CSCE 435 Group 16 project

0. Group number: 16

1. Group members:

1. Anjali Kumar
2. Shawn Mathen
3. Ashrita Vadlapatla
4. Robert Eads

2. Project topic (e.g., parallel sorting algorithms)

Parallel Sorting Algorithms

2a. Communication Method

Our team's primary method of communication will be GroupMe with Slack as a secondary method.

2b. Brief project description (what algorithms will you be comparing and on what architectures)

Each of the selected sort algorithms, Bitonic, Merge, Selection, & Sample, will be run using MPI and CUDA separately.

2c. Pseudocode for each parallel algorithm

Algorithm 1: Bitonic Sort

MPI

```
1. Distribute Data: Use MPI_Scatter to distribute the data among processes. Each
process will receive a portion of the dataset.
2. Local Bitonic Sort: Each process performs a local Bitonic Sort on its portion
of the data. This involves performing Bitonic Sort on the local dataset using
parallel threads or any parallel programming model within the process.
3. Bitonic Merge: Implement the Bitonic Merge operation for pairs of processes.
This involves exchanging data between neighboring processes to create a bitonic
sequence.
4. Global Bitonic Sort: Perform multiple iterations of the Bitonic Sort while
exchanging and comparing adjacent elements between neighboring processes. In each
iteration, processes exchange data and perform Bitonic Merge.
5. Repeat for Required Iterations: Repeat the Bitonic Sort and data exchange steps
for the required number of iterations to ensure a fully sorted dataset. The number
of iterations depends on the size of the dataset and the number of processes.
6. Verify Correctness: After the sorting is complete, use a verification step to
ensure the correctness of the sorted data. You can implement a global verification
```

```
step where each process checks the correctness of its portion, or you can gather
data to the root process for a centralized verification.
```

CUDA

```
1. Allocate Device Memory: Use cudaMalloc to allocate memory on the GPU for the
dataset. Allocate memory for temporary buffers if needed.
2. Copy Data to GPU: Use cudaMemcpy to copy the data from the host to the GPU.
Launch Kernel for Local Bitonic Sort:
3. Write a CUDA kernel to perform local Bitonic Sort on the GPU. Each thread will
work on a portion of the dataset. Threads in the same block can cooperate to
perform the sort using shared memory.
4. Synchronize Threads: Use __syncthreads() to synchronize threads within a block
after the local sort.
5. Launch Bitonic Merge Kernel: Write a CUDA kernel to perform Bitonic Merge
operation for pairs of blocks. This may involve exchanging data between
neighboring blocks and creating bitonic sequences.
6. Synchronize Threads Again: Use __syncthreads() to synchronize threads after the
merge operation.
7. Repeat for Multiple Iterations: Repeat steps 3-6 for the required number of
iterations. The number of iterations depends on the size of the dataset and the
structure of the bitonic sequence.
8. Copy Data Back to CPU: Use cudaMemcpy to copy the sorted data from the GPU back
to the host.
9. Free Device Memory: Use cudaFree to release the allocated memory on the GPU.
10. Verify Correctness: After the sorting is complete, use a verification step to
ensure the correctness of the sorted data. You can use a similar verification
function as before.
```

General Pseudocode

```
void bitonicMerge(int a[], int low, int cnt, int dir)
{
    if (cnt>1)
    {
        int k = cnt/2;
        for (int i=low; i<low+k; i++)
            compAndSwap(a, i, i+k, dir);
        bitonicMerge(a, low, k, dir);
        bitonicMerge(a, low+k, k, dir);
    }
}

void bitonicSort(int a[],int low, int cnt, int dir)
{
    if (cnt>1)
    {
        int k = cnt/2;
```

```
        // sort in ascending order since dir here is 1
        bitonicSort(a, low, k, 1);

        // sort in descending order since dir here is 0
        bitonicSort(a, low+k, k, 0);

        // Will merge whole sequence in ascending order
        // since dir=1.
        bitonicMerge(a,low, cnt, dir);
    }
}
```

MPI Pseudocode

```
int main(int argc, char **argv) {
    // MPI Initialization

    // Scatter data among processes
    MPI_Scatter(/* ... */);

    // Local Bitonic Sort on each process
    localBitonicSort(/* ... */);

    // Global Bitonic Sort
    for (int k = 2; k <= numProcesses; k *= 2) {
        for (int j = k / 2; j > 0; j /= 2) {
            // Bitonic Merge for pairs of processes
            bitonicMerge(/* ... */);
        }
    }

    // Gather sorted data to root process
    MPI_Gather(/* ... */);

    // MPI Finalization

    return 0;
}
```

CUDA Pseudocode

```
__global__ void bitonicSortKernel(int *data, int dataSize) {
    // Calculate thread index
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    // Bitonic Sort
    for (int k = 2; k <= dataSize; k *= 2) {
```

```
        for (int j = k / 2; j > 0; j /= 2) {
            int ixj = tid ^ j;

            if ((ixj) > tid) {
                if ((tid & k) == 0) {
                    // Sort ascending
                    if (data[tid] > data[ixj]) {
                        // Swap elements
                        int temp = data[tid];
                        data[tid] = data[ixj];
                        data[ixj] = temp;
                    }
                }
                if ((tid & k) != 0) {
                    // Sort descending
                    if (data[tid] < data[ixj]) {
                        // Swap elements
                        int temp = data[tid];
                        data[tid] = data[ixj];
                        data[ixj] = temp;
                    }
                }
            }

            __syncthreads(); // Synchronize threads after each iteration
        }
    }
}
```

Algorithm 2: Merge Sort

MPI

```
1. Distribute data among processes
2. Each process performs a local sequential merge sort
3. Pairwise merging of sorted sublists using MPI_Send and MPI_Recv
4. Recursively perform merging until the data is fully sorted
5. Verify correctness
```

