis regarding strong scaling was done with an array size of 65536. While weak scaling on the random datasets still indicates the conclusion that Cuda performs better overall, analyzing strong scaling with greater array sizes would help strengthen such an observation. Assumably, MPI's overall slower performance at higher processor counts compared to Cuda's overall increase in performance as thread counts increased could mean that there is an optimal number of processors that benefit parallelizing Radix in MPI and that increasing processors will not lead to performance speedups.

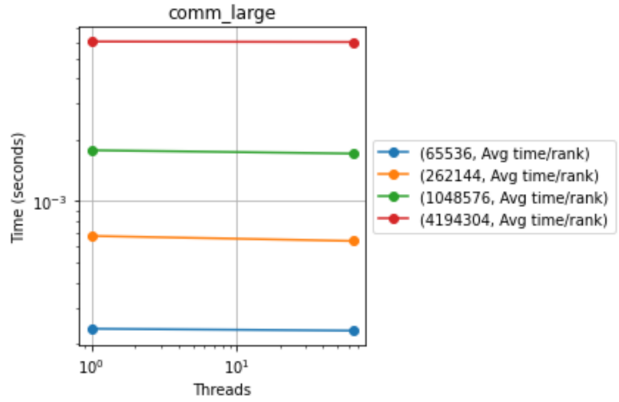
Since radix sort sorts based on the number of digits, there could be cases depending on the magnitude of data points being sorted where it may not be the most optimal. Data points in values ranging from 0 to 9999 were in the arrays when being tested. For a smaller number of digits per data point on average though, the cuda implementation definitely seems like a solid way to sort through the vast number of data points as a result of the speed ups given.

Quick Sort:

Cuda

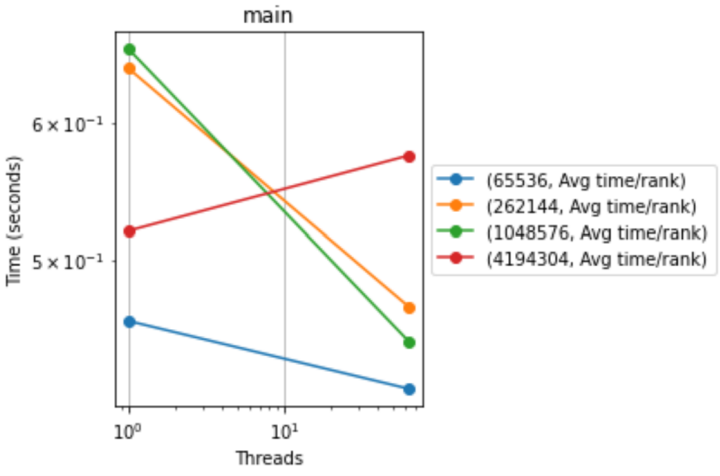
Strong scaling

Figure 3.1 - comm_large region



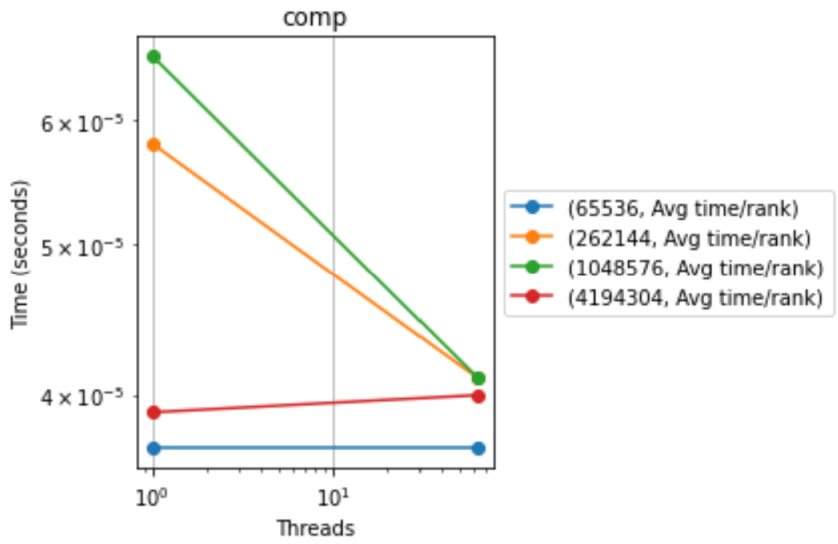
We should expect to see a linear decrease in time as we double the number of threads. However, because my quicksort cuda algorithm would time out at larger input sizes and processes, I wasn't able to plot more data. This goes for my other quicksort plots as well.

Figure 3.2 - main region



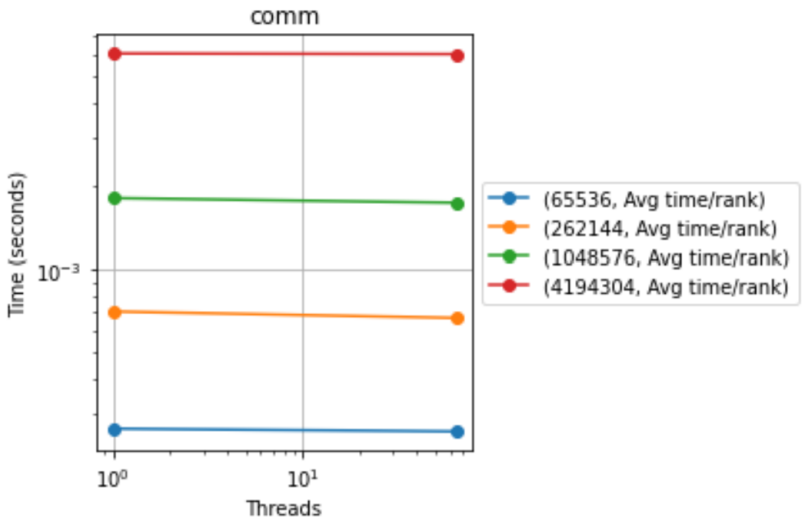
The line representing input size of 4194304 is counterintuitive since the line increases rather than decreases.

Figure 3.2 - comp region



The line representing input size of 4194304 isn't as expected as it shows an increasing trend.

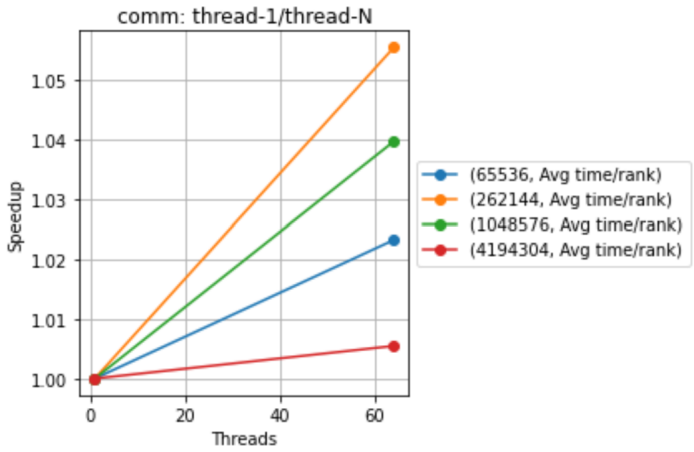
Figure 3.3 - comm



We should expect to see a linear decrease in time as we double the number of threads. However, because my quicksort cuda algorithm would time out at larger input sizes and processes, I wasn't able to plot more data. This goes for my other quicksort plots as well.

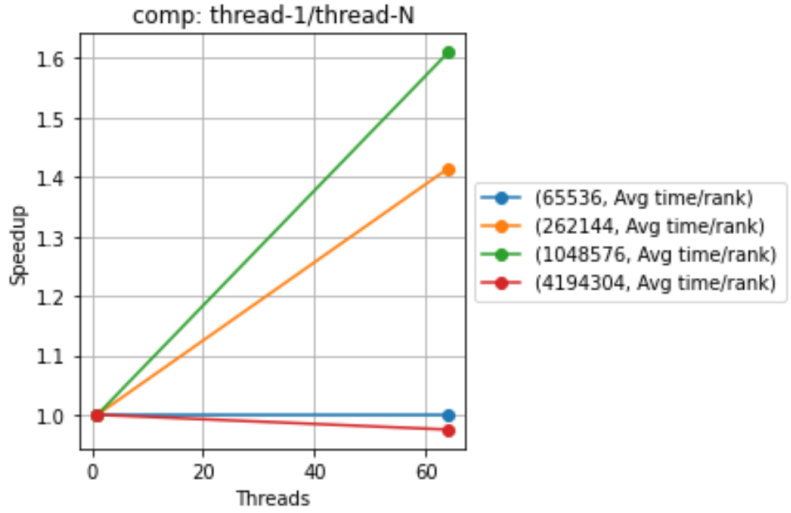
Speedup

Figure 3.4 - comm 1 thread



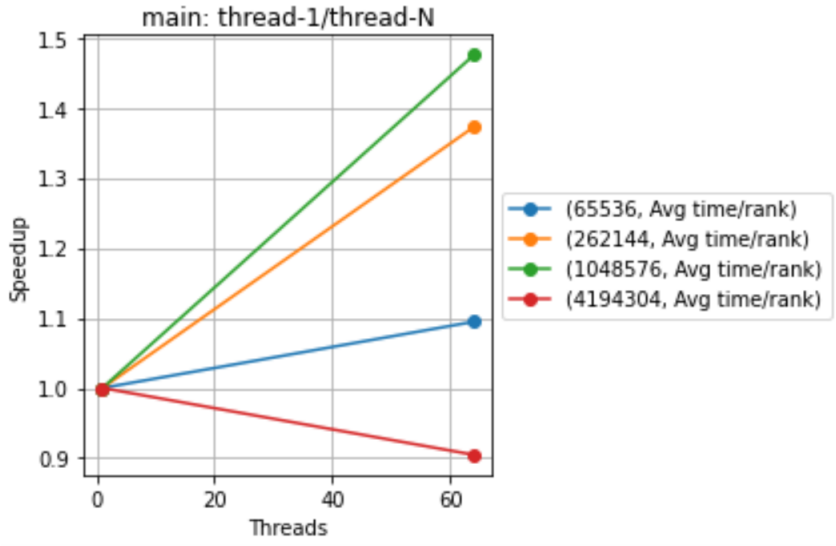
This speedup is as expected since the trend is increasing.

Figure 3.5 - comp



This speedup is as expected since the trend is generally increasing.

Figure 3.6 - main



This speedup is as expected since the trend is generally increasing.

