CSCE 435 Group project

1. Group members:
1. Emma Ong
2. Naimur Rahman
3. Anna Huang
4. Luis Martinez Morales

We will be communicating through Discord and text messaging.

2. _due 10/25_ Project topic: Comparative Analysis of Sorting Algorithms Using MPI and CUDA

3. _due 10/25_ Brief project description (what algorithms will you be comparing and on what architectures)

Our project aims to compare the performance of various parallel sorting algorithms using two different parallel computing technologies: Message Passing Interface (MPI) and Compute Unified Device Architecture (CUDA). We will be comparing Merge Sort, Quick Sort, Radix Sort, & Bitonic Sort. For Merge Sort and Quick Sort we will be comparing the MPI implementation versus CUDA implementation. For Radix Sort we will be comparing the MPI implementation to the CUDA implementation. For Bitonic Sort we will be comparing the implementation to the CUDA implementation.

Merge Sort (MPI) **
1. Generate and populate an array with random numbers.
2. Initialize MPI for parallel processing.
3. Divide the array into equal-sized chunks based on the number of available processes.
4. Distribute these subarrays to different processes.
5. On each process, perform a local merge sort on its subarray.
6. Gather the sorted subarrays into one central location, typically on the root process.
7. On the root process:
   a. Perform a final merge sort on the gathered subarrays to create a fully sorted array.
   b. Display the sorted array.
8. Clean up resources, including memory used for subarrays and temporary arrays.
9. Finalize MPI to end parallel processing.

We'll be leveraging sources for implementing Merge Sort with MPI like:
https://www.geeksforgeeks.org/merge-sort/ ,
https://curc.readthedocs.io/en/latest/programming/MPI-C.html

Merge Sort (MPI) **

Merge Sort (CUDA) **
1. Allocate GPU Memory: Reserve memory on the graphics processing unit (GPU) for the input list.

2. Copy to GPU: Transfer the original list from the CPU to the allocated GPU memory.

3. Determine Configuration: Decide how many threads to use in each GPU block and how many blocks based on the input list size and GPU capabilities.

4. Launch Merge Sort Kernel: Start the GPU process by running a CUDA kernel designed for parallel merge sorting.

5. Copy to CPU: Retrieve the sorted list from the GPU memory and place it back into the CPU memory.

6. Free GPU Memory: Release the GPU memory that was allocated for the input list.

7. Main Function:
a. Read or generate the input list.
b. Call the "ParallelMergeSort" function.
c. Print the sorted list to the console.

We'll be leveraging sources for implementing Merge Sort with CUDA like:
https://www.geeksforgeeks.org/merge-sort/ ,
https://cuda-tutorial.readthedocs.io/en/latest/tutorials/tutorial01/

Radix Sort (CUDA): **********

```
function RadixSort(data):
    // First determine the number of elements
    numElements = length(data)

    // Allocate GPU memory for data and temp storage
    d_data = allocateGPUArray(numElements)

    d_temp = allocateGPUArray(numElements)

    // We must then copy the data from the CPU to GPU
    copyDataToGPU(data, d_data)

    // Determine the maximum number of bits needed for the data
    numBits = calculateMaxNumBits(data)

    // Perform radix sort using CUDA
    RadixSort(d_data, d_temp, numElements, numBits)

    // Copy the sorted data back from GPU to CPU
    sortedData = copyDataToCPU(d_data, numElements)

    // Free GPU memory
    freeGPUArray(d_data)
    freeGPUArray(d_temp)

    return sortedData
end function
```

For RadixSortCUDA function we plan to implement Radix Sort much like this reference from Geeks2Geeks https://www.geeksforgeeks.org/radix-sort/#:

```
function RadixSort(data, numElements):
    // Find the maximum value in the data to determine the number of digits
    maxVal = findMaxValue(data)
    numDigits = countDigits(maxVal)

    // Initialize a count array to hold the count of digits (0-9)
    count = new int[10]

    // Initialize an output array to store the sorted data
    output = new int[numElements]

    // Perform counting sort for each digit, from the least significant to the most significant
    for digitPosition from 1 to numDigits:
        // Reset the count array
        for i from 0 to 9:
            count[i] = 0

        // Count the occurrences of each digit in data
        for i from 0 to numElements:
            digit = getDigit(data[i], digitPosition)
            count[digit]++

        // Update the count array to contain the actual position of each digit in the output
        for i from 1 to 9:
            count[i] += count[i - 1]

        // Build the output array using the count array
        for i from numElements - 1 down to 0:
            digit = getDigit(data[i], digitPosition)
            output[count[digit] - 1] = data[i]
            count[digit]--

        // Copy the output array back to the data array
        for i from 0 to numElements:
            data[i] = output[i]

    // Free memory
    free(count)
    free(output)
end function
```

Radix Sort (MPI): **

Initialize MPI environment.

Each process receives a chunk of the unsorted array.

In parallel, each process performs Radix Sort on its local chunk.

For each digit (starting from the least significant digit):
    Each process counts the occurrences of each digit within its chunk.
    All processes participate in a collective communication to determine the global count for each digit.
    Each process determines the target process for each element based on the global digit counts and sends the elements to the respective processes.

After all digits have been processed, each process has a portion of the globally sorted array.

If necessary, merge the sorted chunks from each process to form the fully sorted array.

Finalize MPI environment.

****

Quicksort (MPI)
https://www.geeksforgeeks.org/implementation-of-quick-sort-using-mpi-omp-and-posix-thread/#

```
function quicksort(arr, start, end)
// declare pivot and index
// consider base case if arr has only one element
// swap pivot with first element (pivot is middle element)
// set index to start of the arr
// loop from start+1 to start+end
    // swap if arr[i] < pivot
// swap pivot into element at index
// recursively call sorting function from both sides of the list
```

Quicksort (CUDA)

```
// define partitioning function
// __device__ int partition(int* arr, int left, int right)
// choose right element as pivot
// calculate index of left element
// loop from left to right j=left->right
    // swap arr[i] and arr[j] if arr[j] < pivot
// swap pivot element with element at i+1
// return index of pivot element


// define cudaQuicksort function
// void cudaQuicksort(int* arr, int size)
// declare device array
// launch quicksort kernel on device array
// launch quicksort kernel on device
// cudaMemcpy(arr, device array,..,cudaMemcyDeviceToHost)
// cudaFree(device array)
```

```
*****************************************************************************************************
Bitonic Sort (MPI)
function bitonicSort(up, sequence)
  if length(sequence) > 1 then
    firstHalf = first half of sequence
    secondHalf = second half of sequence

    bitonicSort(true, firstHalf)  // sort in ascending order
    bitonicSort(false, secondHalf) // sort in descending order

    bitonicMerge(up, sequence) // merge whole sequence in ascending or descending order

function bitonicMerge(up, sequence)
  if length(sequence) > 1 then
    // bitonic split
    compareAndSwap(up, sequence)

    firstHalf = first half of sequence
    secondHalf = second half of sequence

    bitonicMerge(up, firstHalf)
    bitonicMerge(up, secondHalf)

function compareAndSwap(up, sequence)
  distance = length(sequence) / 2
  for i = 0 to distance - 1 do
    if (up and sequence[i] > sequence[i + distance]) or (!up and sequence[i] < sequence[i +
distance]) then
      swap(sequence[i], sequence[i + distance])

// To sort a sequence in ascending order using bitonic sort:
bitonicSort(true, sequence)

*****************************************************************************************************
Bitonic Sort (CUDA)

Set number of threads, blocks, and number of values to sort

Allocate memory for values on the host
Fill the host memory with random floating-point numbers

Allocate memory on the device (GPU) for sorting

Copy the unsorted values from host to device memory

For each stage of the bitonic sequence:
  For each step of the current stage:
    Launch the bitonic sort kernel with the current step and stage parameters
    The kernel will compare and swap elements to achieve the bitonic sequence

Wait for GPU to finish sorting
```

Copy the sorted values from device back to host memory

Free the device memory

Print the sorted values (if needed)

2c. Evaluation plan - what and how you will measure and compare

Regarding input types, one will be an integer value for the size of the array that holds the values to be sorted and the other is the thread count used for the algorithm. Array sizes will be {16, 64, 128, 256, 1024, 2048, 4096} and thread sizes will be {2, 4, 16, 32, 64, 128, 256, 512}. Each array size will be tested with every thread size. All arrays will be filled with integer values as radix sort is only possible with integers and not floating point numbers. The values for arrays will be randomly generated within the program depending on problem size.

For strong scaling, each problem size will be tested with the aforementioned increasing amount of thread sizes. For weak scaling, all thread counts will be tested with increasing array problem sizes for sorting.

Overall, we will be testing and comparing overall run times with all four algorithms and their subset types. The run times will be compared based on the factors of thread count, algorithm, and problem size.

3c. Project Implementation

Originally, our plan involved identifying and utilizing sources to implement our algorithms, followed by compiling and executing our code on the Grace platform. We had meticulously conducted our research and prepared pseudocode well in advance. However, as soon as we transitioned from pseudocode to actual code that we could run, Grace underwent an extended maintenance period, preventing us from testing our code and generating calibration files.

Hypothesis:

We estimate that quicksort would have a lower runtime than merge, radix, and bitonic sort because of their overall scaled predicted run times. When comparing, the runtime for parallel implementation of Quick sort (theoretical best) is O(logn/p) where p is the number of parallel processors, but the worst case is O(n^2). The runtime for parallel implementation of Merge sort using cuda is O(nlog(n/p)) where p is the number of processors with a good implementation of parallelized merge sort.

The time complexity of Radix Sort is O(w*n) for the serial/sequential version, where n is the number of elements needed to be sorted and w is the number of bits required to store each key. Radix Sort works by processing each bit of the numbers to be sorted, which leads to its linear

time complexity in the number of bits processed. However, in a parallel CUDA implementation, Radix Sort's time complexity can be significantly reduced by distributing the counting and prefix sum computation across the multiple cores of the GPU. Theoretically, if you have as many processing units as items to be sorted, the time complexity could approach O(w), since you could potentially process each bit of all items simultaneously.

The Bitonic Sort algorithm has an O(logn) time complexity. However when it comes to the parallelized algorithm implementation with CUDA theoretically should be O(log (n/p)) where p is the number of processors.

In conclusion, based on these analytical time complexities when parallelized, it is clear to see why quicksort would be the most optimal. Parallelizing these algorithms as a whole will greatly reduce run times of the algorithms.

4. Performance evaluation

Include detailed analysis of computation performance, communication performance.

Include figures and explanation of your analysis.

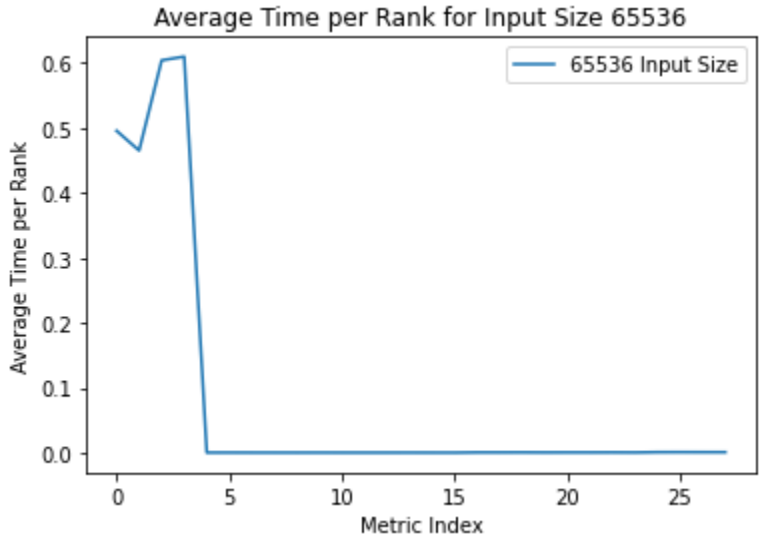
We observed a notable trend in our analysis: as the number of processes increased, there was a corresponding decrease in performance time. This effect was particularly pronounced at lower process counts, where the runtime remained significantly high. This phenomenon can be attributed to the limited impact of parallelization at a smaller scale, where the overhead may not be justified by the performance gains.