CUDA

```
1. Each CUDA thread loads a portion of the data into shared memory
2. Perform a local sequential merge sort within each thread's shared memory
3. Use CUDA parallel reduction to find the pivot elements
4. Broadcast to all threads
5. Each thread partitions its data around pivot
6. Calculate the offsets for each partition
```

```
7. CUDA scatter to move elements to their respective partitions
8. Recursively sort each partition using CUDA parallel sort
9. Perform parallel merge or merge in shared memory, depending on the size of the
partitions
```

Pseudocode

Parallel: The following parallel Merge Sort algorithms will be implemented using MPI.

```
function parallel_merge_sort(arr, num_threads)
    if length(arr) <= 1
        return arr

    mid = length(arr) // 2
    left = arr[0...mid-1]
    right = arr[mid...]

    if num_threads > 1
        left_thread = start_thread(parallel_merge_sort(left, num_threads/2))
        right_thread = start_thread(parallel_merge_sort(right, num_threads/2))
        join_thread(left_thread)
        join_thread(right_thread)

    else
        left = merge_sort(left)
        right = merge_sort(right)

    return parallel_merge(left, right)
```

The following is pseudocode for a CUDA implementation

```
__global__ void mergeCUDA(int* arr, int left, int mid, int right) {
    // Sequential merge
}

__global__ void mergeSortParallelCUDA(int* arr, int size) {
    if (size <= 1) return;

    int mid = size / 2;

    int* left = arr;
    int* right = arr + mid;

    mergeSortParallelCUDA<<<1, 1>>>(left, mid);
    mergeSortParallelCUDA<<<1, 1>>>(right, size - mid);

    mergeCUDA(arr, 0, mid - 1, size - 1);
}
```

Algorithm 3: Selection Sort

MPI

```
1. Distribute the data among processes using MPI_Scatter.
2. Each process applies the selection sort algorithm to its portion of the data.
3. The sort selects the minimum element and swaps it with the first unsorted
element so that there is a local sorted segment.
3. Communicate with neighboring processes using MPI_Send and MPI_Recv to exchange
boundary data so that elements are placed correctly.
4. There is no need for a scatter operation, as Selection Sort is an in-place
sorting algorithm.
5. Repeat the Selection Sort for the required number of iterations, ensuring all
elements are correctly sorted.
6. Verify the correctness of the sorted data.
```

CUDA

```
1. Each CUDA thread loads a portion of the data into shared memory
2. Implement selection sort within the shared memory to locally sort the data.
3. Utilize CUDA parallel reduction techniques to find the minimum element within
the shared memory.
4. Share the minimum element among all threads.
5. Partition the data based on the broadcasted minimum element.
6. Determine offsets for each partition to facilitate data movement.
7. Utilize CUDA scatter operations to move elements to their respective
partitions.
8. Continue the selection sort for a certain number of iterations.
9. As selection sort is an exchange-based sorting algorithm that works directly on
the data in place, there is no merging.
```

```
void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    // One by one move boundary of
    // unsorted subarray
    for (i = 0; i < n - 1; i++) {

        // Find the minimum element in
        // unsorted array
        min_idx = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx])
                min_idx = j;
```

```
        }

        // Swap the found minimum element
        // with the first element
        if (min_idx != i)
            swap(arr[min_idx], arr[i]);
    }
}
```

Algorithm 4: Sample Sort

MPI

```
FOR number of splitters selected by each process
      Push back randomly selected value from local data to sample vector
Allocate memory for all sampled splitters
MPI_Allgather sampled splitters to each process
Sort sampled splitters and choose bucket bounds from thoughout sampled splitters
Evaluate local elements and place into send buffers
Calculate required buffer size from each process
FOR number of processes
      MPI_Gather the buffer sizes from each process on each process
Allocate receive buffer for incoming data
FOR number of processes
      MPI_Gatherv to send data to the correct process
Sort with sequential quicksort
```

CUDA

```
*All steps are performed with CUDA kernals*
Sample elements
Sort samples
Select pivots from sampled elements
Gather required size and start position for each bucket
Group elements into buckets within an array
Have each threads sort the data within the bounds of its respective bucket
```

2d. Citations

Bitonic Sort

- https://www.geeksforgeeks.org/bitonic-sort/
- https://www.tools-of-computing.com/tc/CS/Sorts/bitonic_sort.htm

Merge Sort

- https://compucademy.net/algorithmic-thinking-with-python-part-3-divide-and-conquer-strategy/#:~:text=There%20is%20a%20really%20clever%20trick%20that,the%20same%20type%20as%20the%20original%20problem.
- https://teivah.medium.com/parallel-merge-sort-in-java-e3213ae9fa2c
- https://pushkar2196.wordpress.com/2017/04/19/mergesort-cuda-implementation/

Selection Sort

- https://www.geeksforgeeks.org/selection-sort/

Sample Sort

- https://en.wikipedia.org/wiki/Samplesort
- https://cse.buffalo.edu/faculty/miller/Courses/CSE702/Nicolas-Barrios-Fall-2021.pdf
- https://www.geeksforgeeks.org/quicksort-using-random-pivoting/

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

Each algorithm with be run with the input types of randomized, sorted, reverse sorted, and 1% perturbed. The input sizes are planned to be 2^16, 2^18, 2^20, 2^22, 2^24, 2^26 and 2^28. Graphs for strong and weak scaling, as well as strong speedup, will be created and used in the analysis and drawing of conclusions for each algorithm across the different input types and input sizes. A strong scaling comparison will also be run for the algorithms against each other to see if and where one will stand out from the rest. The number of threads in a block on the GPU to be tested will be [64, 128, 256, 512, 1024].