Weak Scaling

Figure 3.7 - comp

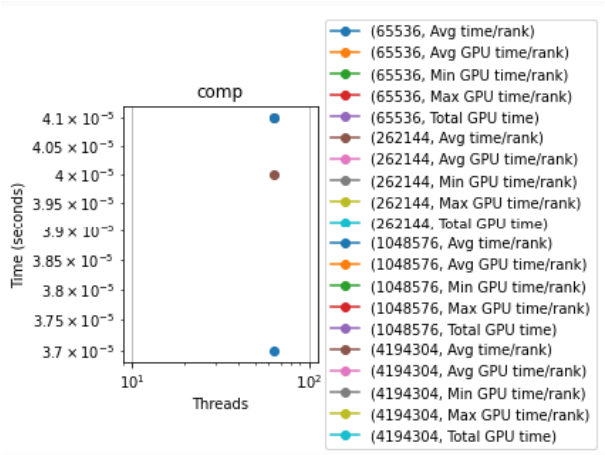


Figure 3.8 - main

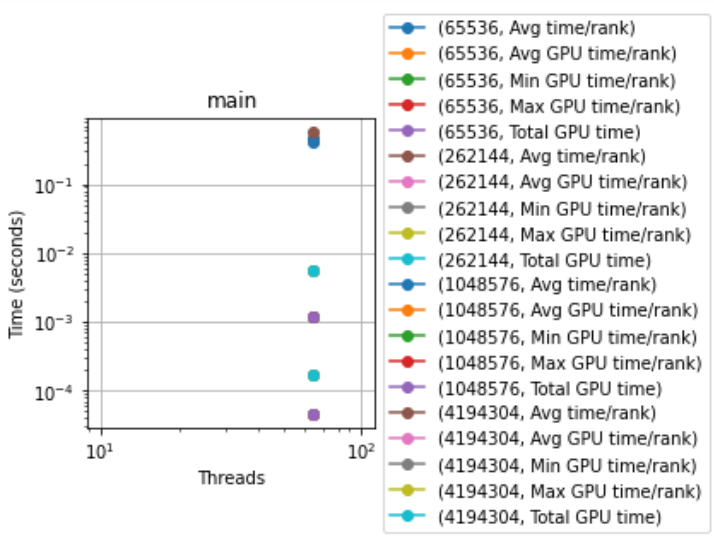
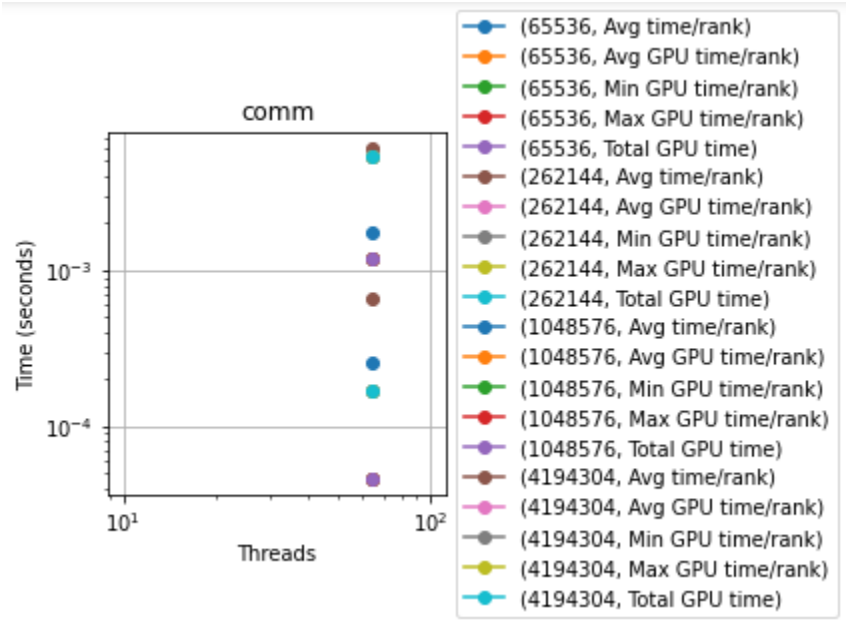


Figure 3.9 - comm

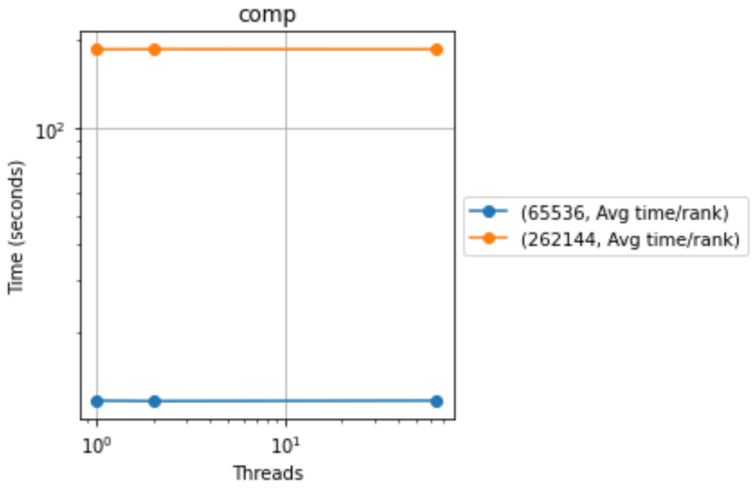


My weak scaling graphs appear to have just a single thread value at each input size because I was unable to run larger input and thread values due to time out.

MPI

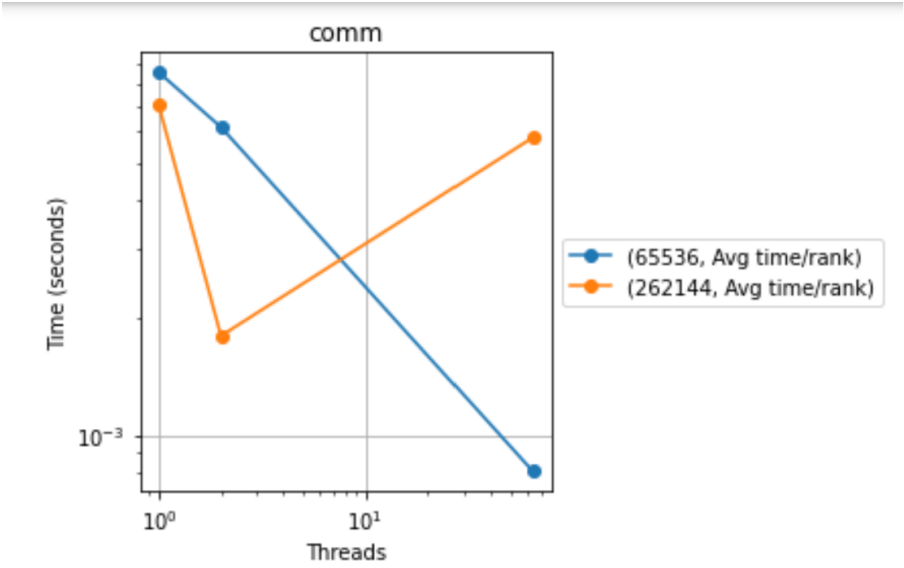
Strong scaling

Figure 3.10

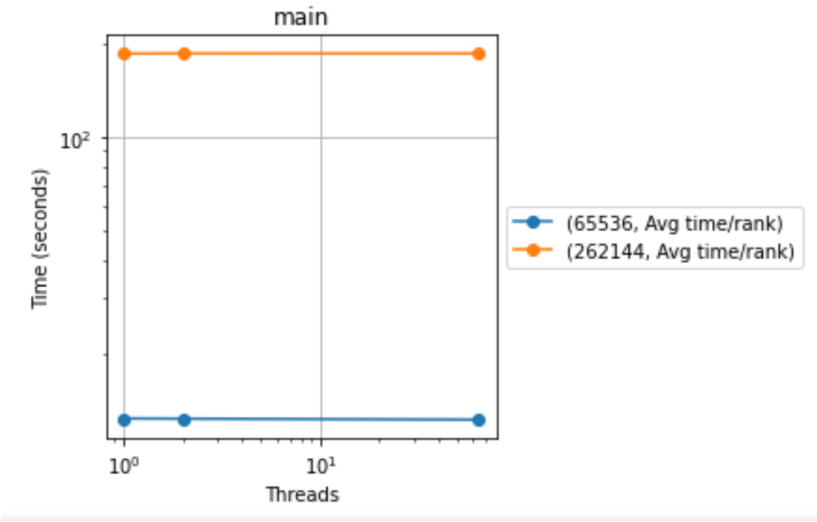


The MPI quicksort also appears to have two lines since I was unable to run larger input sizes and processes due to runtime. Both trends appear to be flat, when decreasing trends are expected.

Figure 3.11



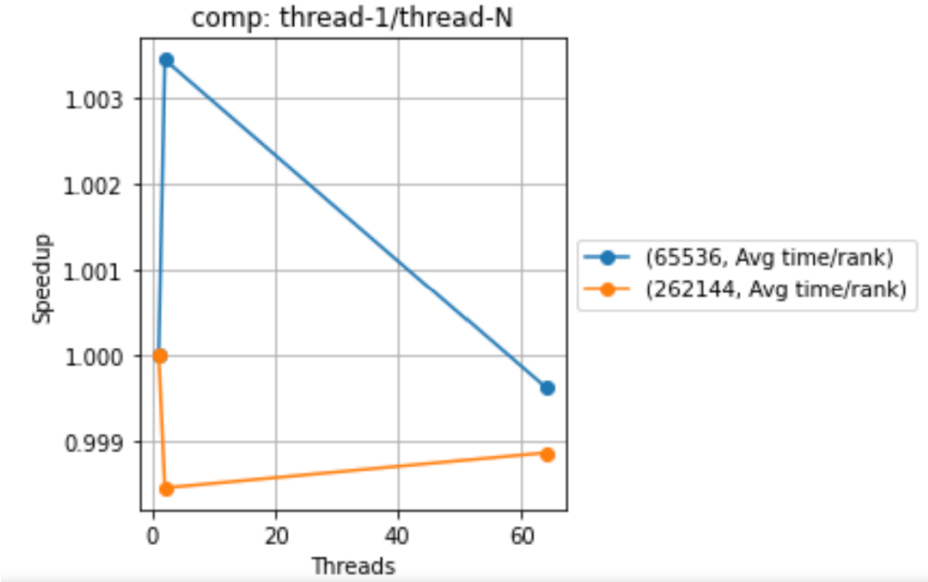
The line representing input size 262144 appears to increase when the general trend should be decreasing.



The MPI quicksort also appears to have two lines since I was unable to run larger input sizes and processes due to runtime. Both trends appear to be flat, when decreasing trends are expected.