Our primary objective was to evaluate the impact of escalating the number of processes or threads on the average processing time of parallelized sorting algorithms. To maintain consistency in our analysis, we ensured that each algorithm was tested with arrays of identical input sizes.

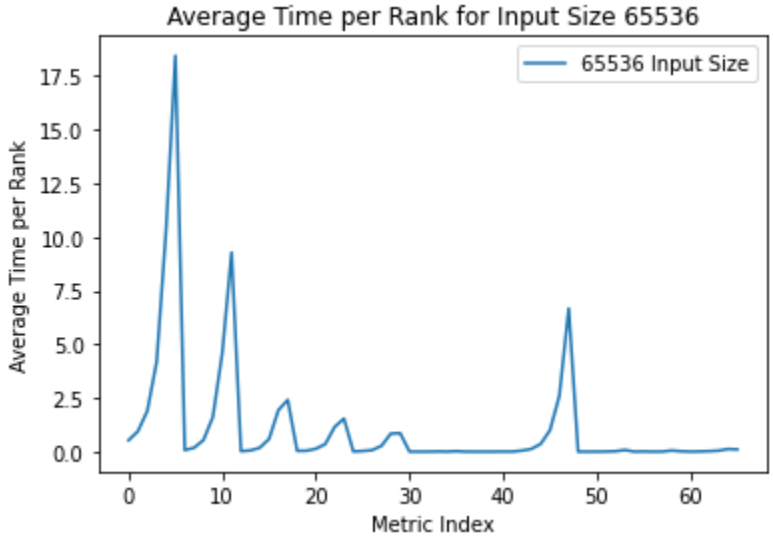
Currently, our testing has been confined to arrays with randomly generated values. However, to gain a more comprehensive understanding of performance trends, future tests should include arrays with already sorted and reverse-ordered data. The randomness of our current data is influenced by specific seeds, reinforcing the notion that exploring different data types could yield further insights. It’s also possible that certain algorithms may demonstrate enhanced efficiency with specific types of datasets."

This revised version presents the same information in a more structured and formal tone, enhancing clarity and emphasizing key points for better understanding.

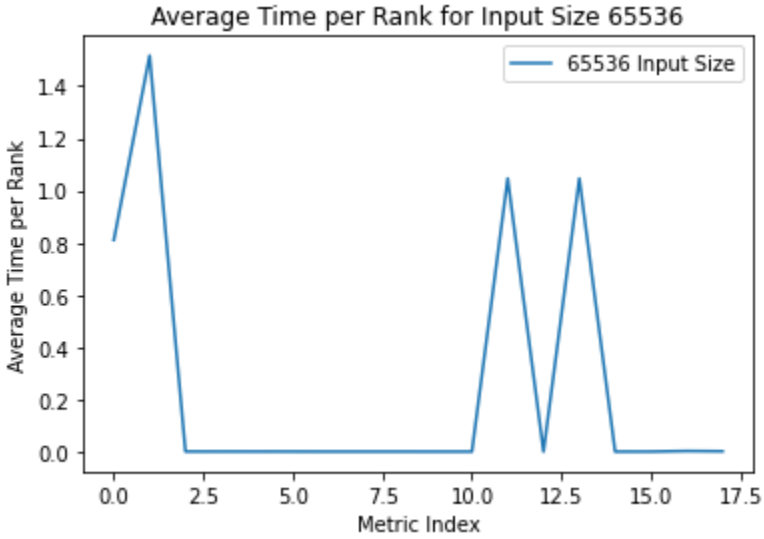
Bitonic Sort Cuda:



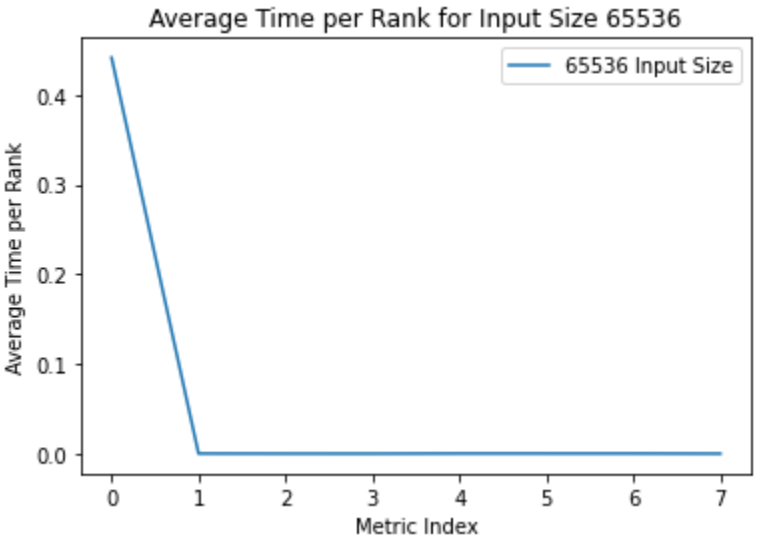
Bitonic Sort MPI:



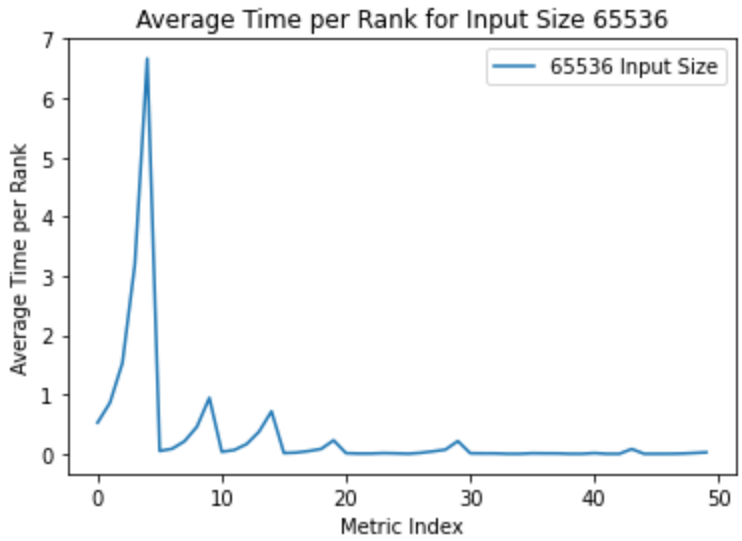
Quick Sort Cuda:



Merge Sort CUDA:



Merge Sort MPI:



4a. Vary the following parameters

For inputSizes:

- 2^16, 2^18, 2^20, 2^22, 2^24, 2^26, 2^28

We opted for an input size of 2^16 for our algorithms, primarily due to performance constraints encountered on the Grace Cluster, which included significant slowdowns and frequent node terminations. Notably, for our implementation of Quicksort using MPI, we managed to extend the input size up to 2^22, showcasing its relative efficiency under these conditions.

For inputTypes:

- Sorted, Random, Reverse sorted, 1%perturbed

Bitonic Sort:

- InputType: Sorted
- MPI num_procs: 2, 4, 8, 16, 32, 64
- CUDA num_threads: 64, 128, 256, 512

Quick Sort:

- InputType: Random
- MPI num_procs: 2, 4, 8, 16, 32, 64
- CUDA num_threads: 64, 128, 256

Merge Sort:

- InputType: Random
- MPI num_procs: 2, 4, 8, 16, 32
- CUDA num_threads: 64, 128, 256, 512

Radix Sort:

- InputType: Random
- MPI num_procs: 2, 4, 8, 16, 32, 64
- CUDA num_threads: 64, 128, 256, 512, 1024

num_procs, num_threads:

- MPI: num_procs:

  - 2, 4, 8, 16, 32, 64

- CUDA: num_threads:

  - 64, 128, 256, 512, 1024

4b. Hints for performance analysis

To automate running a set of experiments, parameterize your program.

- inputType: If you are sorting, "Sorted" could generate a sorted input to pass into your algorithms

- algorithm: You can have a switch statement that calls the different algorithms and sets the Adiak variables accordingly

- num_procs: How many MPI ranks you are using

- num_threads: Number of CUDA or OpenMP threads

When your program works with these parameters, you can write a shell script

that will run a for loop over the parameters above (e.g., on 64 processors,

perform runs that invoke algorithm2 for Sorted, ReverseSorted, and Random data).

4c. You should measure the following performance metrics

Bitonic Sort Cuda Data

		nid	spot.channel	Min time/rank	Max time/rank	Avg time/rank	Total time	name
node	profile							
{'name': 'main', 'type': 'function'}	23056950	1	regionprofile	0.608071	0.608071	0.608071	0.608071	main
	277482448	1	regionprofile	0.495571	0.495571	0.495571	0.495571	main
	342617063	1	regionprofile	0.342254	0.342254	0.342254	0.342254	main
	1032065064	1	regionprofile	0.604006	0.604006	0.604006	0.604006	main
	1491153857	1	regionprofile	0.457133	0.457133	0.457133	0.457133	main
	1642830958	1	regionprofile	0.328475	0.328475	0.328475	0.328475	main
	2104588869	1	regionprofile	0.666225	0.666225	0.666225	0.666225	main
	2172300790	1	regionprofile	0.531673	0.531673	0.531673	0.531673	main
	2496855500	1	regionprofile	0.609716	0.609716	0.609716	0.609716	main
	2565622787	1	regionprofile	0.689057	0.689057	0.689057	0.689057	main
	2626279528	1	regionprofile	0.664524	0.664524	0.664524	0.664524	main
	3446572744	1	regionprofile	0.465160	0.465160	0.465160	0.465160	main
	3452477975	1	regionprofile	0.332067	0.332067	0.332067	0.332067	main
	3526872179	1	regionprofile	0.659693	0.659693	0.659693	0.659693	main
	3764152635	1	regionprofile	0.235333	0.235333	0.235333	0.235333	main
	3925186993	1	regionprofile	0.343786	0.343786	0.343786	0.343786	main

We were able to successfully extract the data from each of our cali files from each algorithm. We chose to graph the average time that it took for each main function. However, we weren't sure how to graph additional features because it is our first time using thicket. The general trend as stated before was that there was a slight increase in run time at smaller processors/threads and at the end the run time was almost 0.

5. Presentation

Plots for the presentation should be as follows:

- For each implementation:

  - For each of comp_large, comm, and main:

- Strong scaling plots for each InputSize with lines for InputType (7 plots - 4 lines each)

- Strong scaling speedup plot for each InputType (4 plots)

- Weak scaling plots for each InputType (4 plots)

Analyze these plots and choose a subset to present and explain in your presentation.