3. Project Implementation

The listed algorithms were fully implemented using both MPI and CUDA separately.

4. Performance Evaluation

Sample Sort

Note: The CUDA implementation of sample sort did not require any comm regions (cudaMemcpy), so the following section will have no comm graphs for the CUDA subsections.

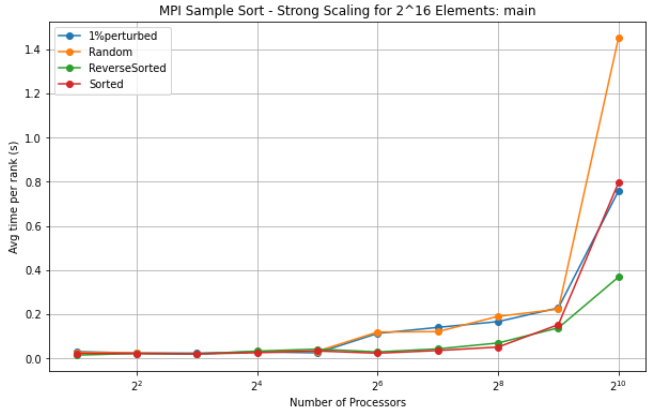
Strong Scaling

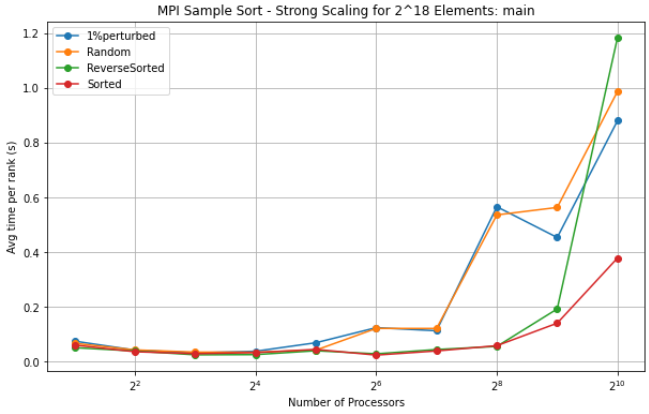
MPI

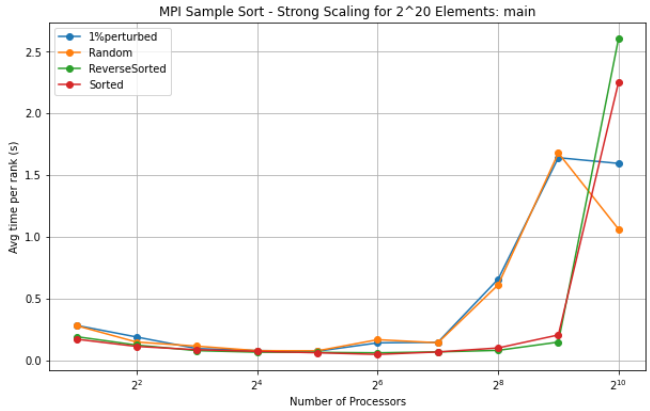
main

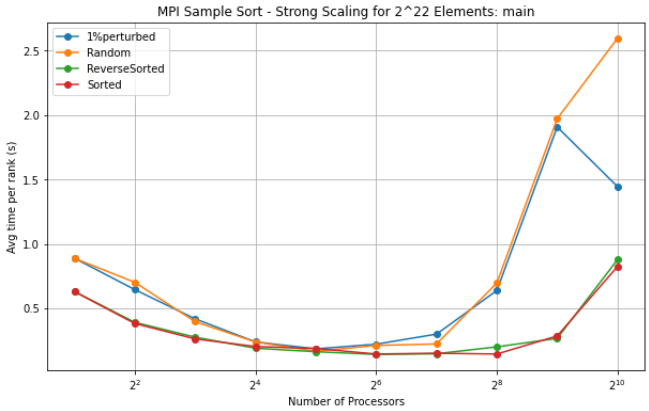
The strong scaling graphs for main tell conflicting stories. Looking at the lower input sizes, up to 2^22, the algorithm looks to have poor strong scaling performance as it is relatively flat for the beginning before spiking up as it gets closer to the end. However, looking at the larger input sizes, from 2^24 and up, the graphs start to display good strong scaling performance as they slope downwards and flatten off once a point of diminishing returns is reached, around the 2^6 or 2^7 number of processes mark. This behavior is caused

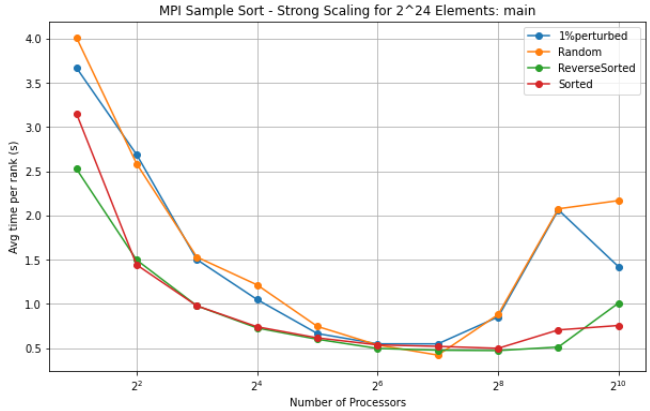
by the algorithm shifting from being communication-bound in the smaller input sizes to being computation-bound for the larger ones.