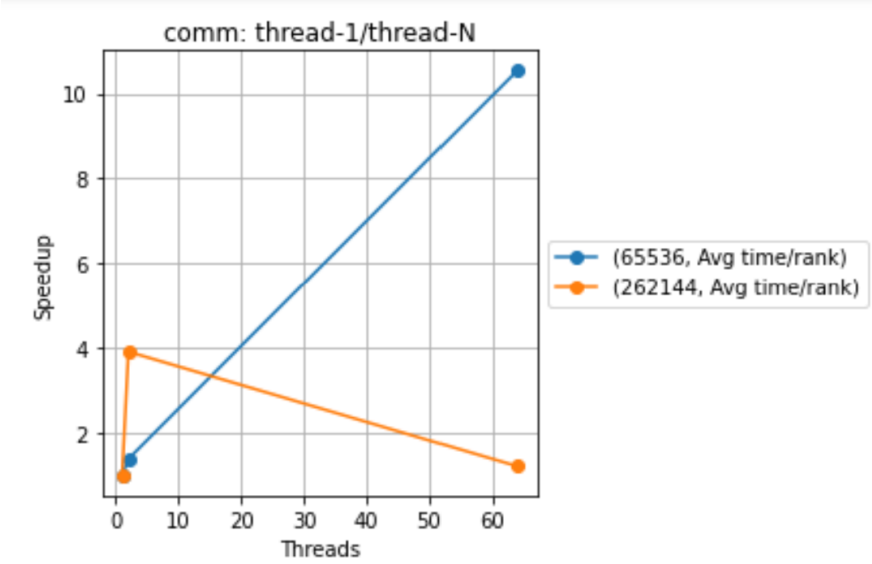
Speedup

Figure 3.13



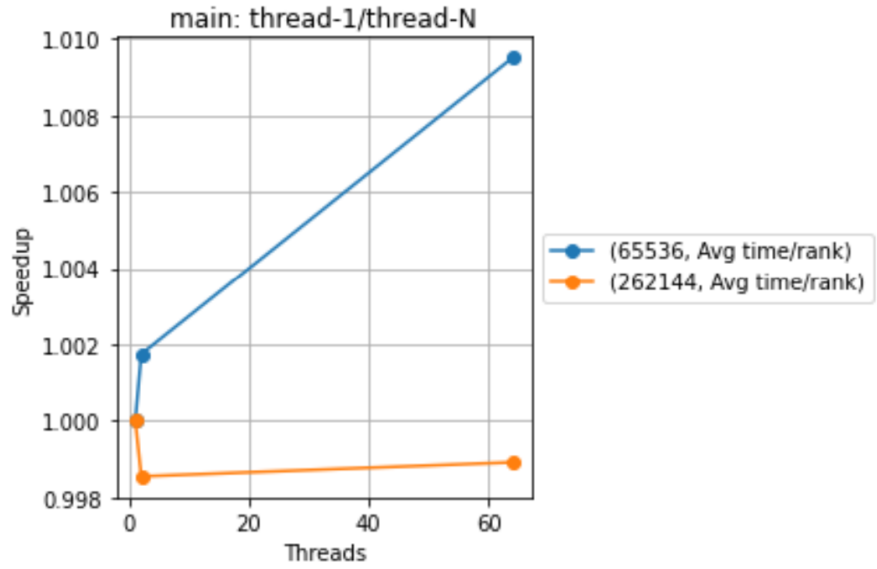
Speedup graphs are expected to have an increasing trend. This shows performance isn't efficient.

Figure 3.14



Speedup graphs are expected to have an increasing trend. Line representing input size 262144 doesn't represent this idea compared to 65536.

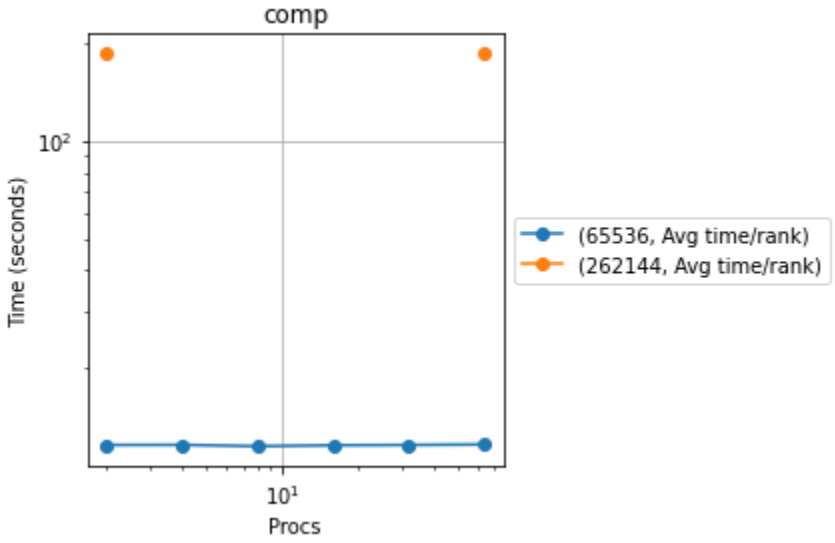
Figure 3.15



Speedup graphs are expected to have an increasing trend. Line representing input size 65536 doesn't represent this idea compared to 262144.

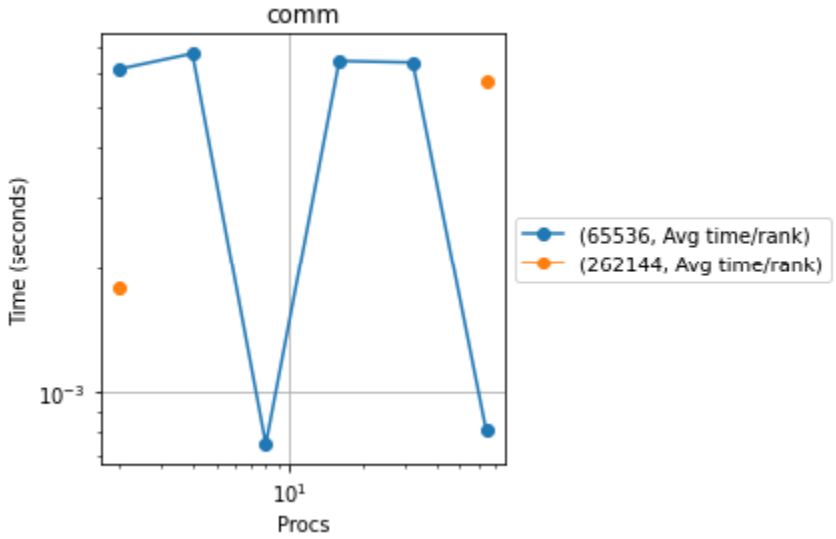
Weak scaling

Figure 3.16



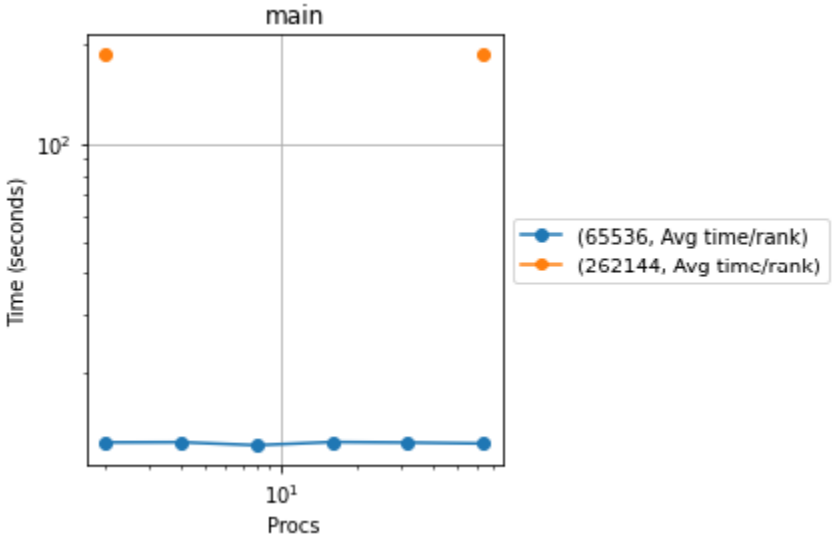
Ideally if the quicksort scales well, the weak scaling graph should have a constant trend. This shows that the algorithm scales weakly for the number of processors shown.

Figure 3.17



The line representing input size 65536 fluctuates and isn't expected. This suggests irregularity and inefficiencies in the algorithm. The line representing 26214 increases gradually, which shows more scaling compared to the 65536.

Figure 3.18

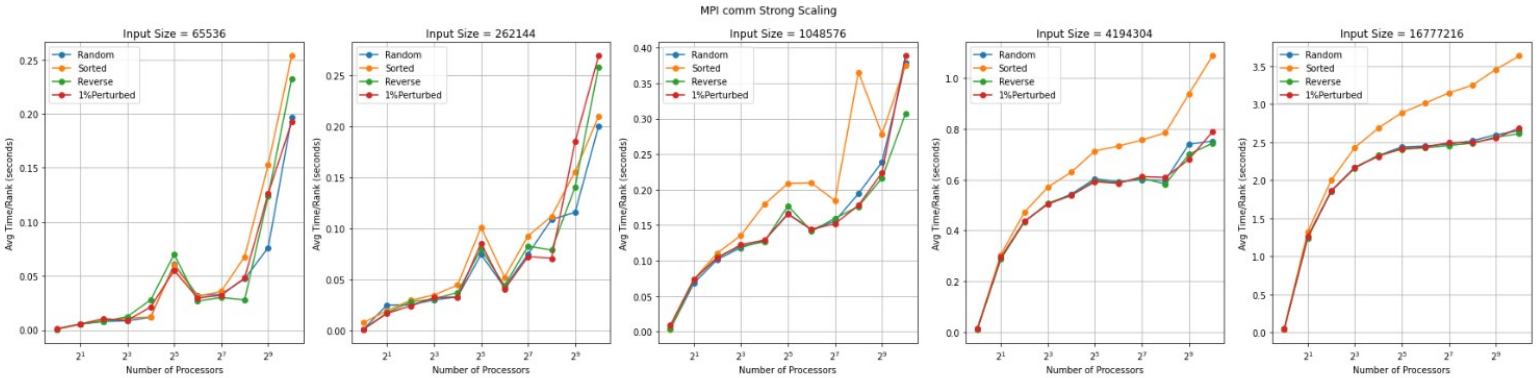


Ideally if the quicksort scales well, the weak scaling graph should have a constant trend. This shows that the algorithm scales weakly for the number of processors shown.