6. Final 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.

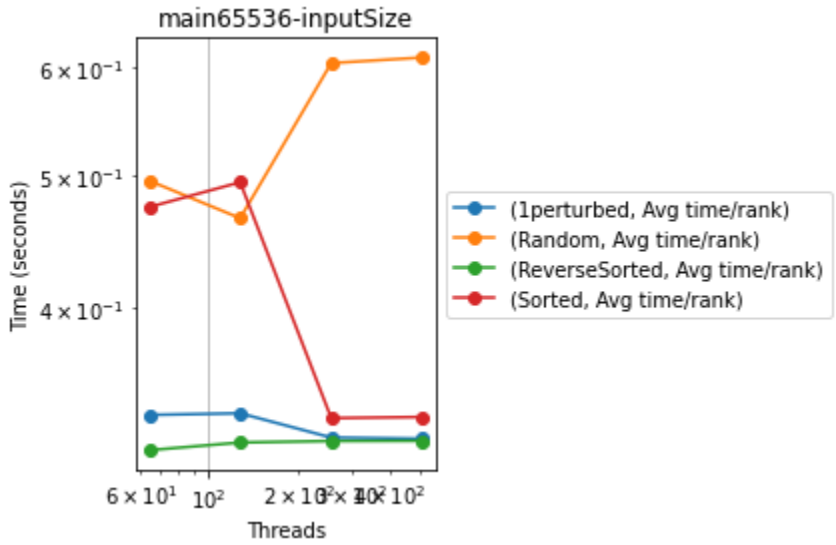
In our analysis, we compared algorithm performance through graphs depicting strong scaling, weak scaling, and speed-up metrics. Our final comparative study focused on evaluating the main runtime performance of CUDA implementations against MPI for Bitonic and Radix Sorts, providing insights into their 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:

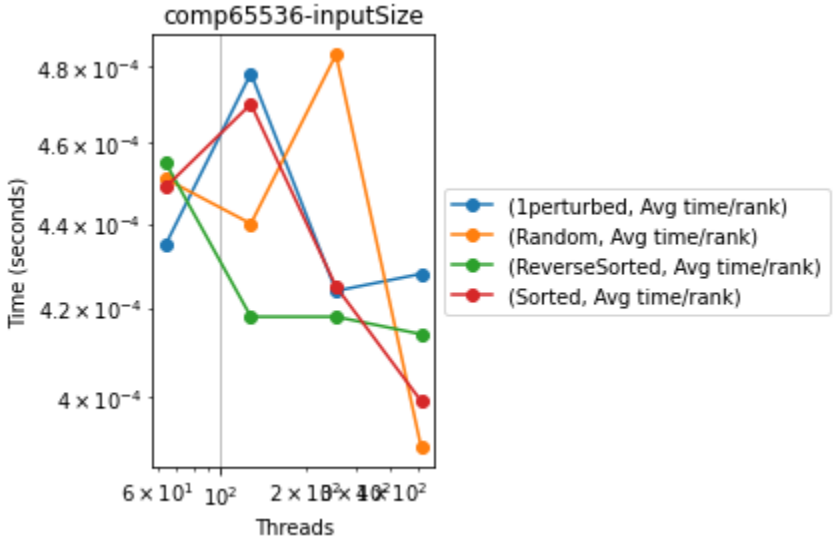
Figure 1.1 - **Main** Function Region
inputSize = 65536



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

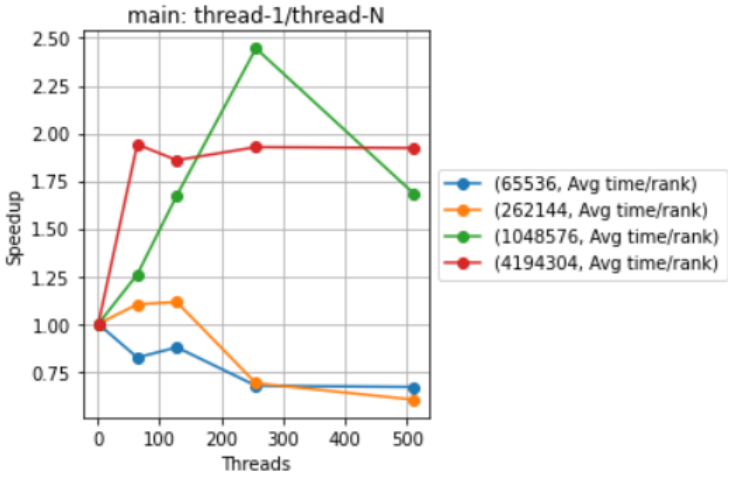
inputSize = 65536



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.

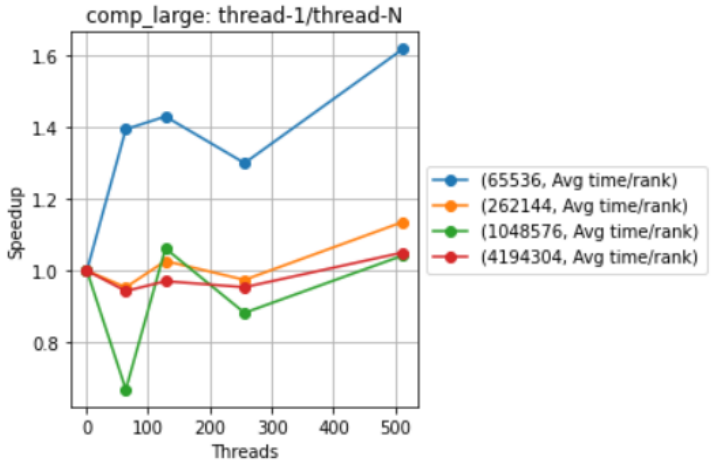
Speedup:

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.

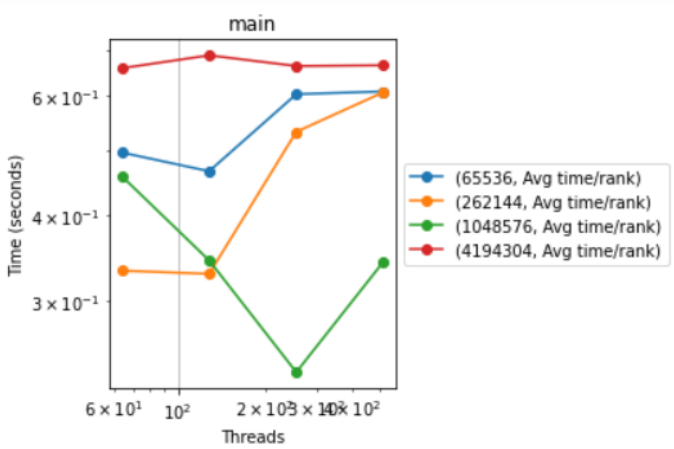
Figure 1.4 - **Comp Large** Function Region
InputType = Random



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.

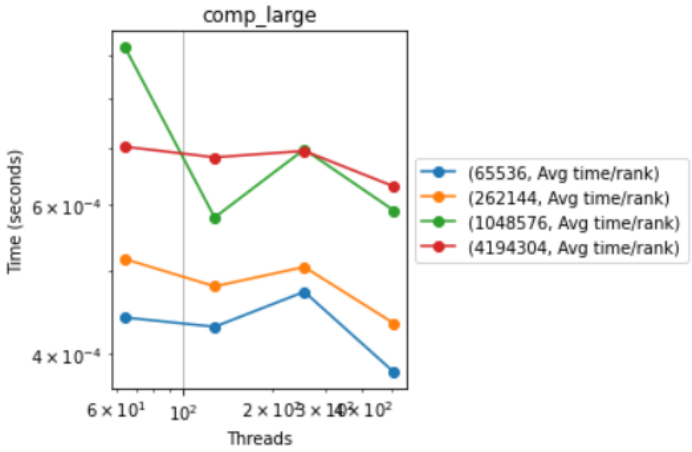
Weak Scaling:

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.

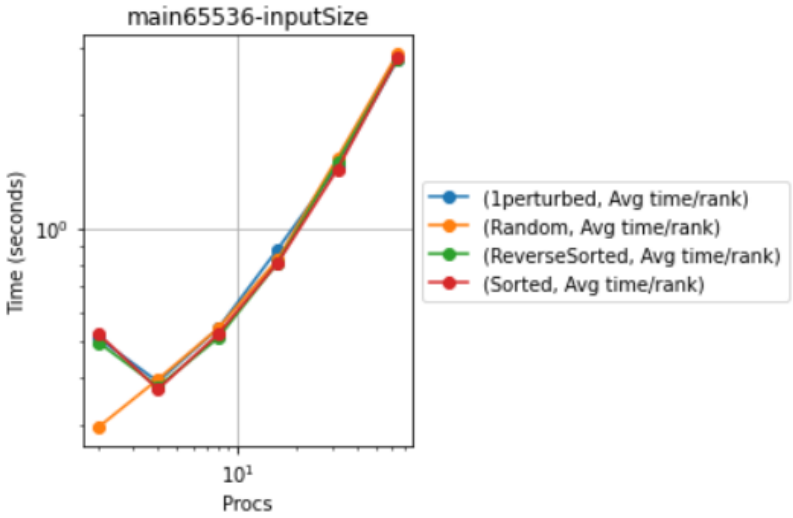
Cuda Observations:
Overall, I am pleased with the results that came from the Cuda Implementation of Bitonic Sort. Speedup and Strong Scaling are easily the most accurate and weak scaling has a slight hiccup due to communication overhead. It is true that the main method didn't always output the intended 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:

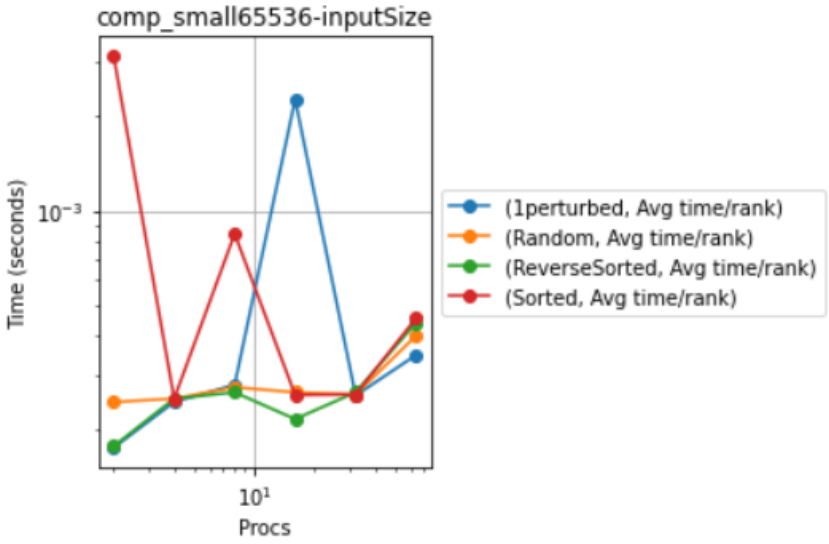
Strong scaling:

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.

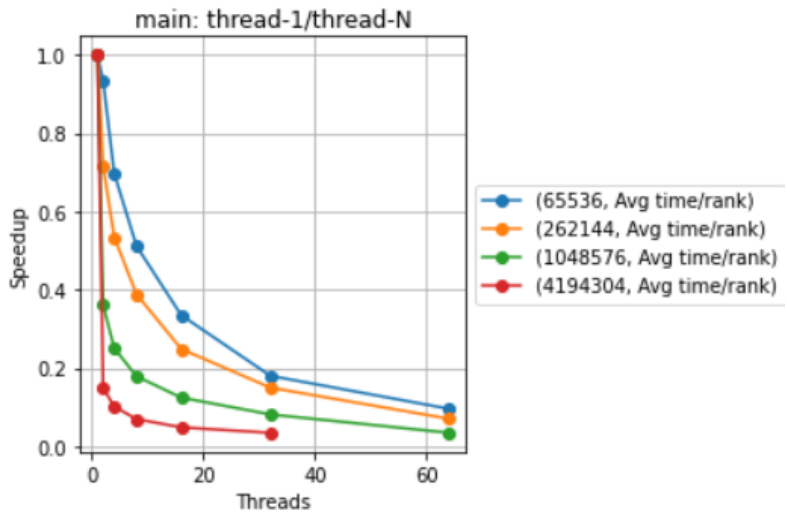
Figure 1.8 - **Comp Small** Function Region
InputSize = 65536



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.