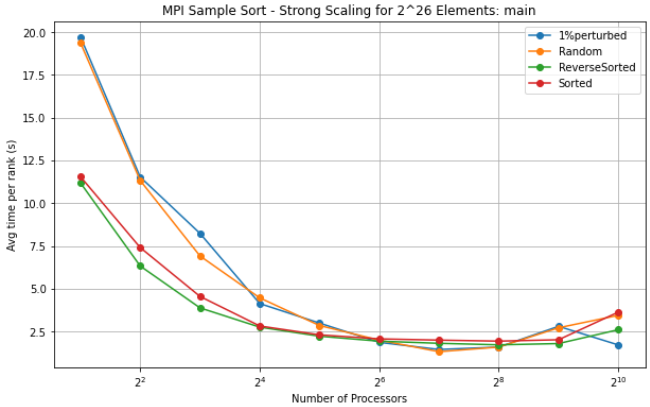


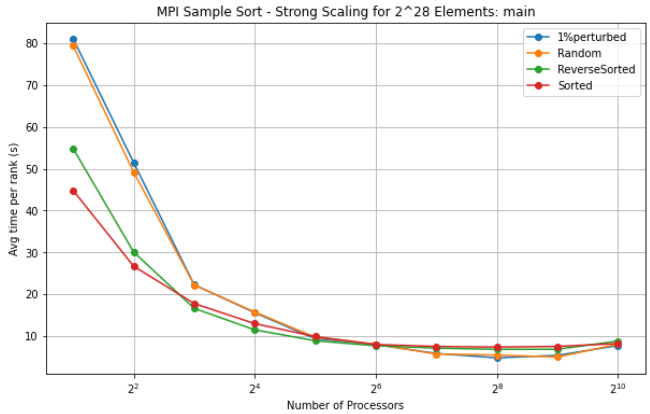








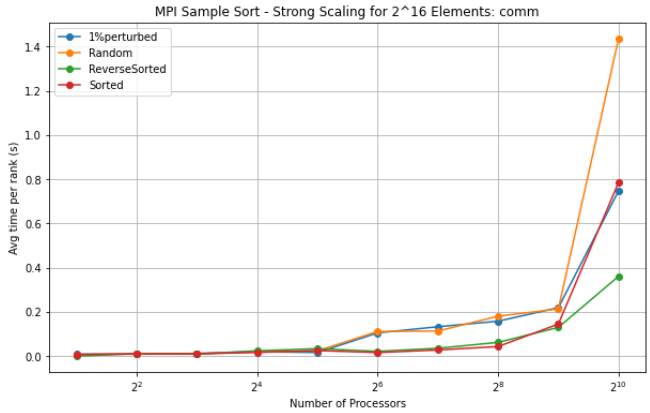


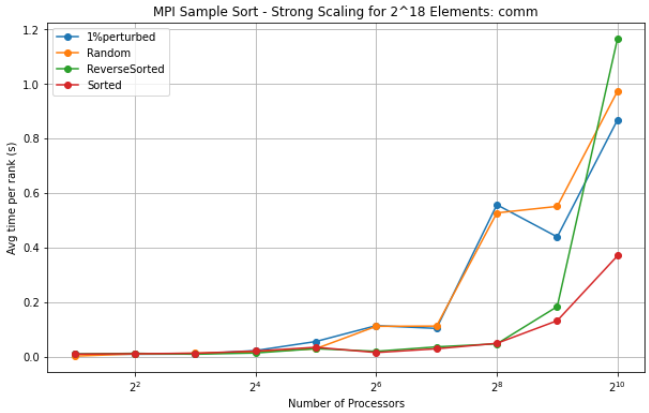


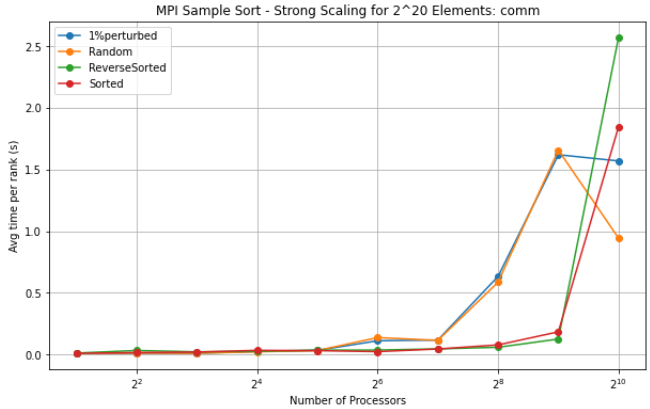
comm

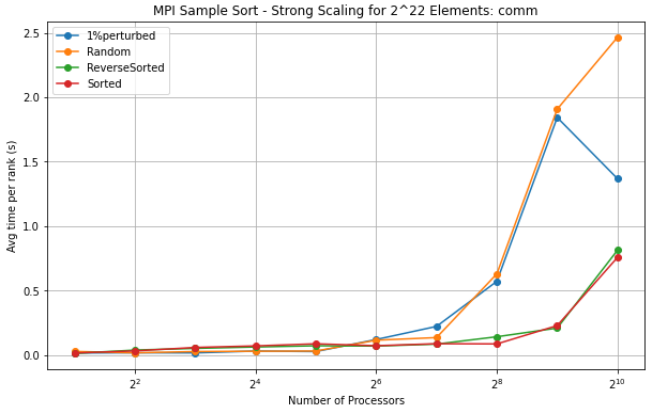
The communication part of this algorithm has poor strong scaling but does not make huge jumps in time as the computation does. Comm starts at less than 0.1 seconds for the smaller input sizes and takes up to a little

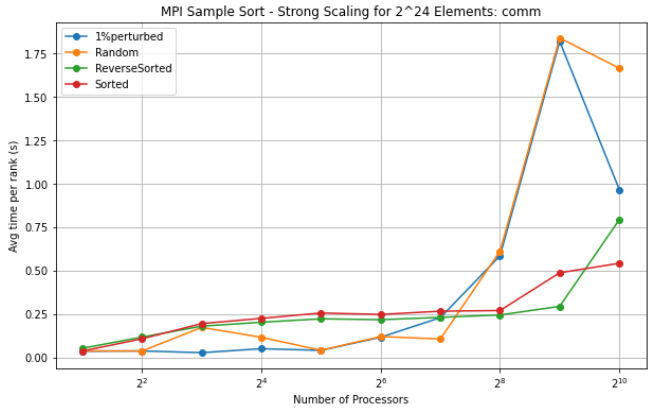
over 5 seconds for the largest input size. This means comm will dominate the overall runtime in the beginning, but then become less important as input sizes get larger. This matches what is seen in the graphs for main.