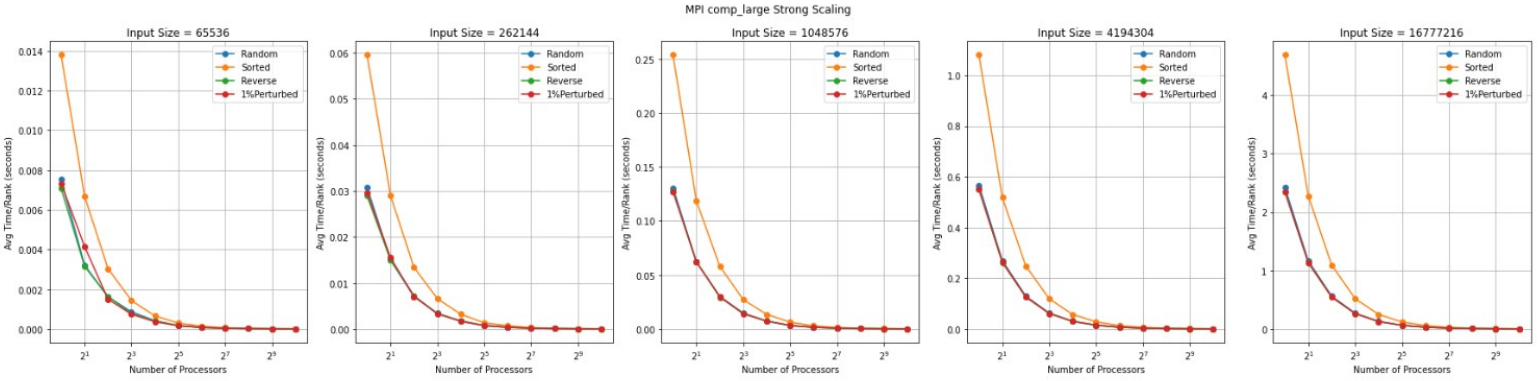
Merge Sort:

MPI

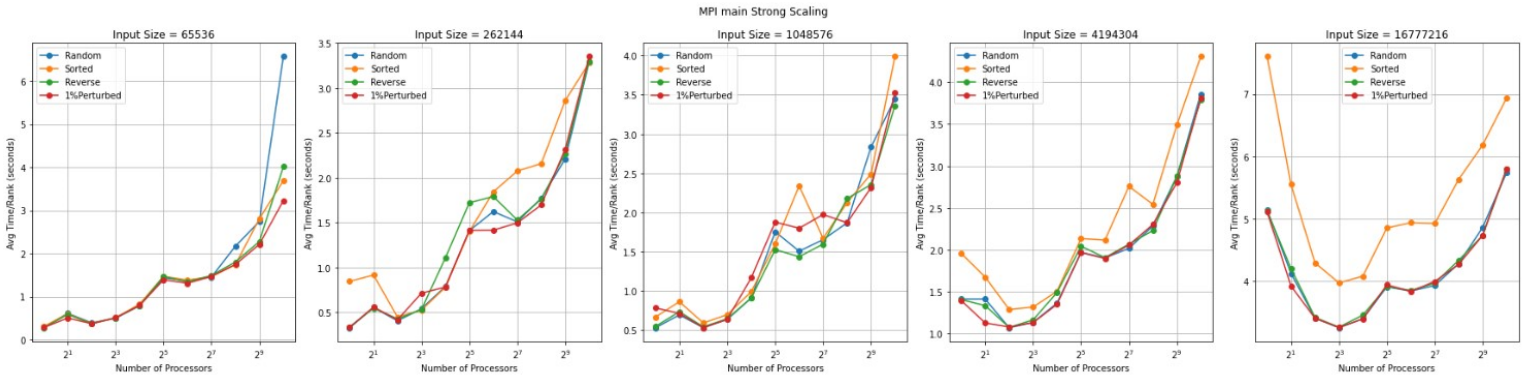
Strong Scaling



For all input sizes, the average time decreases as the number of processors increases, which is consistent with the expected behavior in strong scaling. However, there are noticeable spikes in time at certain points, particularly with 2^3 and 2^6 processors, which may suggest inefficiencies at those specific points. This could be due to communication overhead or uneven data distribution among processors. The times for sorted and reverse data tend to be close together, while the random data shows more variability. The 1% perturbed data demonstrates a middle ground between these behaviors.

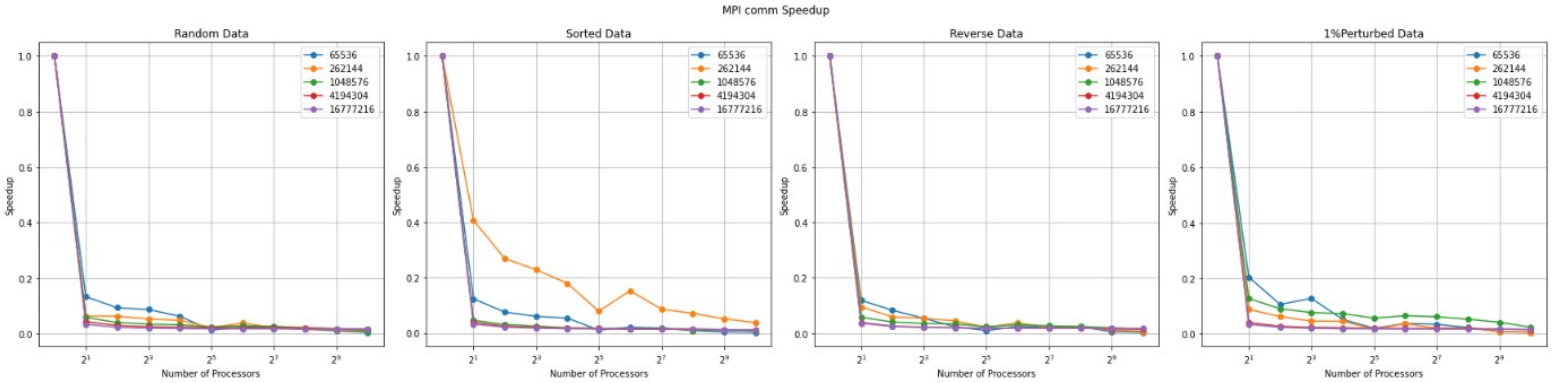


Similar to the MPI comm plots, the average time generally decreases as the number of processors increases. The time for random data seems to decrease more consistently compared to the other data types, indicating better scalability. For the largest input size, the average time decreases significantly as the number of processors goes up, which indicates good scaling for computation-intensive tasks with larger data sets.

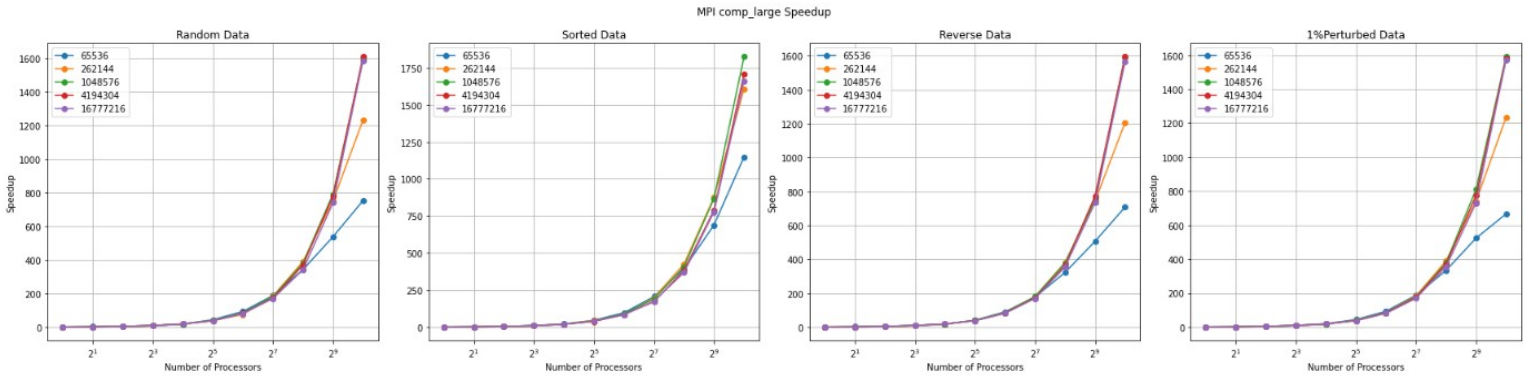


This plot shows more variability in the average times, especially for the sorted and reverse data types, which may point to specific stages in the sorting algorithm that do not parallelize as well as others. Notably, the average time for the 1% perturbed data shows significant increases at intermediate numbers of processors before decreasing again, which might indicate that the algorithm's performance on nearly sorted data can be unpredictable or sensitive to the number of processors.

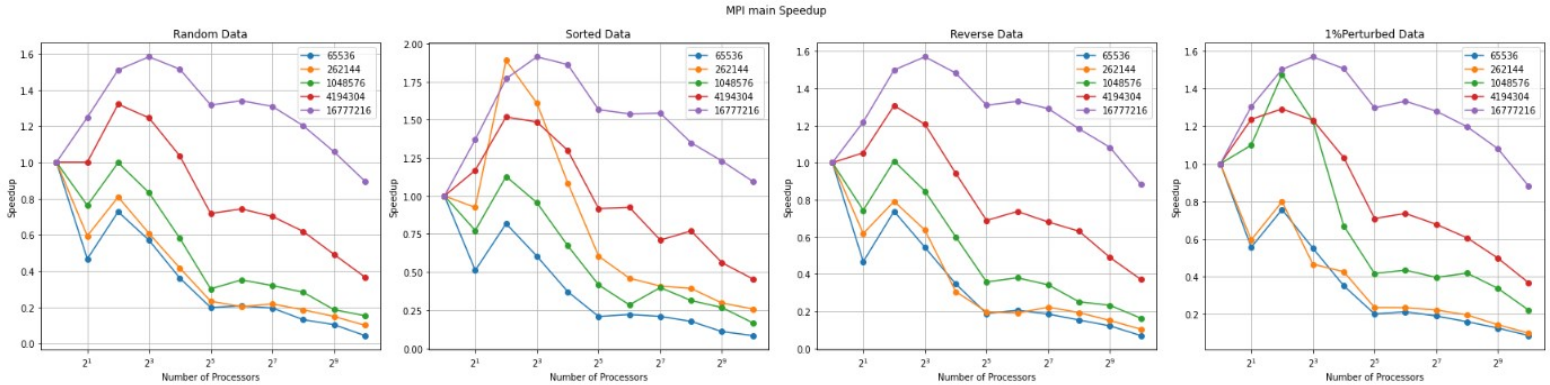
Speedup



For all input sizes, the speedup sharply decreases as the number of processors increases from 2^1 to 2^2, indicating a significant communication overhead when the number of processors is doubled. Beyond 2^2 processors, the speedup remains nearly constant across all input sizes, suggesting that the communication overhead dominates and that adding more processors does not yield additional benefits. The speedup is generally less than 1, which is unusual as speedup is expected to be greater than or equal to 1. This might suggest that the parallel algorithm is less efficient than the serial version, possibly due to high communication costs in the parallel environment.