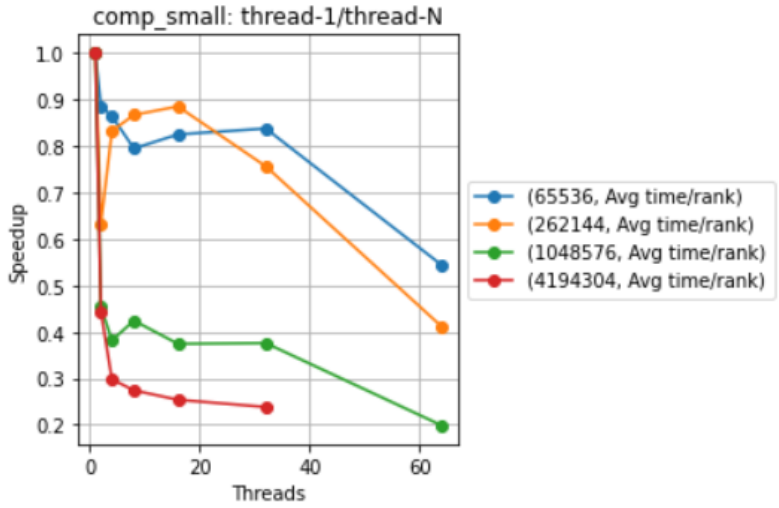
Speedup:

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

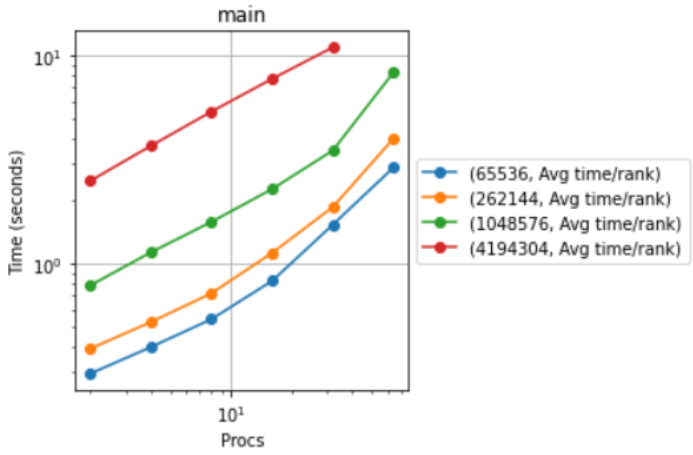


Note: The speedup graph above for comp small function region looks a little more promising than the previous graph. It appears that speedup again is good at the middle number of procs. However the speedup is a little wrong at the smaller and larger end of the procs. There is a distinct plateau at the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

Weak Scaling:

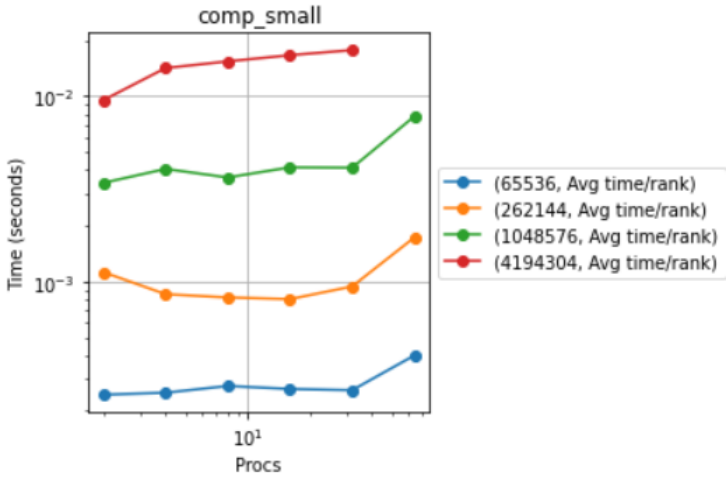
Figure 1.11 - **Main** Function Region

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

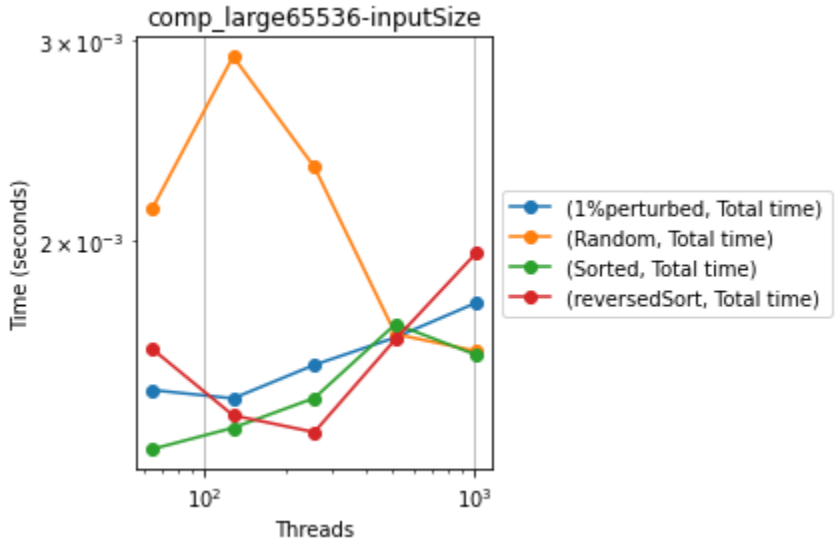
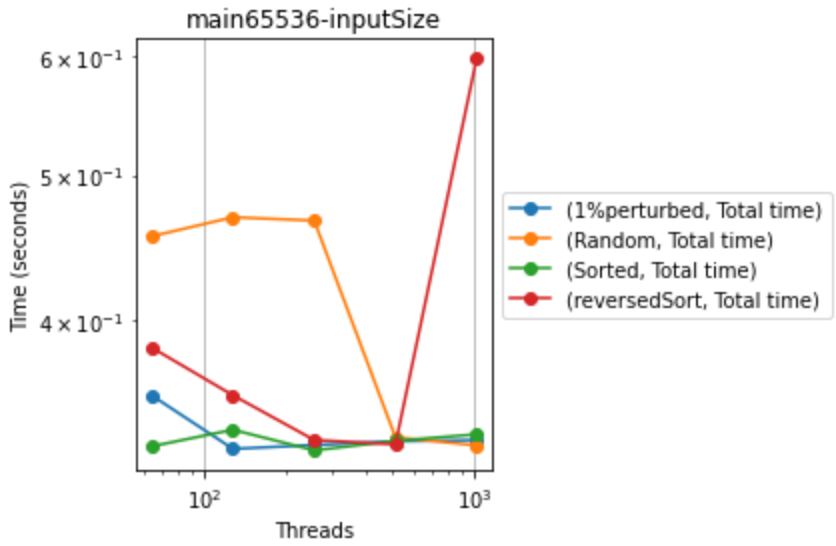


Figure 2.2 - Main Function Region



Weak Scaling for size of 65536:

Figure 2.3 - Large Computation Weak Scaling

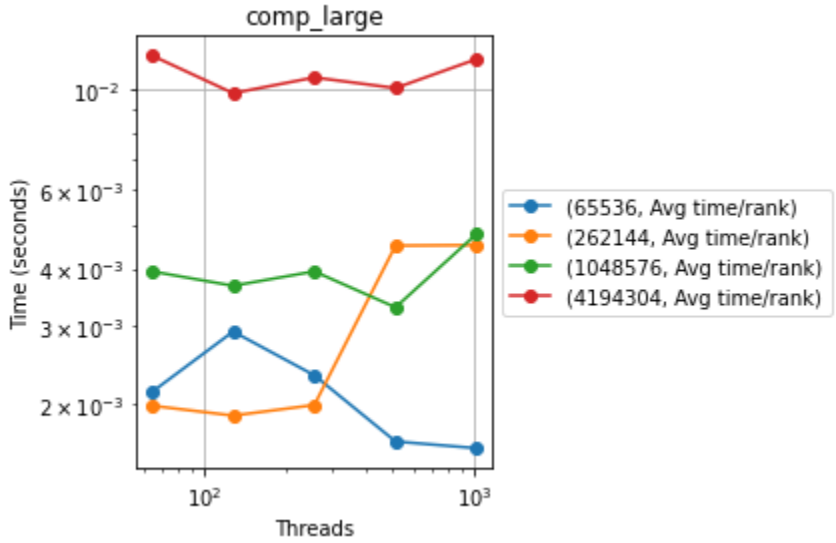
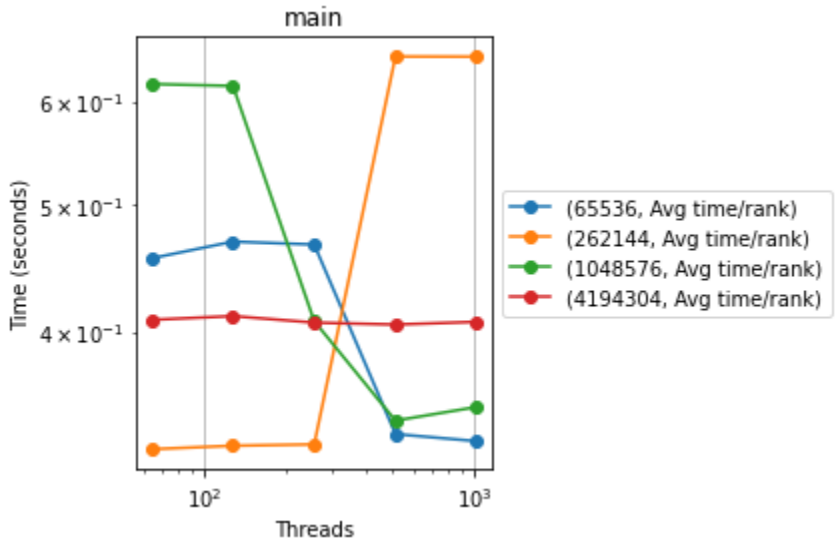


Figure 2.4 - Main Function Weak Scaling



Speedup for Random Datasets:

Figure 2.5 - Large Computation Speedup

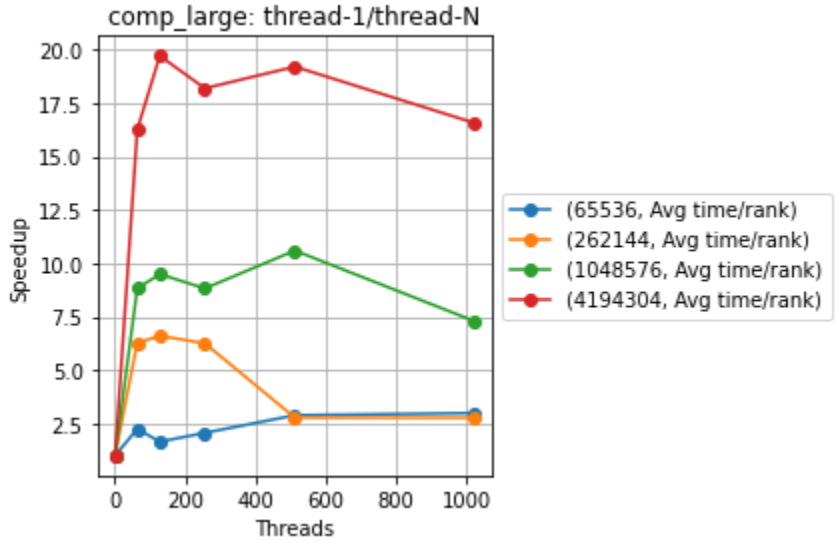
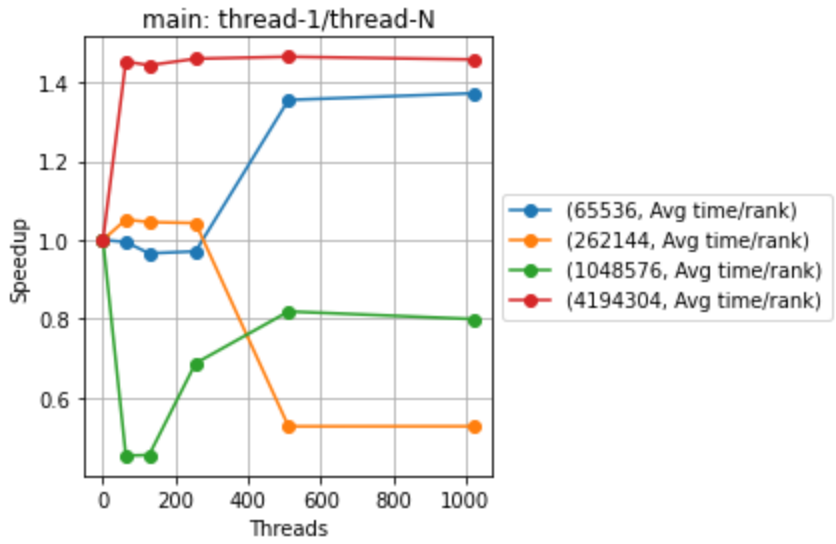


Figure 2.6 - Main Function Speedup



MPI Graphs:

Cuda Graphs:

Strong Scaling for size of 65536:

Figure 2.7 Large Computation

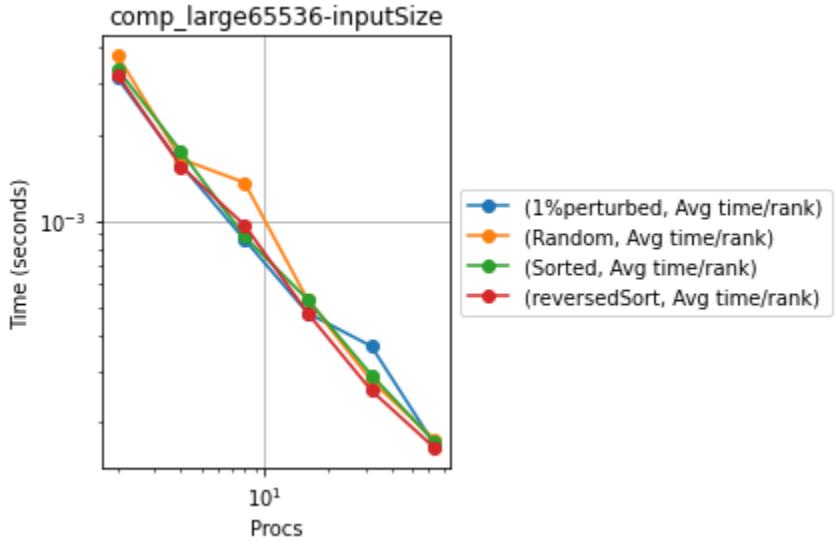
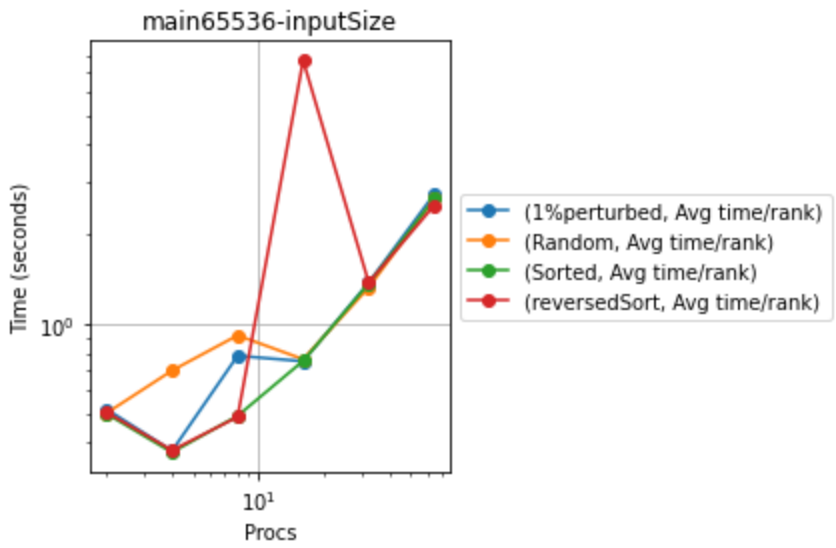


Figure 2.8 Main Function



Weak Scaling for Random Datasets:

Figure 2.9 Large Computation

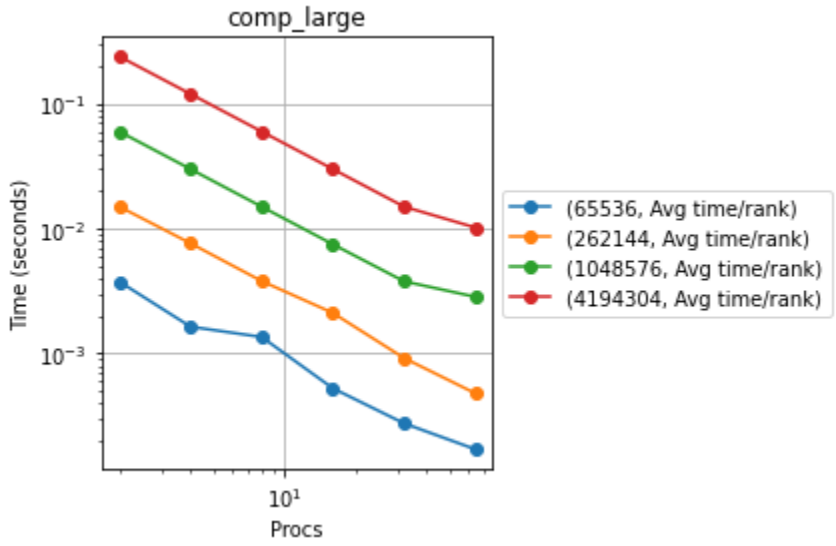
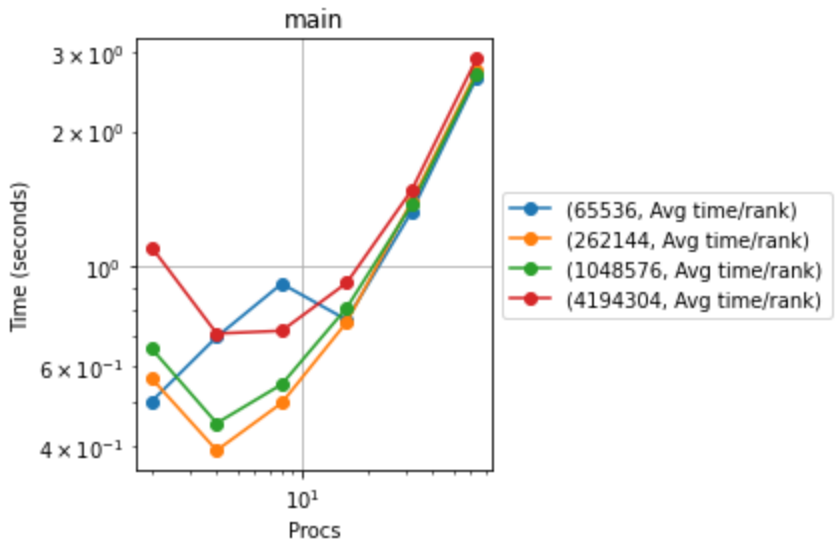


Figure 2.10 Main Function



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

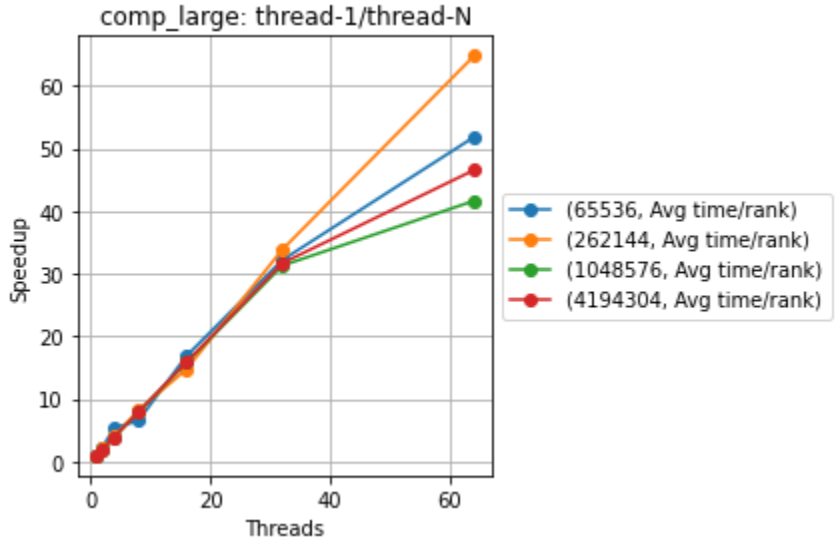
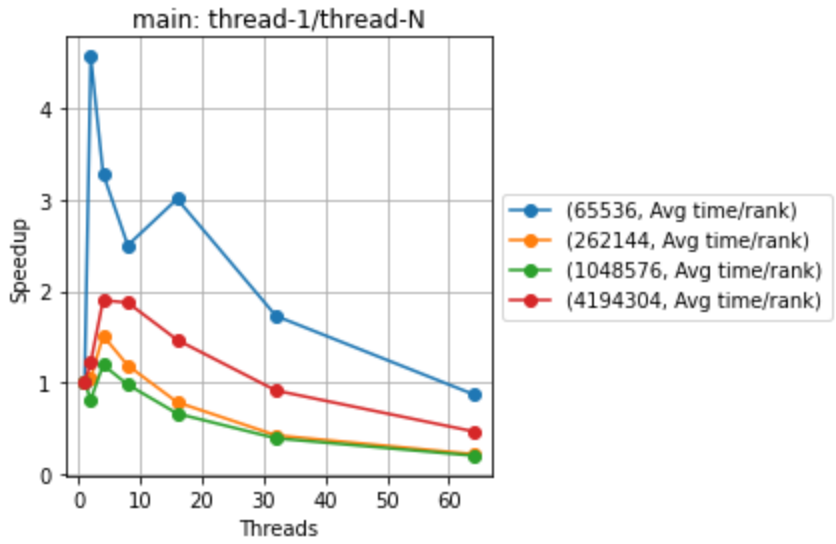


Figure 2.12 Main Function



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.

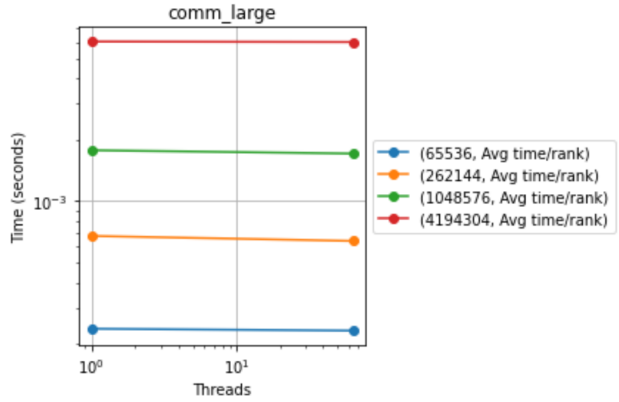
Since radix sort sorts based on the number of digits, there could be cases depending on the magnitude of data points being sorted where it may not be the most optimal. Data points in values ranging from 0 to 9999 were in the arrays when being tested. For a smaller number of digits per data point on average 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

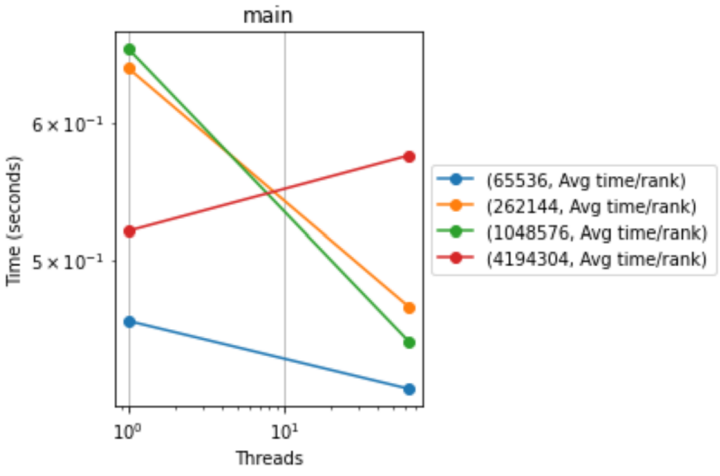
Strong scaling

Figure 3.1 - comm_large region



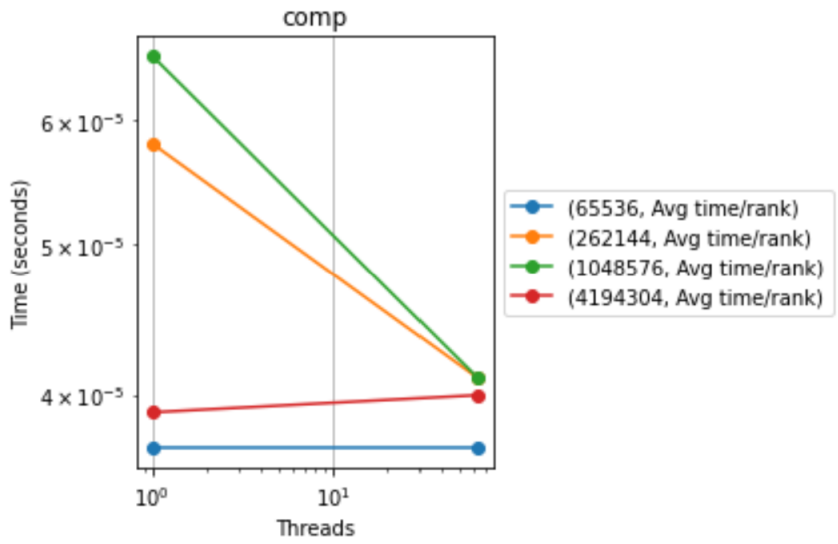
We should expect to see a linear decrease in time as we double the number of threads. However, because my quicksort cuda algorithm would time out at larger input sizes and processes, I wasn't able to plot more data. This goes for my other quicksort plots as well.