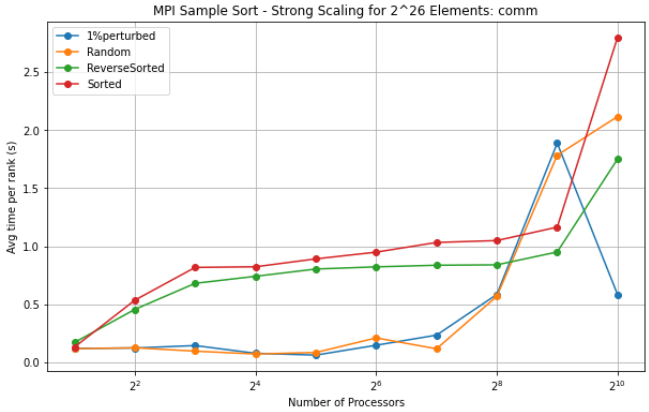


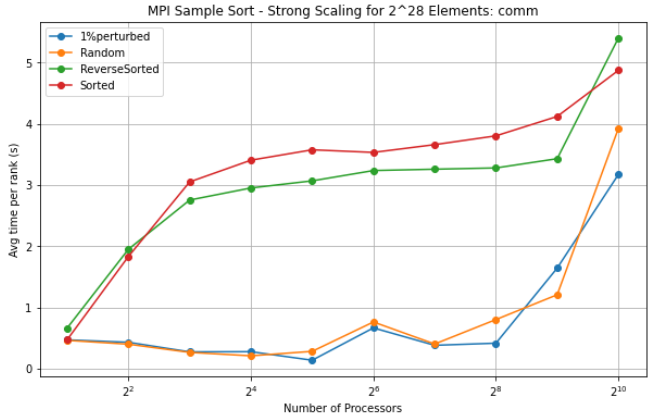






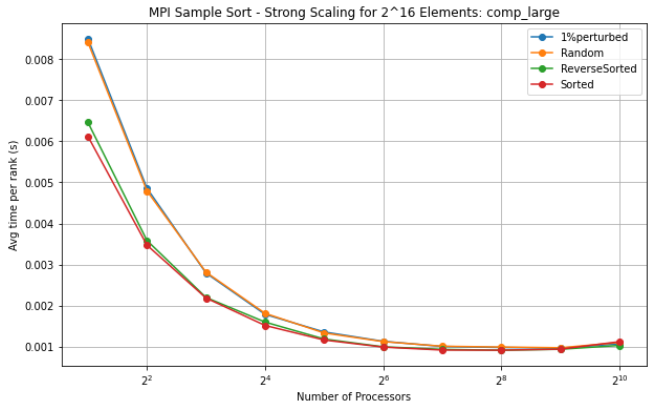


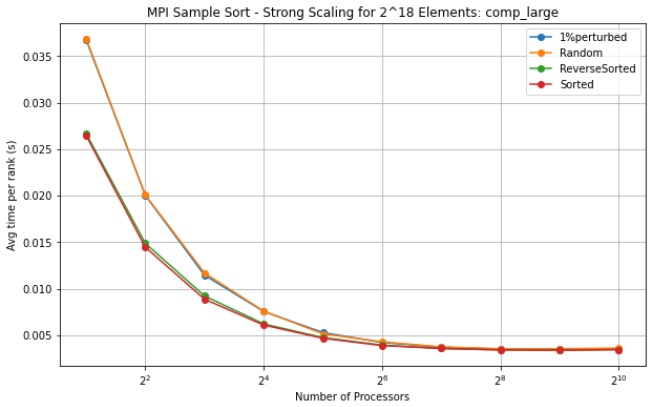


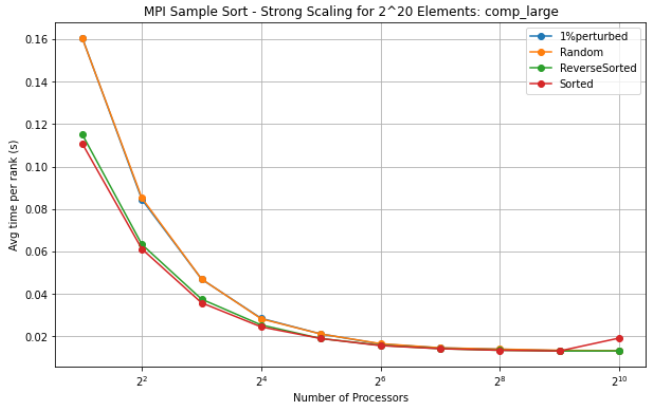


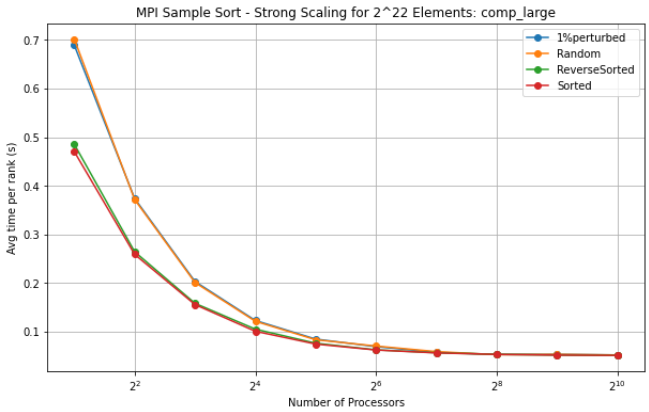
comp_large

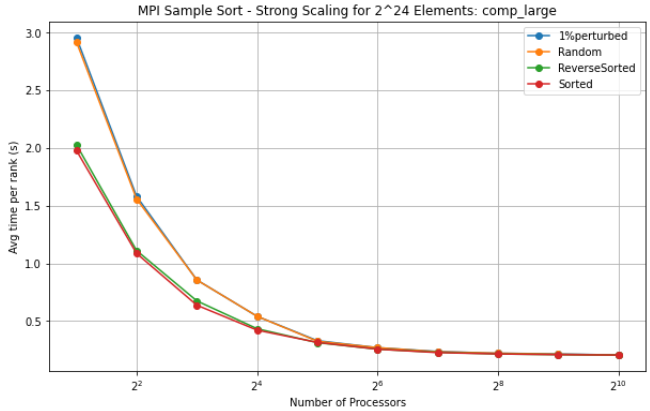
Comp_large has good strong scaling for all input sizes and hits a point of diminishing returns somewhere in the 2^6 to 2^7 processes range. As expected, the runtime grows from less than a second to around 50 seconds as larger input sizes are tested. This further explains why computation dominates the graphs of main only for the larger input sizes, as 2^24 is where computation begins to consistently take more time than communication.