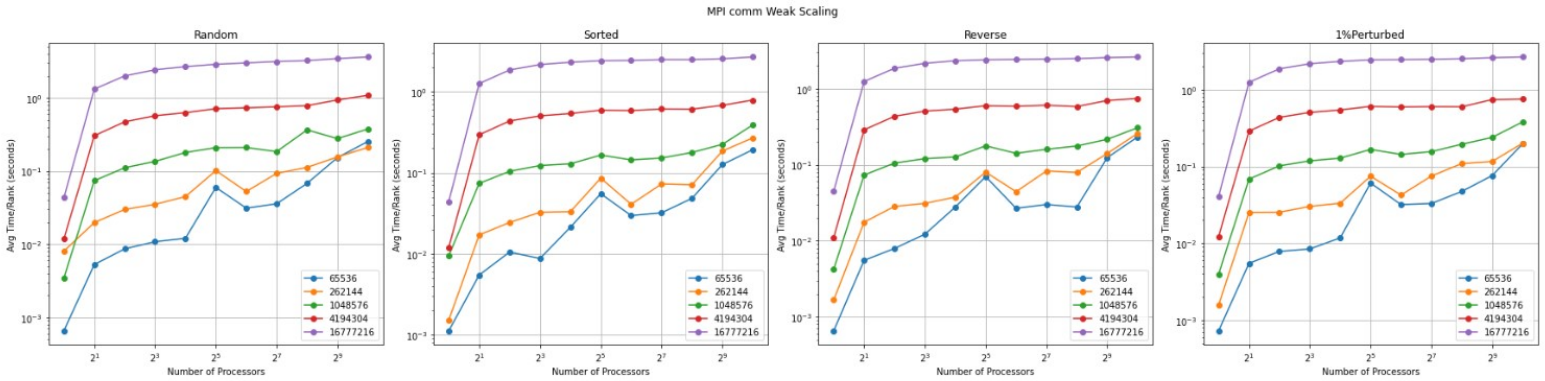


In this plot, the speedup increases significantly with the number of processors, which is the expected behavior in a speedup plot. There's a noticeable trend of diminishing returns as the number of processors increases, which is typical due to communication overhead and diminishing parallelization benefits. However, for the largest input size (purple line), the speedup continues to increase steeply, suggesting that this input size benefits more from parallel processing. The algorithm seems to be handling larger input sizes more efficiently, as indicated by higher speedups for larger data sets.

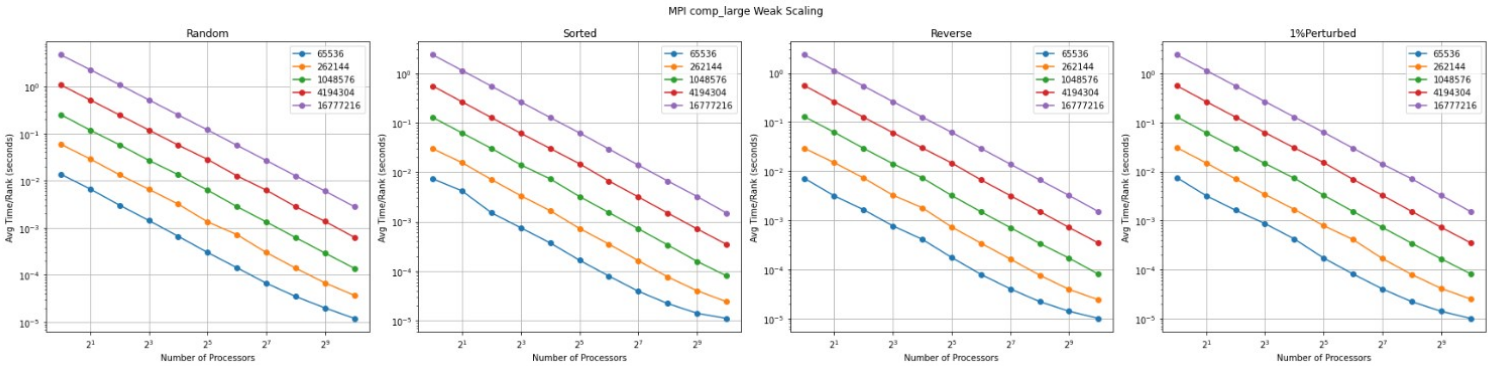


The speedup for random and reverse data distributions decreases after a certain number of processors (around 2^5), which indicates that the algorithm might be encountering bottlenecks such as communication overhead or synchronization issues. For sorted data, the speedup decreases much faster compared to random and reverse data, suggesting that the algorithm may not be efficiently parallelized for data that is already sorted. The 1% perturbed data speedup initially decreases and then flattens out, indicating that after a certain point, increasing the number of processors does not result in any significant performance gains.

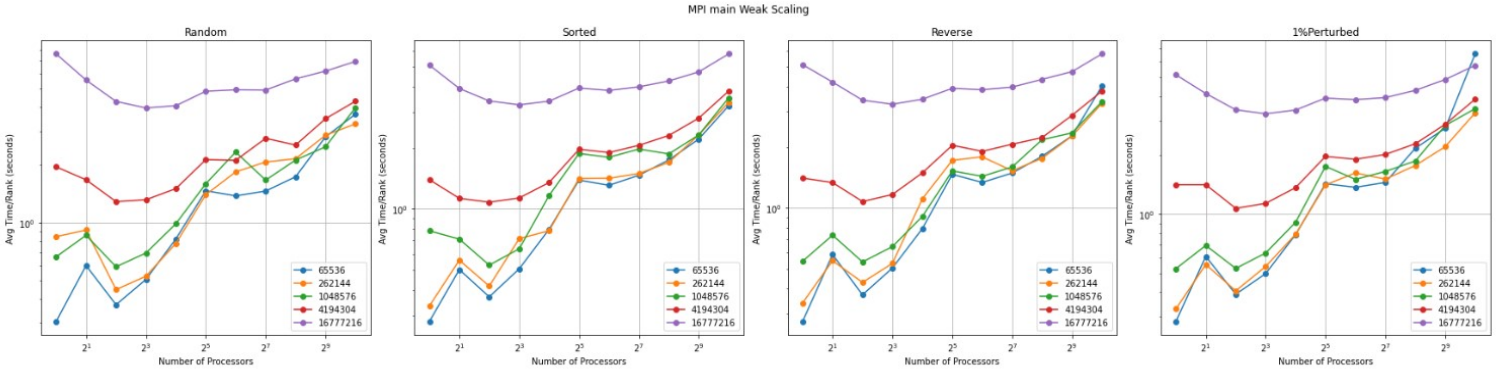
Weak Scaling



The average time increases for all data types as the number of processors increases, which is expected in weak scaling since the total workload is also increasing. The curves are not flat, which would be the ideal scenario for weak scaling, indicating that there is overhead or inefficiency in the system as it scales. For the random and 1% perturbed data, the time increases are more gradual compared to sorted and reverse data, suggesting that the algorithm handles increasing loads better for these distributions.



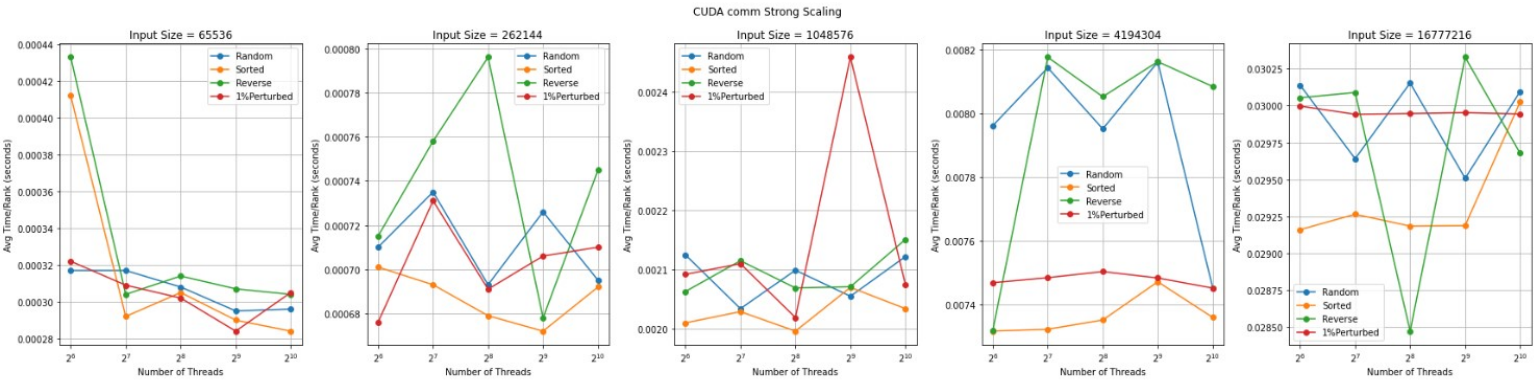
These plots generally show a decrease in time with more processors for smaller input sizes, but this trend reverses for larger input sizes. The fact that the average time decreases initially and then increases suggests there may be a sweet spot in terms of the number of processors for certain input sizes, after which the overhead outweighs the benefit of additional processors.



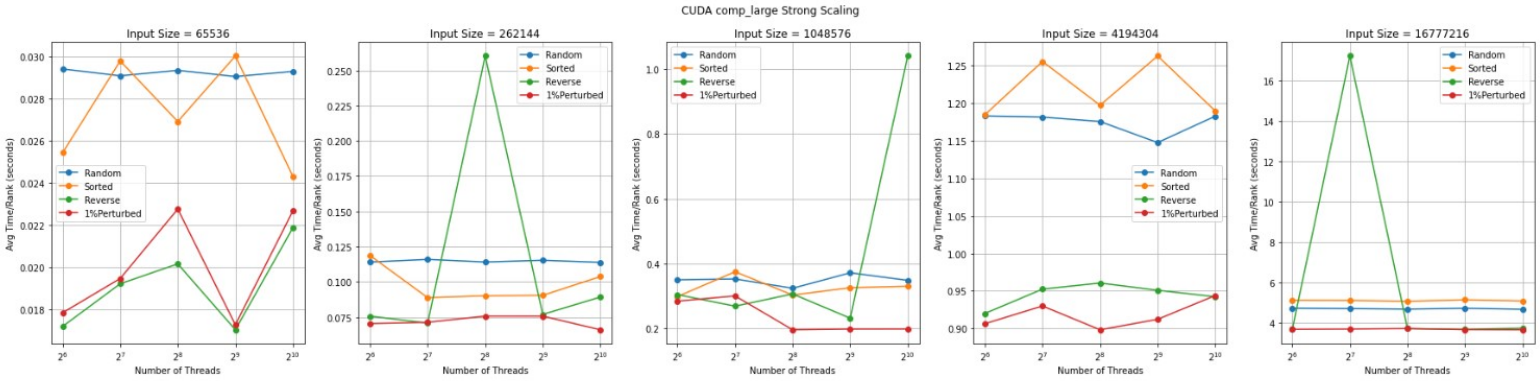
The average time across different data distributions shows a more complex behavior, with times increasing, decreasing, and then increasing again as the number of processors grows. This could indicate that the performance impact of scaling is non-linear and may be affected by factors such as memory bandwidth, cache effects, or communication overhead that do not scale linearly with the number of processors.