Figure 3.2 - main region



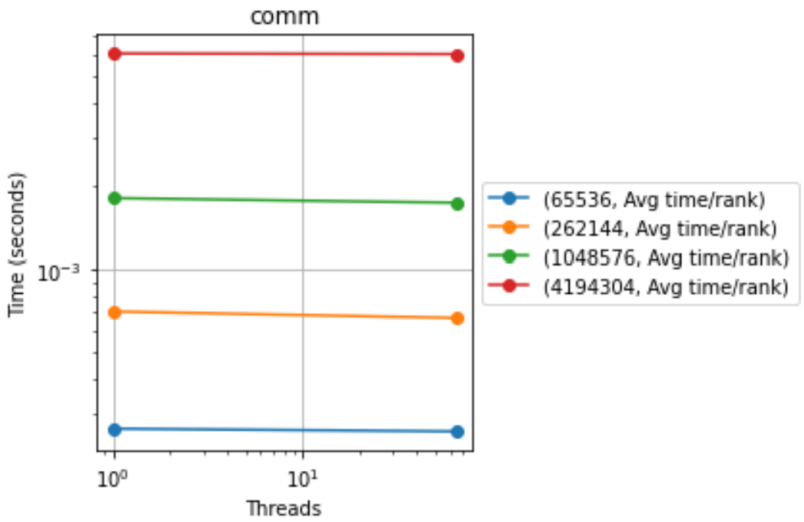
The line representing input size of 4194304 is counterintuitive since the line increases rather than decreases.

Figure 3.2 - comp region



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

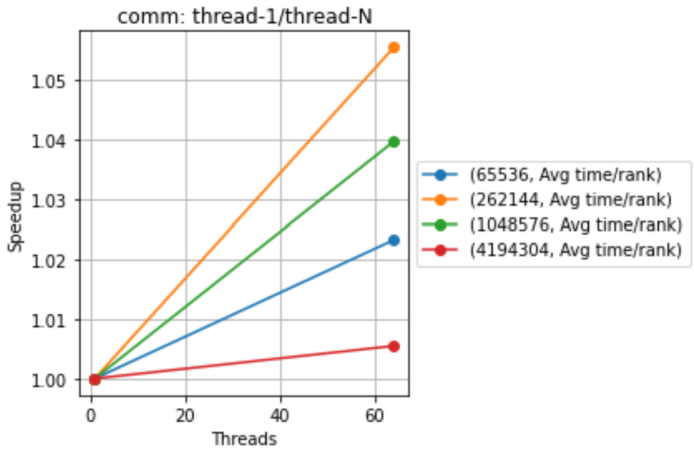
Figure 3.3 - comm



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.

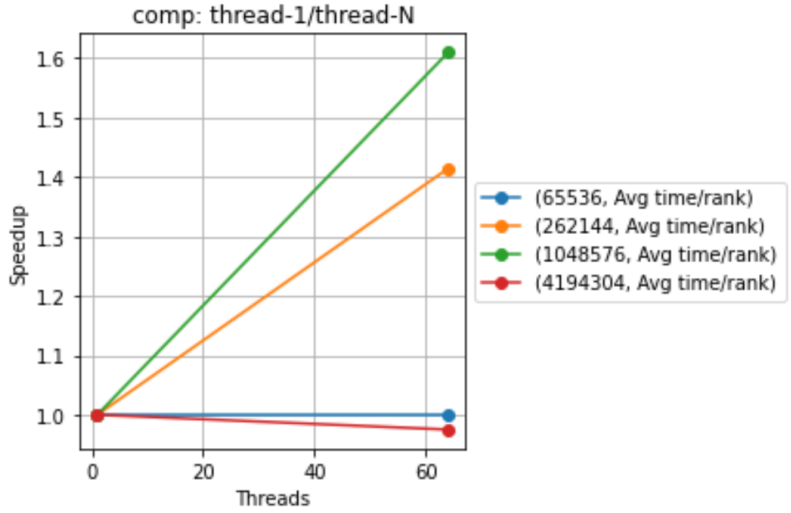
Speedup

Figure 3.4 - comm 1 thread



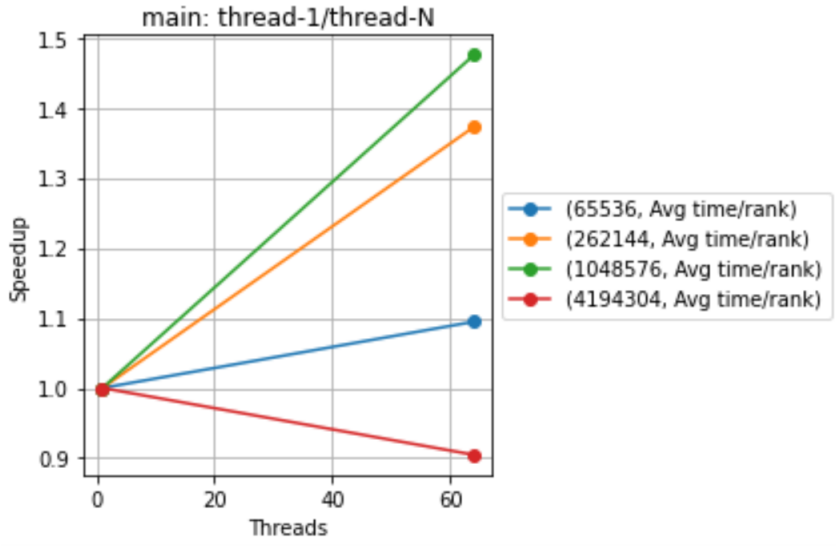
This speedup is as expected since the trend is increasing.

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

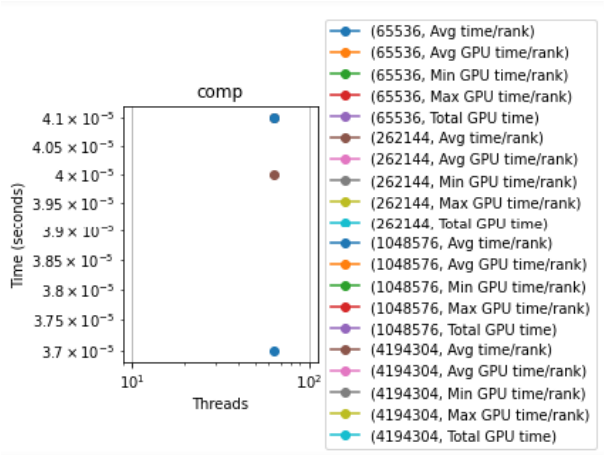


Figure 3.8 - main

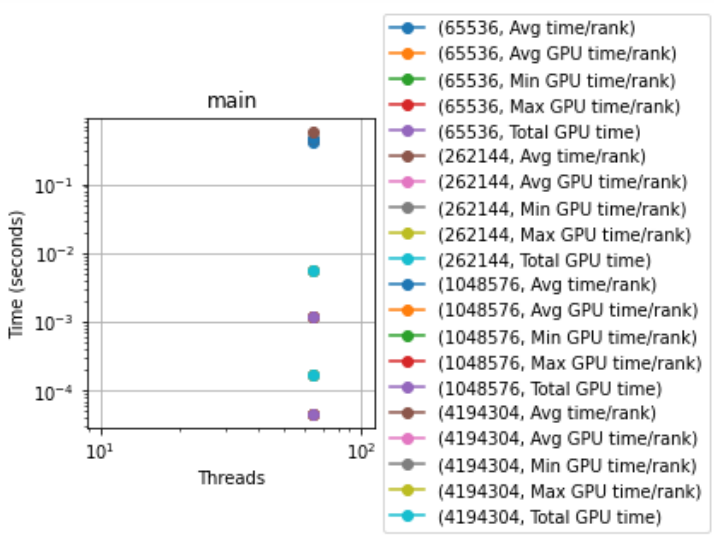
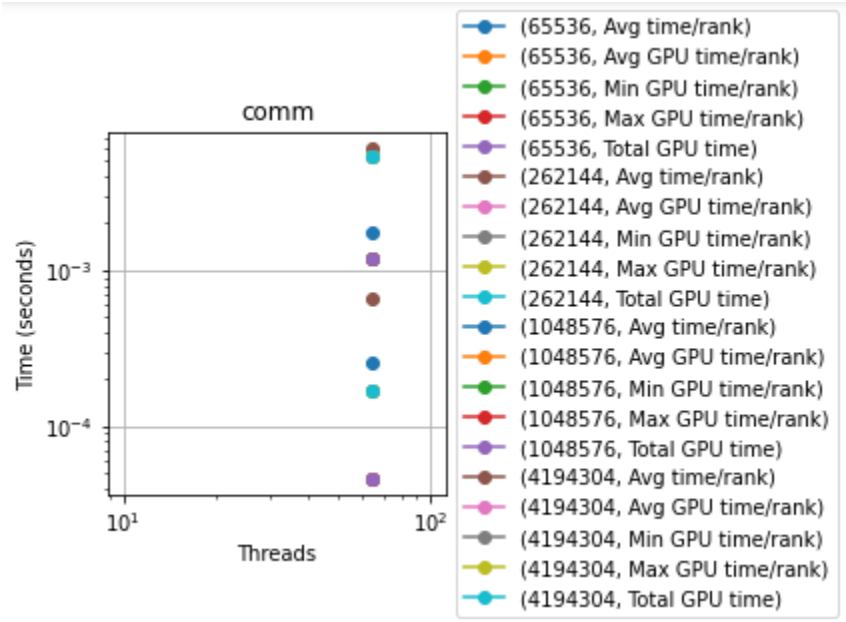


Figure 3.9 - comm

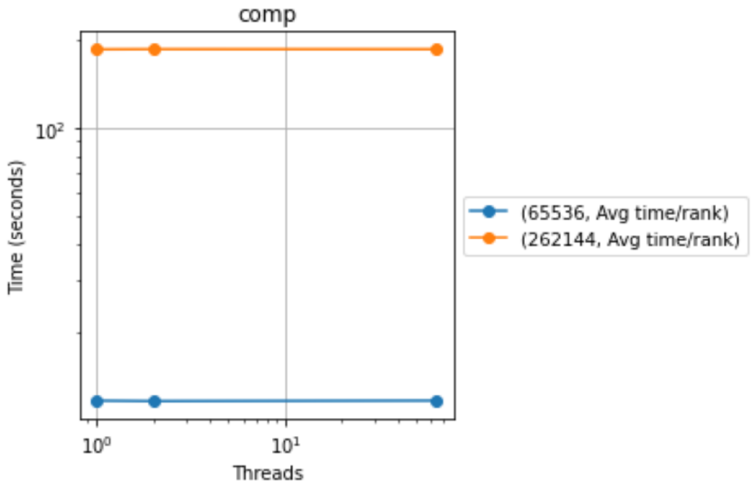


My weak scaling graphs appear to have just a single thread value at each input size because I was unable to run larger input and thread values due to time out.