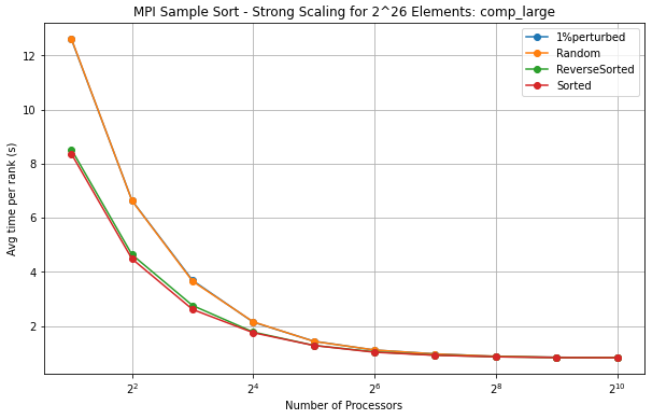


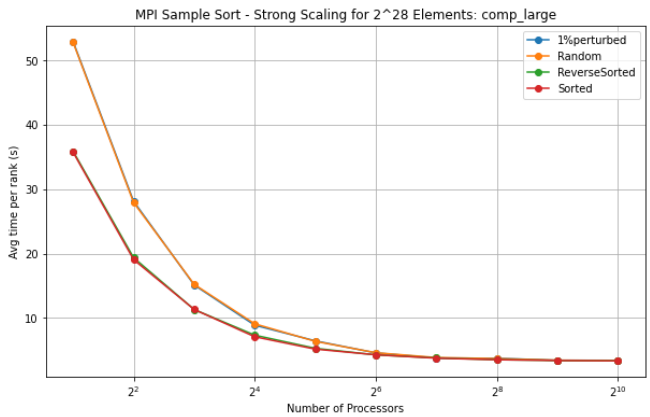








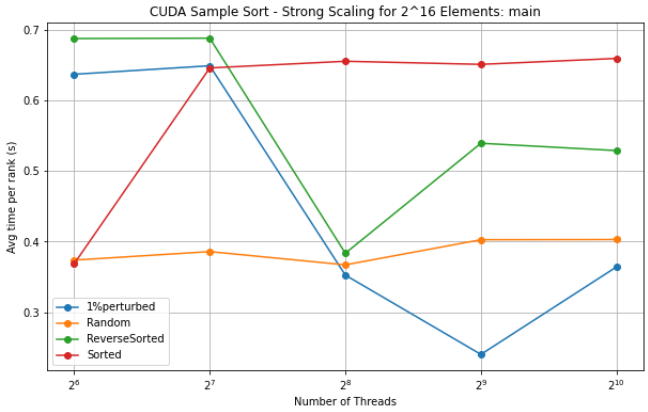


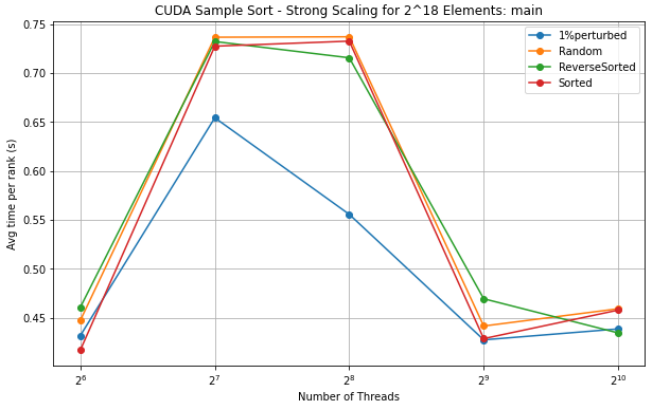


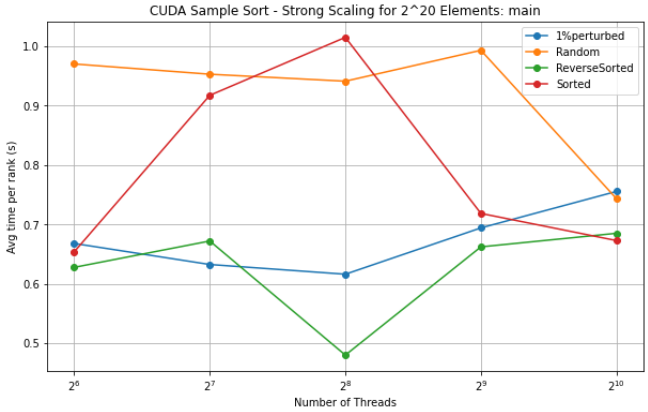
CUDA

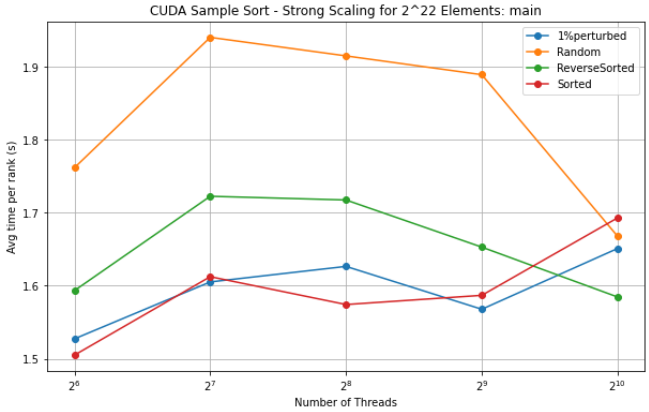
main

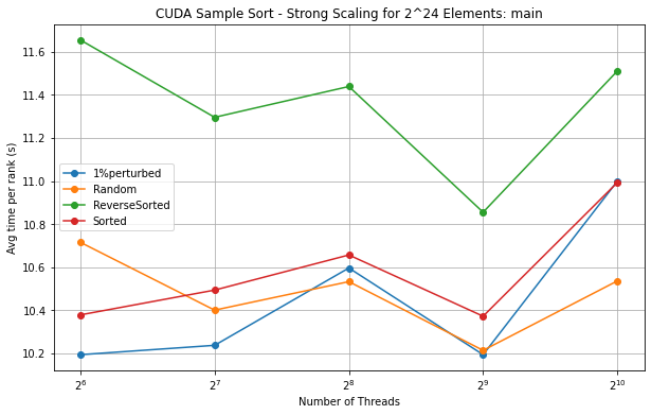
The strong scaling graphs for CUDA do not paint a very clear picture in either direction. The graphs are all over the place, but most tend to generally trend upwards overall, leading to a conclusion of poor strong scaling. This behavior is most likely due to the implementation not taking advantage of the GPU as much as it could. An interesting point to note when looking at these graphs, while the lines are very up and down, relative to the overall time taken, the range on the Y-axis is not that big.