CUDA

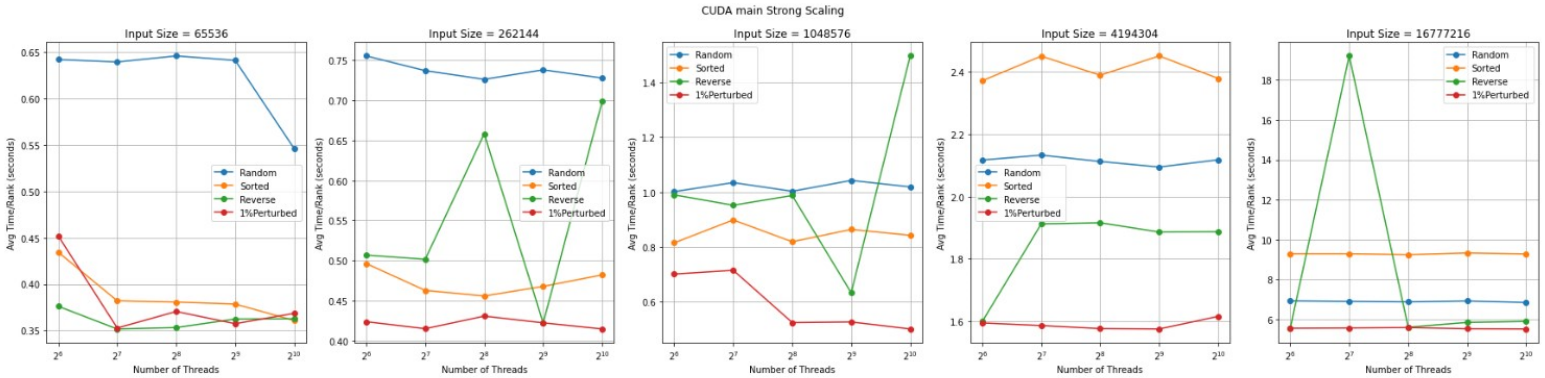
Strong Scaling



The graphs show the average time taken for different data types and input sizes as the number of threads increases. Ideally, we would expect the time to decrease as the number of threads increases, reflecting strong scaling. However, the graphs show fluctuations in the average time, which are not consistent with ideal strong scaling. This suggests that there are overheads or inefficiencies, particularly with specific data distributions and thread counts. There are sharp increases in time at certain points, for example, at 2^9 threads for the input size of 1048576. This could be due to factors such as memory access patterns or thread synchronization issues that are not scaling well.

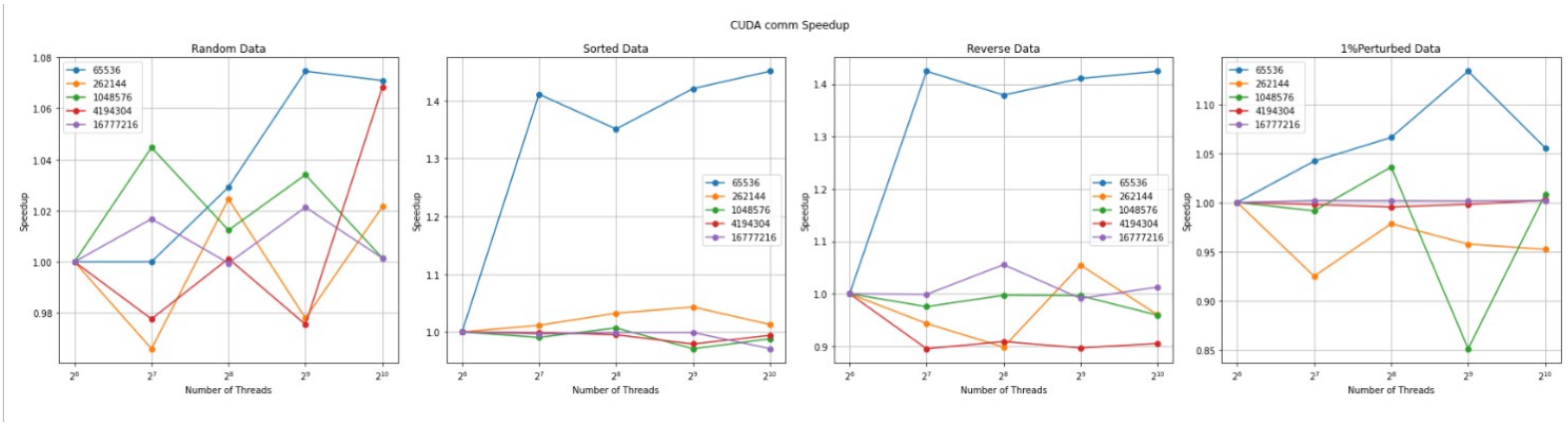


These graphs also show fluctuations and, in some cases, dramatic increases in average time with a higher number of threads, which is counterintuitive to strong scaling. The large spikes for certain thread counts and input sizes suggest that the algorithm or the hardware may be encountering a bottleneck. For some configurations, the time increases as the number of threads is doubled from 2^8 to 2^9, which is particularly noticeable for the input size of 16777216. This could indicate that the algorithm is not efficiently parallelized, or there might be contention for shared resources like global memory or thread contention.

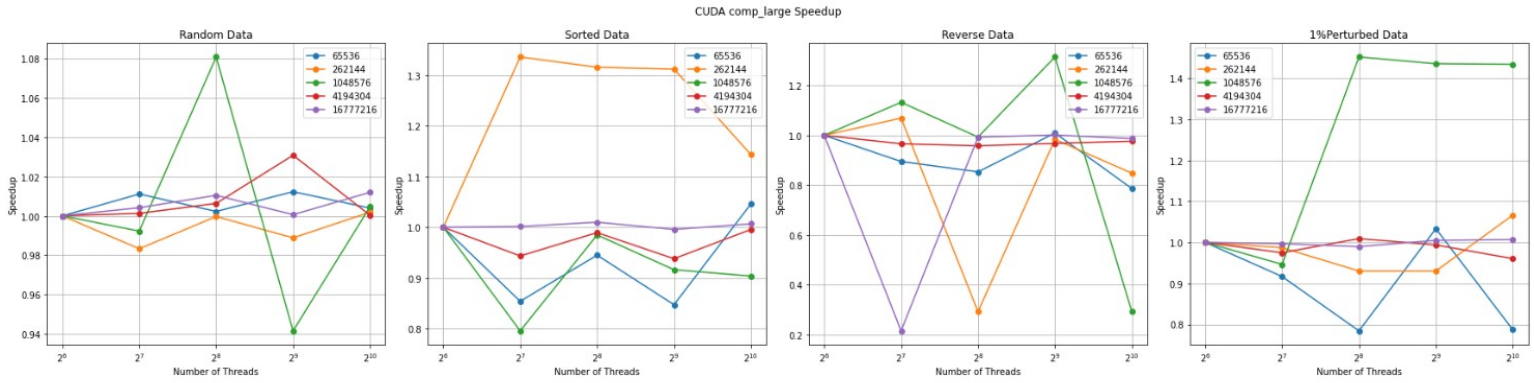


The time decreases for some thread counts but increases or fluctuates for others, which, again, does not align with the expected behavior for strong scaling. The graph for the input size of 16777216 shows a significant spike at 2^8 threads, highlighting a severe scaling issue. The variation in times between different data distributions suggests that the nature of the data significantly impacts the performance of the mergesort algorithm on the GPU.

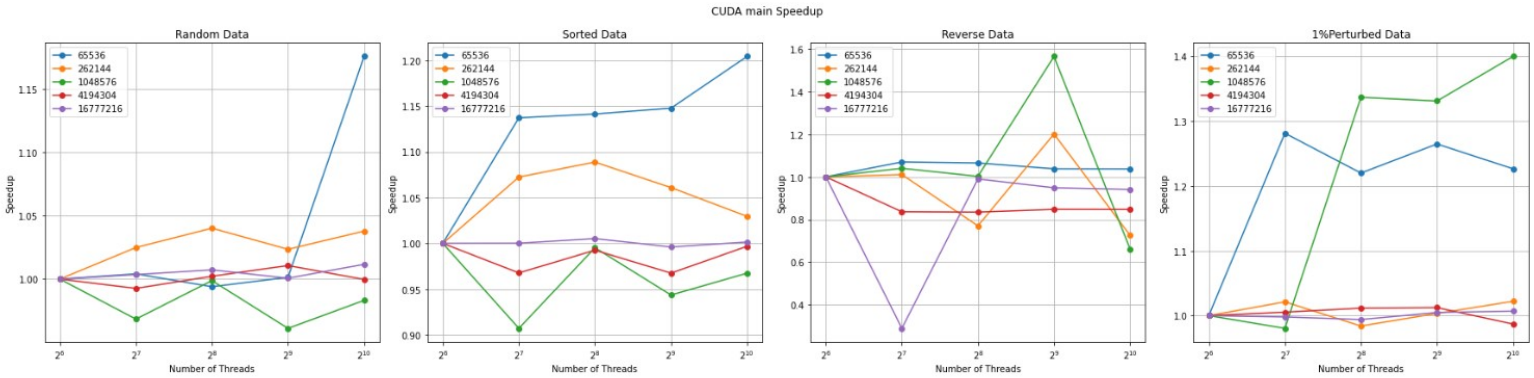
Speedup



The speedup for random data fluctuates significantly as the number of threads increases. This could indicate that the workload distribution across threads is not consistent, or that there are memory access patterns that are not optimal. For sorted data, there is an upward trend with more threads, which is unexpected because sorted data typically offers less room for speedup due to the nature of the mergesort algorithm. This may suggest that your CUDA implementation handles pre-sorted data more efficiently, possibly due to better memory coalescing or reduced branching. The reverse data speedup shows a peak at 2^7 threads before a sharp drop at 2^8 and then a rise again. This might be due to a specific threshold where the GPU's resources are either underutilized or overcommitted.The 1% perturbed data shows relatively stable speedup across thread counts, with a slight dip at 2^8 threads. This may indicate that nearly sorted data has a speedup behavior that is not heavily affected by the number of threads used.