MPI

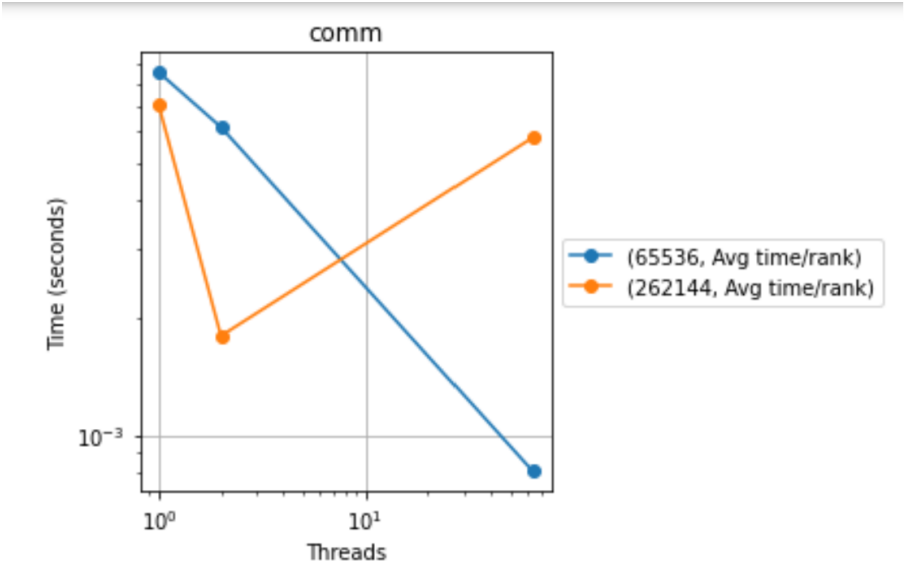
Strong scaling

Figure 3.10

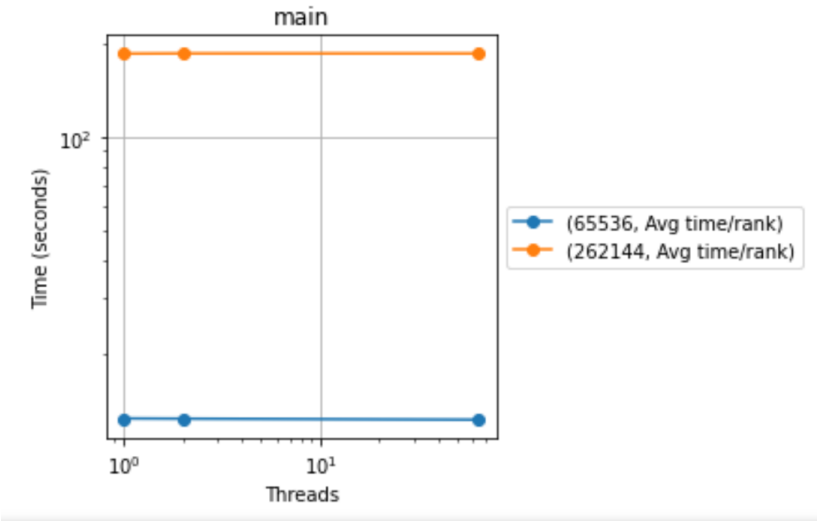


The MPI quicksort also appears to have two lines since I was unable to run larger input sizes and processes due to runtime. Both trends appear to be flat, when decreasing trends are expected.

Figure 3.11



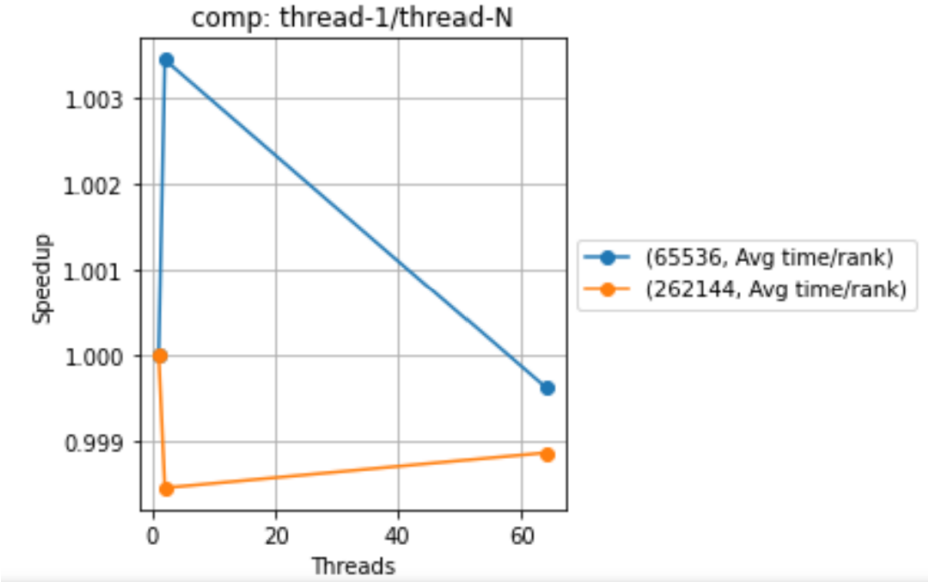
The line representing input size 262144 appears to increase when the general trend should be decreasing.



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.

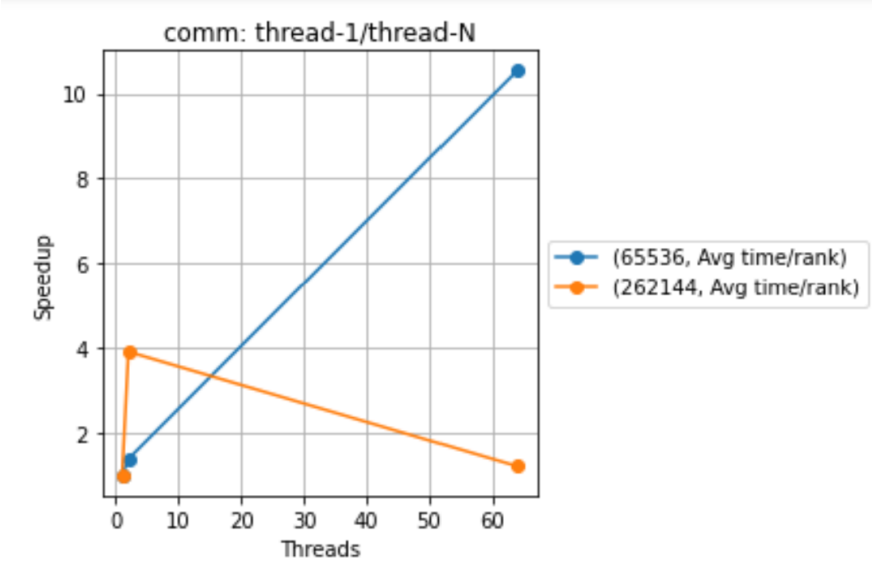
Speedup

Figure 3.13



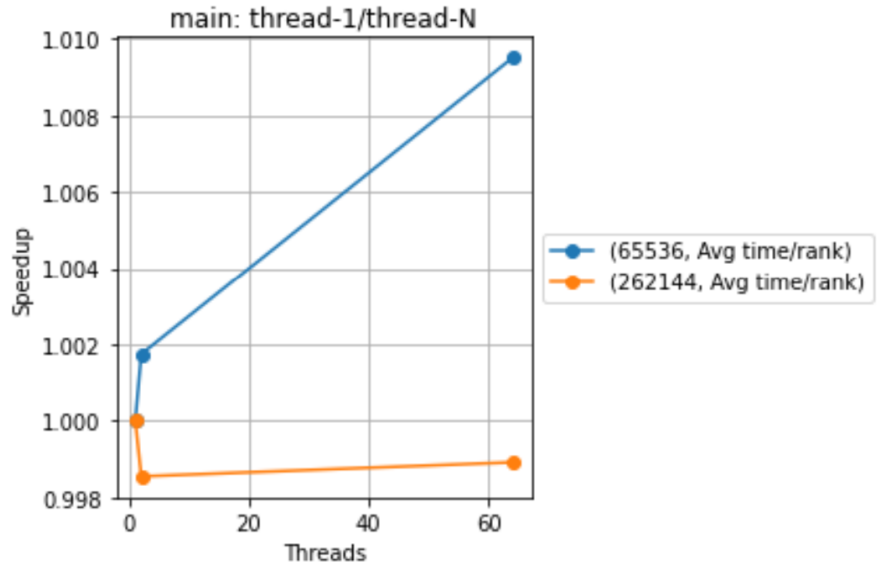
Speedup graphs are expected to have an increasing trend. This shows performance isn't efficient.

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.

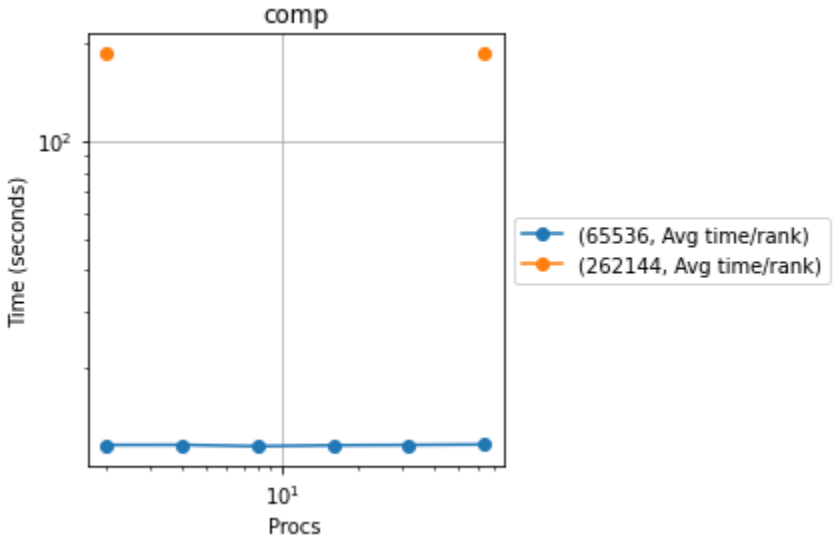
Figure 3.15



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

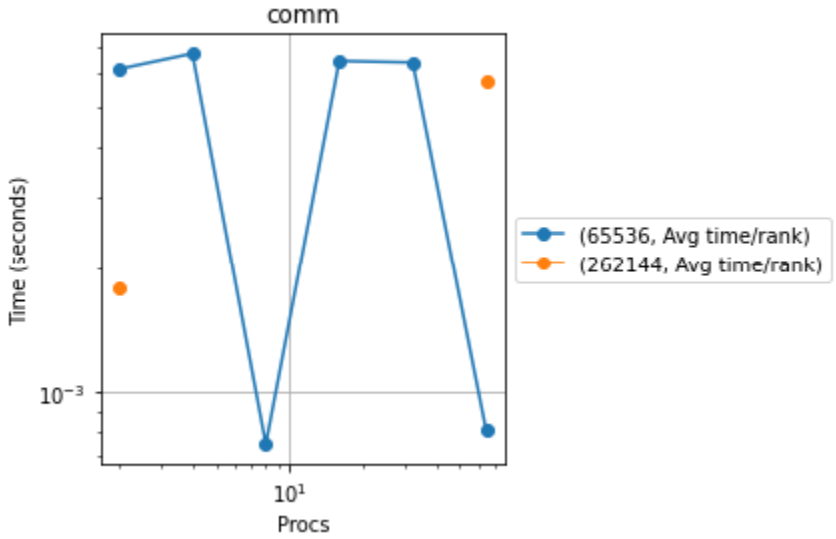
Weak scaling

Figure 3.16



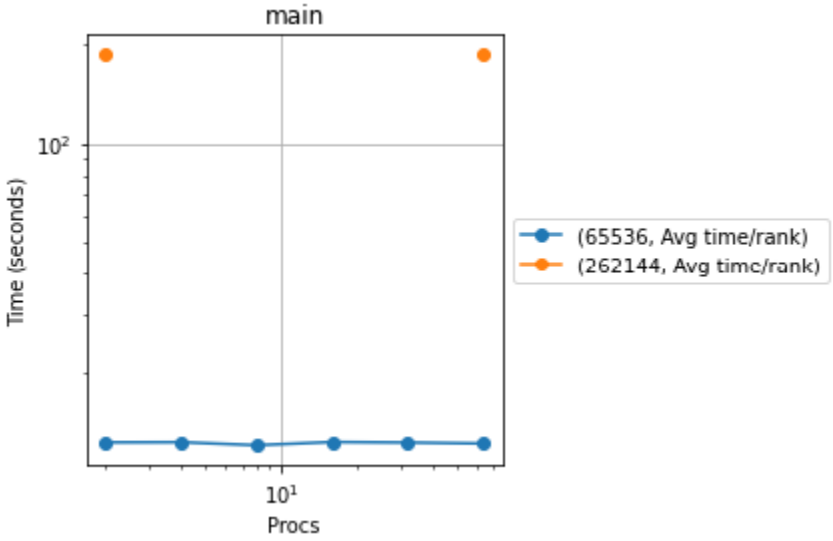
Ideally if the quicksort scales well, the weak scaling graph should have a constant trend. This shows that the algorithm scales weakly for the number of processors shown.

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.