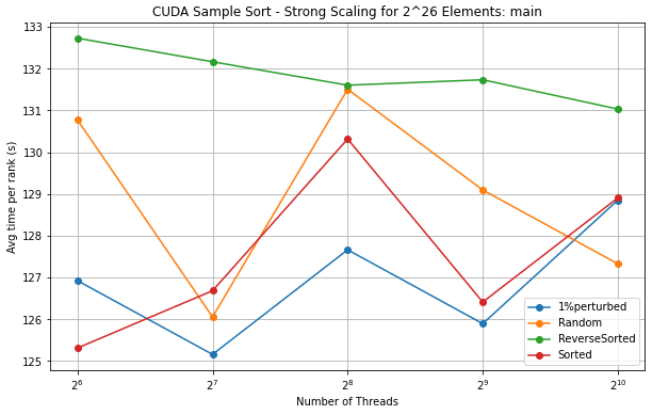


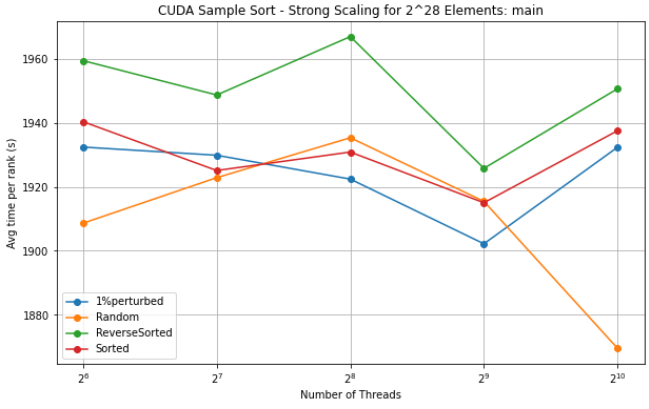






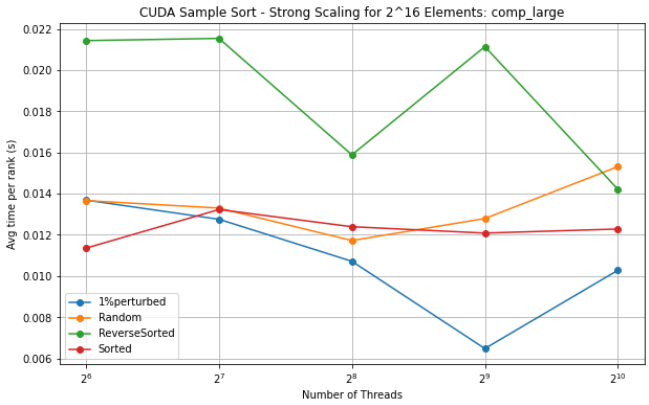


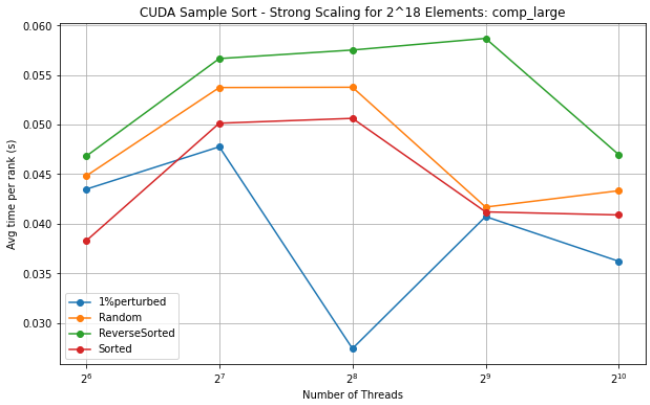


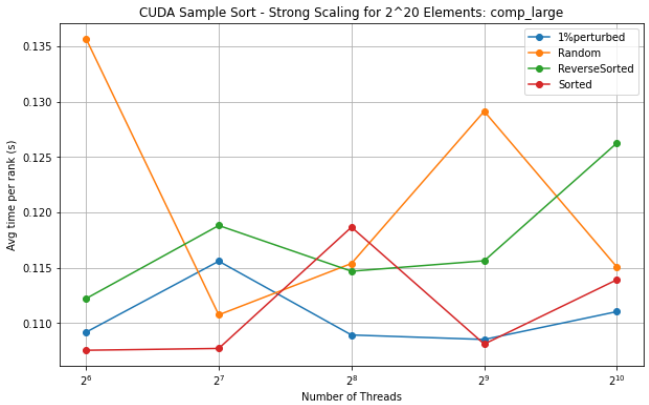


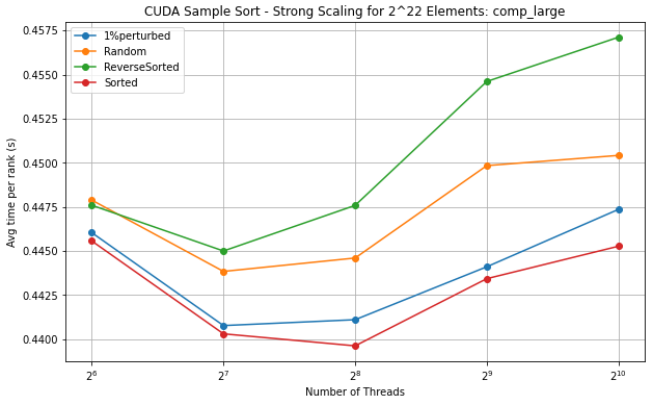
comp_large

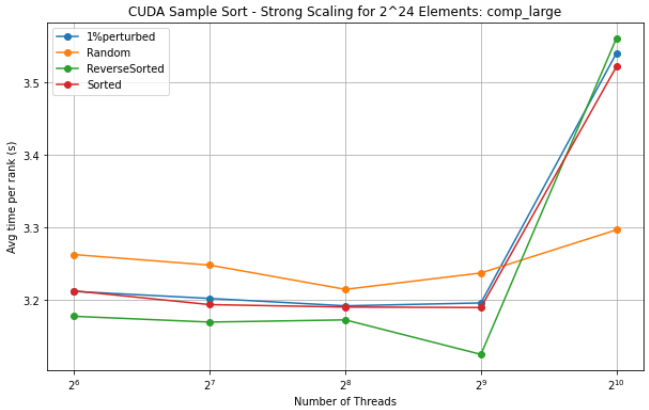
As CUDA did not have communication for the algorithm, comp_large looks fairly