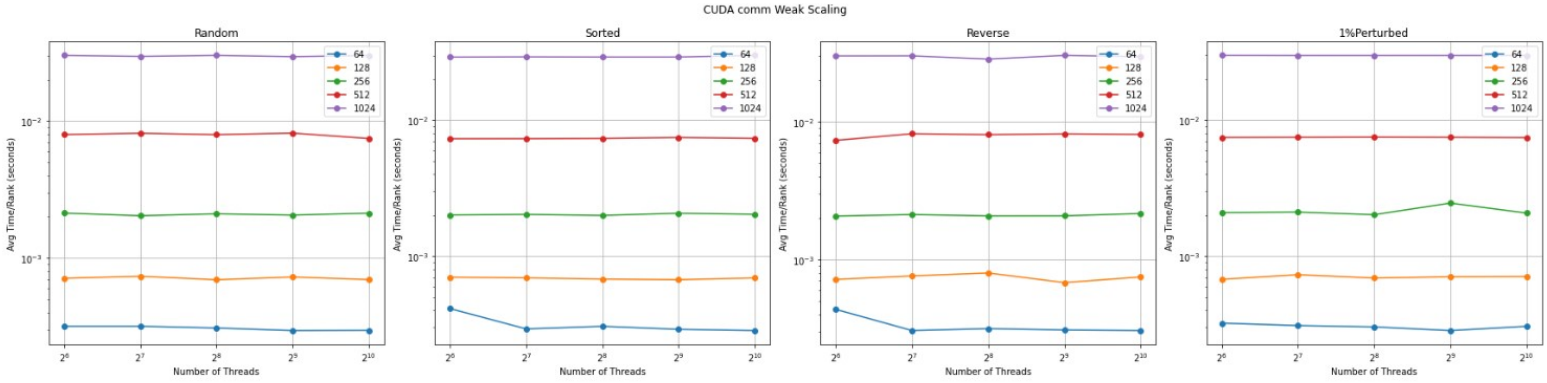


The speedup for random data shows a general decrease as the number of threads increases, which is contrary to expected behavior. This suggests that the algorithm may not be efficiently parallelized, or there could be high contention for shared resources. The sorted data speedup shows significant variation, with particular configurations (like with 2^9 threads) offering higher speedup, possibly due to specific computational phases benefiting from additional threads. For reverse data, the speedup drops dramatically at 2^8 threads. This indicates a potential inefficiency in the algorithm when dealing with data that is in the worst-case order for sorting. The 1% perturbed data again shows a dip at 2^8 threads, suggesting that this is a critical point where resource contention or algorithmic inefficiency becomes pronounced.

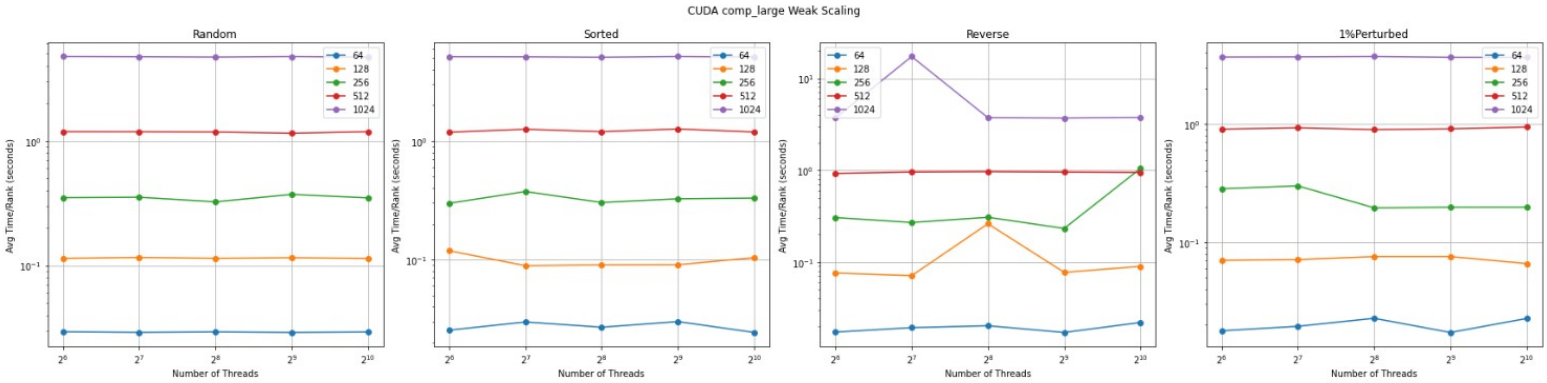


Speedup variability is high across different data distributions and thread counts, suggesting that the performance of the CUDA mergesort is highly dependent on the nature of the data and the specific GPU configuration. There are particular points (notably at 2^8 threads) where speedup either peaks or troughs, which could be indicative of resource contention or optimal utilization depending on the input size.
The algorithm's performance on nearly sorted data shows potential for improvement, as indicated by the high speedup at 2^10 threads for one input size.

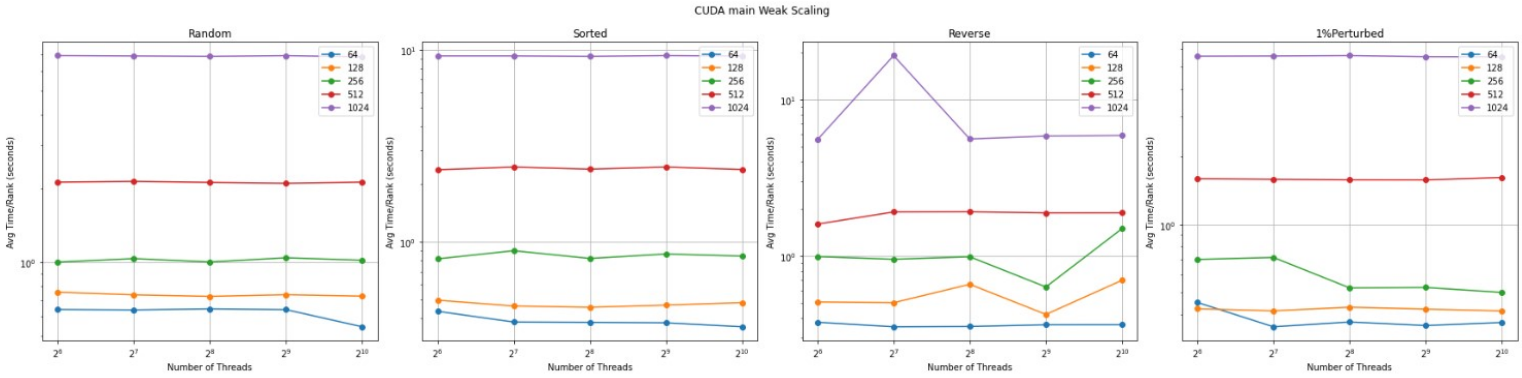
Weak Scaling



The average time remains relatively stable across different thread counts for each input size, which is the expected behavior in weak scaling. However, there are some fluctuations, especially with smaller input sizes, which may suggest some inefficiencies. Generally, the lines are flat, which indicates that the algorithm scales well with the increase in the number of threads when the workload per thread is kept constant. This is good as it suggests that the overhead of adding more threads does not significantly increase with the problem size.



For this set of graphs, there are more noticeable fluctuations in time as the number of threads increases. This could be due to a variety of factors such as memory contention, the inefficiency of memory transfers, or the algorithm's scaling limitations. In several cases, the average time slightly increases with the number of threads, which may indicate that the scaling is not perfect and there might be a bottleneck occurring at larger thread counts.



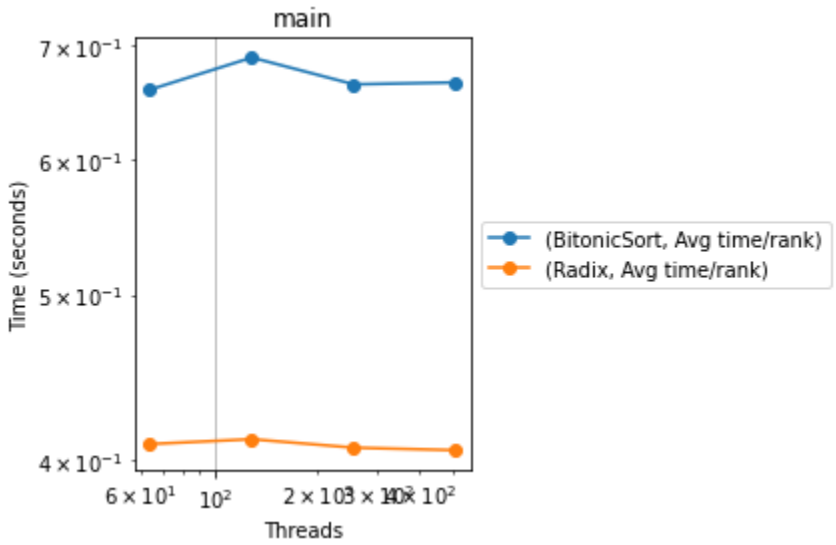
The average time for the main weak scaling also shows relatively flat lines, indicating good weak scaling behavior. However, there are some notable deviations from the expected flat line, especially with reverse data, which may indicate that the workload is not perfectly balanced or that there is some other form of resource contention.The 1% perturbed data shows the least fluctuation, indicating that the application may be more robust to nearly sorted data when scaling across threads.

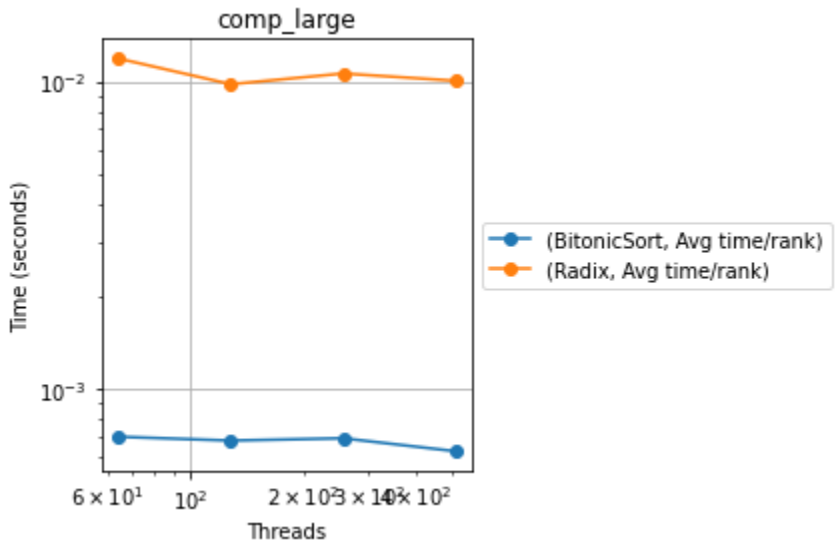
Bitonic vs. Radix:

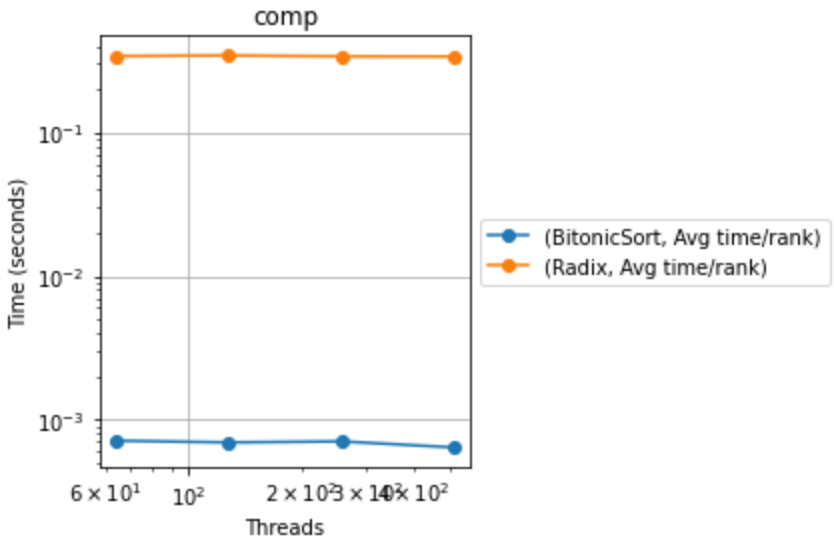
Cuda:

Num_threads = 64, 128, 256, 512

ArraySize = 65536

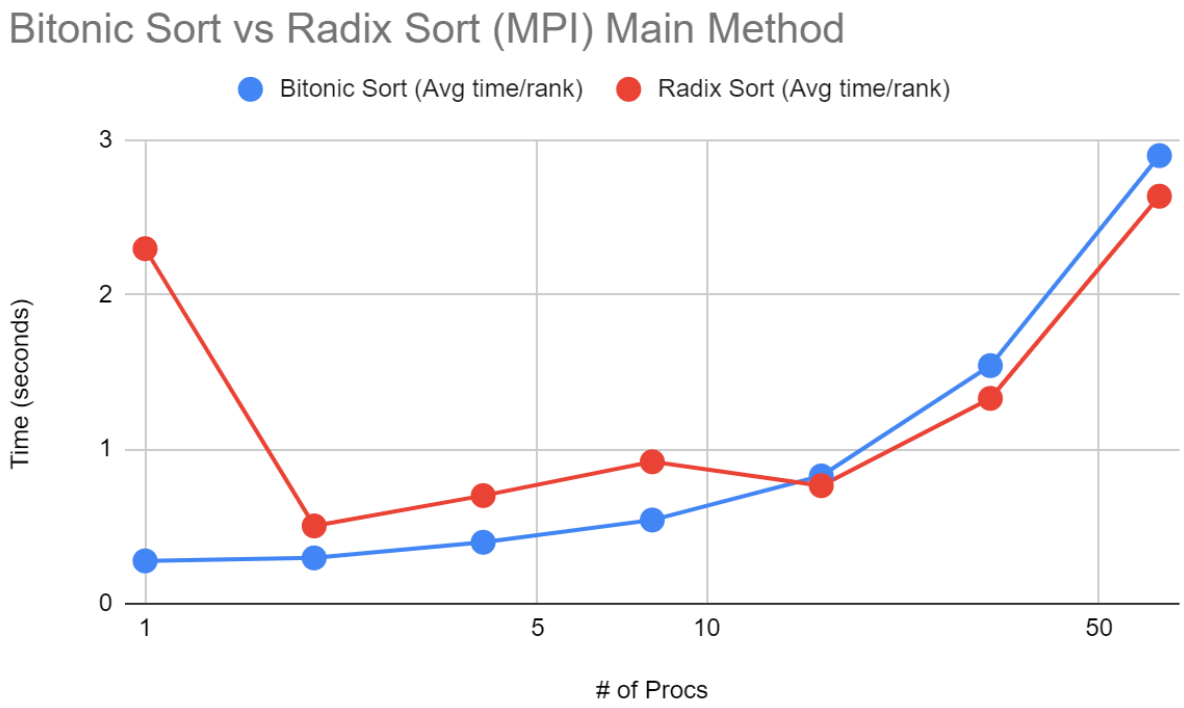


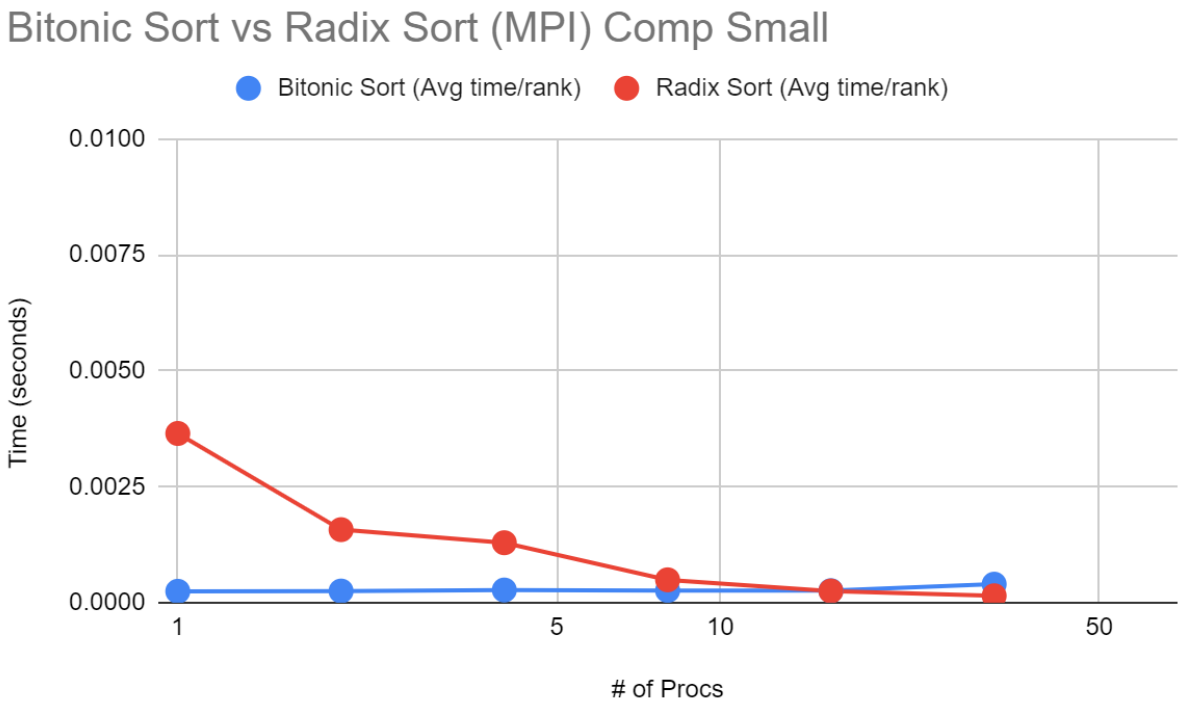




MPI:

Num_procs = 1, 2, 4, 8, 16, 32, 64
ArraySize = 65536

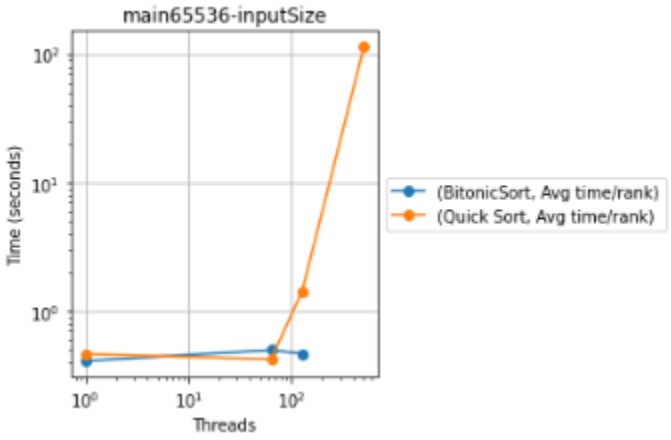


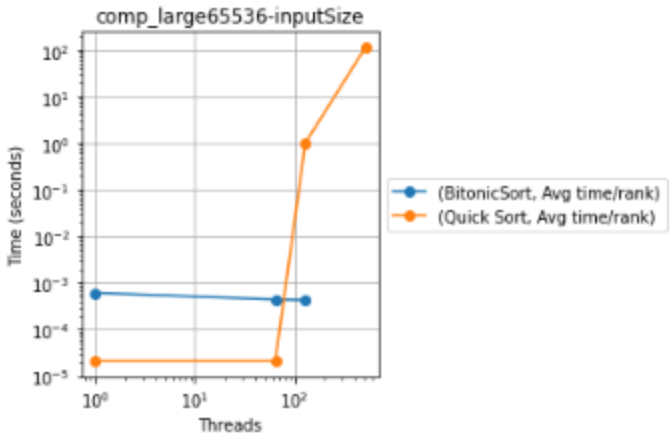


Bitonic vs. Quick Sort:

Cuda:

Num_threads = 64, 128, 256, 512
ArraySize = 65536





Conclusion:

Overall, when observing the findings of all of the algorithms, it is clear that the Radix and Bitonic sort implementations both in theory and in practice benefit from parallelization. The inherently recursive and divide-and-conquer methodologies that underpin quicksort and merge sort certainly render them viable for parallel computation. Nevertheless, actualizing their parallel potential is subject to a variety of factors. These include the complexity of managing recursive operations, the synchronization and communication costs inherent in the merging of sorted segments, and the potential for imbalanced processing loads which can undermine efficiency.

Quicksort's effectiveness in a parallel setting is particularly sensitive to how data is partitioned during recursive calls. Achieving an equitable distribution of data is paramount for optimizing processor workload and maintaining high performance. Conversely, merge sort's primary challenge lies within its merge phase, necessitating a high degree of coordination and, in distributed environments, significant inter-process communication.

Radix and Bitonic sorts, in contrast, are more naturally aligned with parallel processing paradigms. Radix sort excels by circumventing element comparisons entirely, opting instead for a consistent workload distribution guided by individual digit sorting—a process that is notably conducive to parallel execution and particularly well-suited to GPU architectures. Bitonic sort is distinguished by its predetermined sequence of element comparisons and exchanges, facilitating parallel execution without the intricacies of managing data dependencies.

As a result, while parallel implementations of quicksort and merge sort can indeed yield performance gains—most notably within multi-core CPUs employing shared memory—their scalability and setup complexities can be more pronounced, particularly in distributed systems where processors abound and communication is non-trivial. Conversely, Radix and Bitonic sorts demonstrate a more seamless transition to parallel contexts, showcasing enhanced scalability and a propensity for higher performance on systems inherently designed for extensive parallelism.

Upon evaluating the CUDA implementations of Radix and Bitonic sorts, it becomes evident that Bitonic sort displays a distinct advantage with respect to execution time. This efficiency in run time suggests that Bitonic sort not only outperforms Radix sort in a CUDA environment but also surpasses the other algorithms under consideration. Consequently, when it comes to sorting within the parallelized, GPU-accelerated domain, Bitonic sort emerges as the preeminent choice, offering superior speed and overall performance.