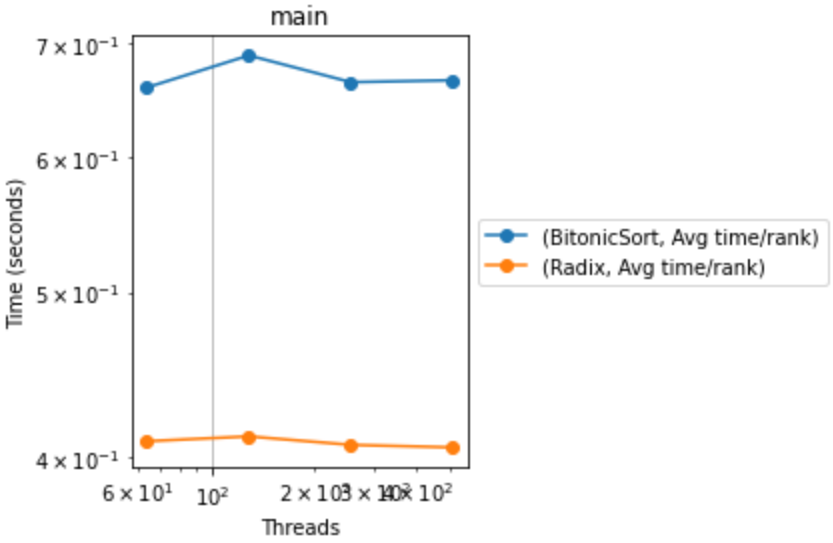
Merge Sort:

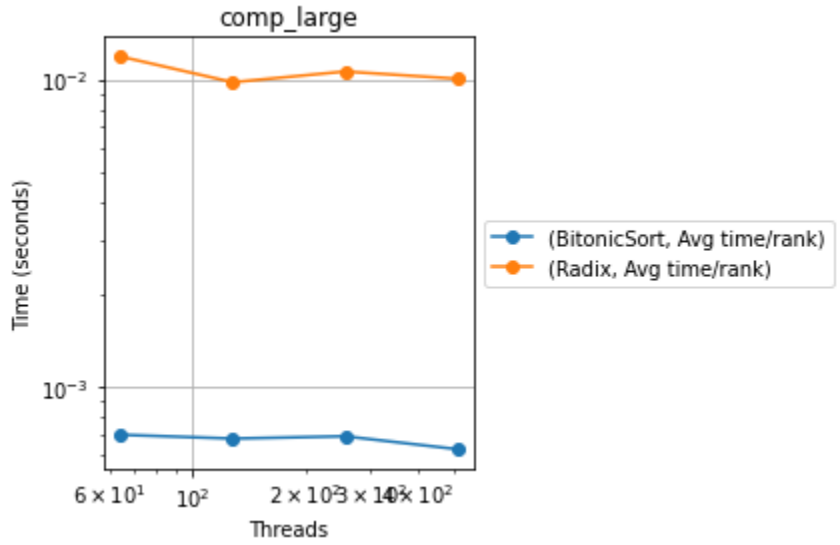
<Intentionally left blank>

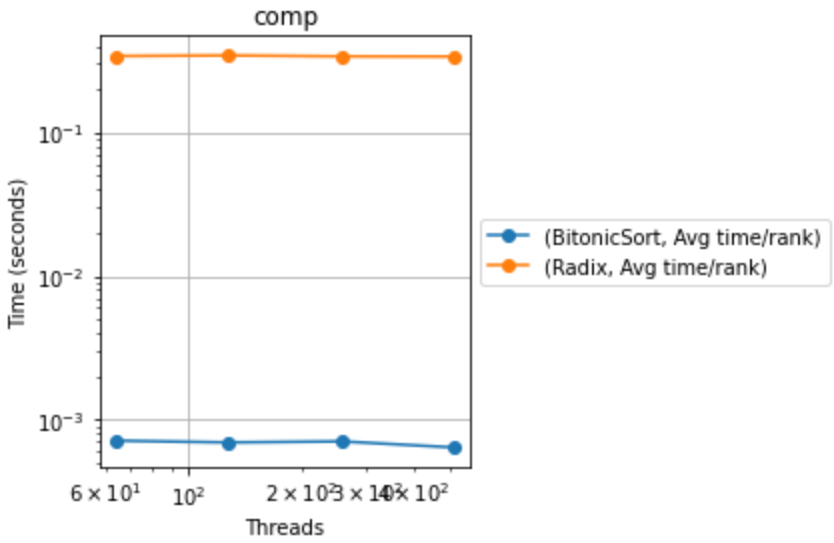
Bitonic vs. Radix:

Cuda:

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

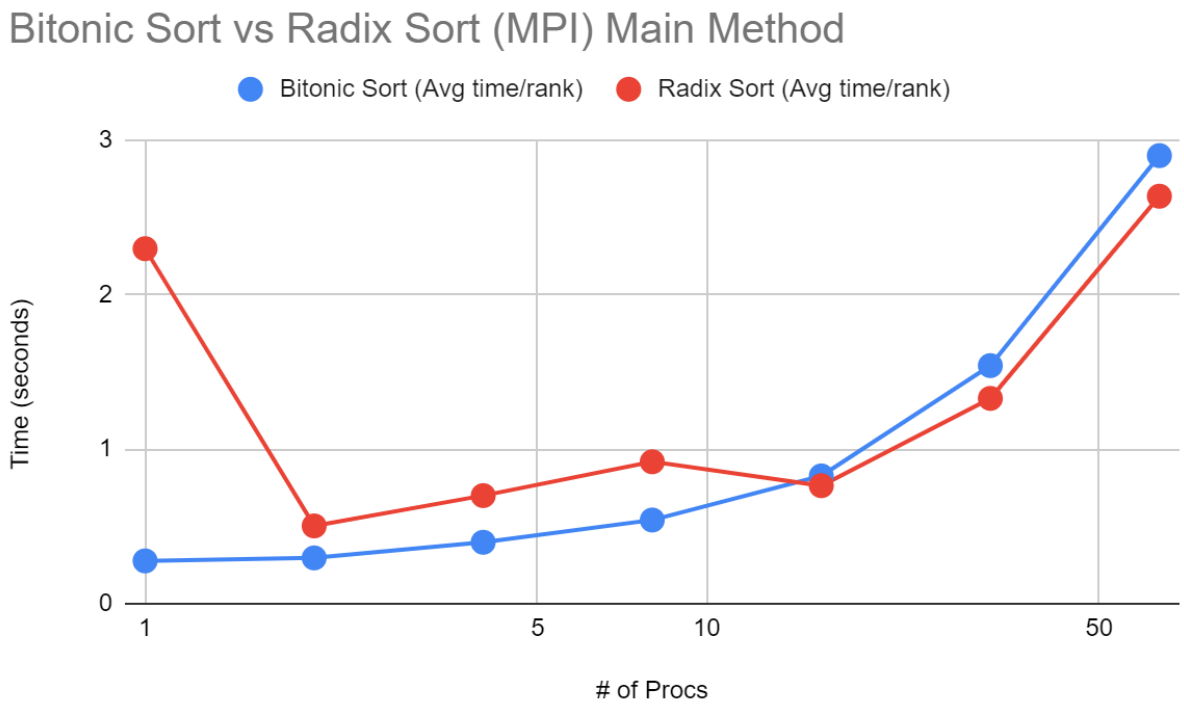

main
Time (seconds)
Threads
(BitonicSort, Avg time/rank)
(Radix, Avg time/rank)

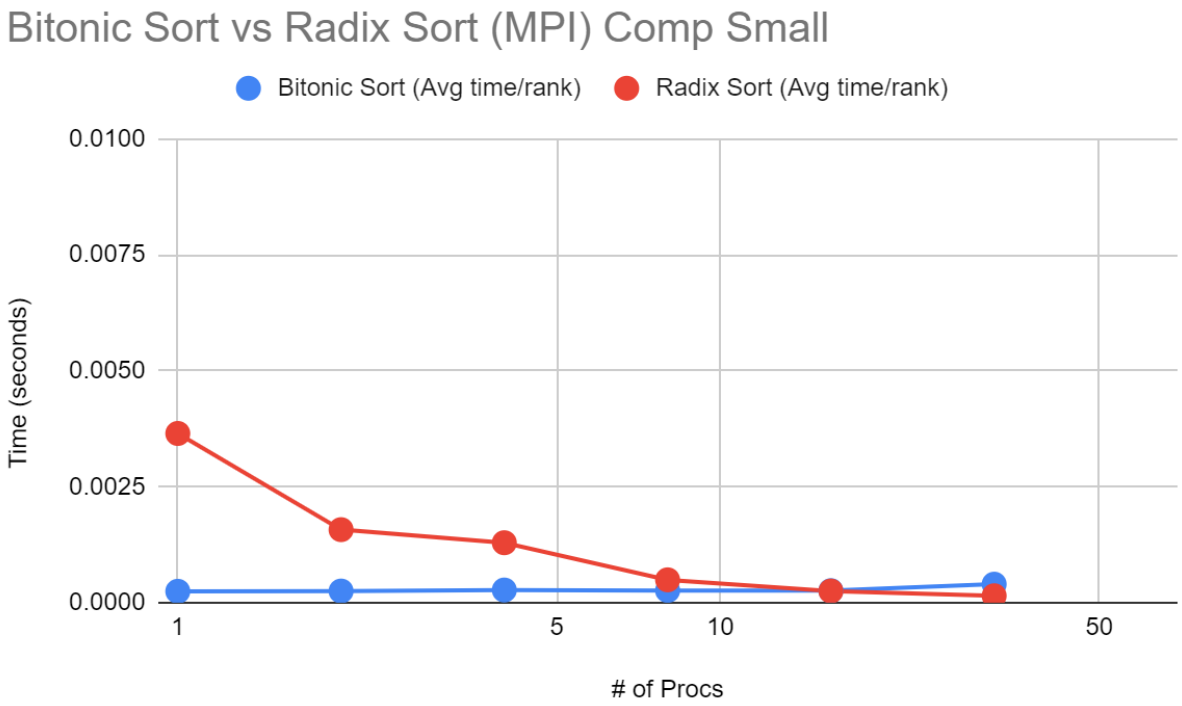

comp_large
Time (seconds)
Threads
(BitonicSort, Avg time/rank)
(Radix, Avg time/rank)



MPI:

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

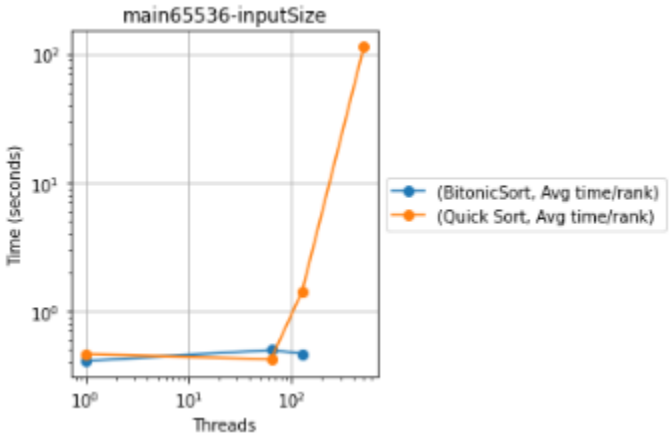


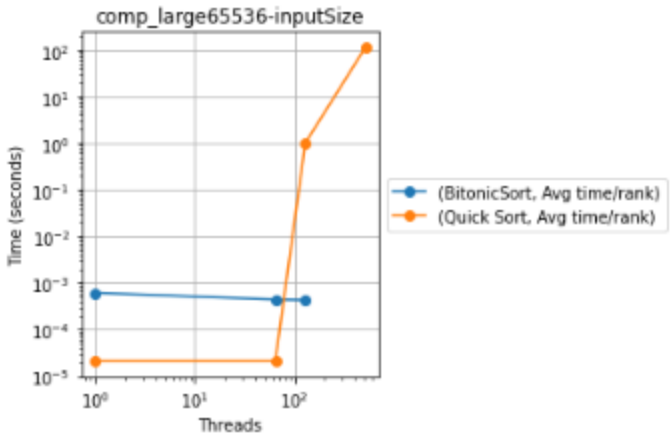


Bitonic vs. Quick Sort:

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.