similar to main. While the 2^28 graph shape seems to be an outlier with the lines trending downwards, the rest generally trend upwards implying poor strong scaling performance. The up-and-down nature of these graphs is again most likely due to the algorithm not taking proper advantage of the GPU.

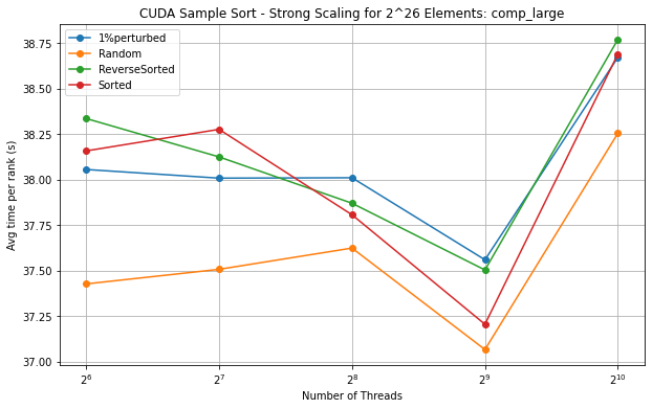


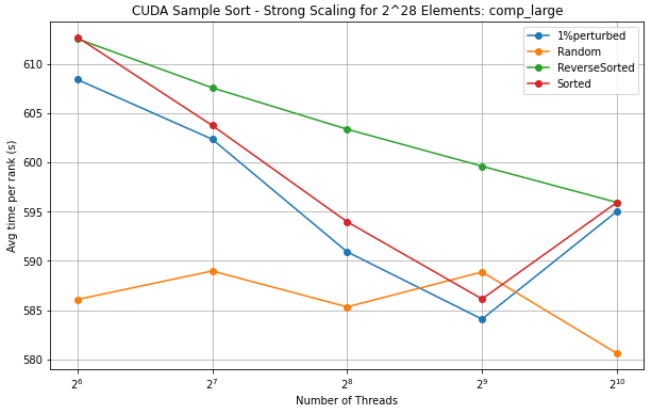










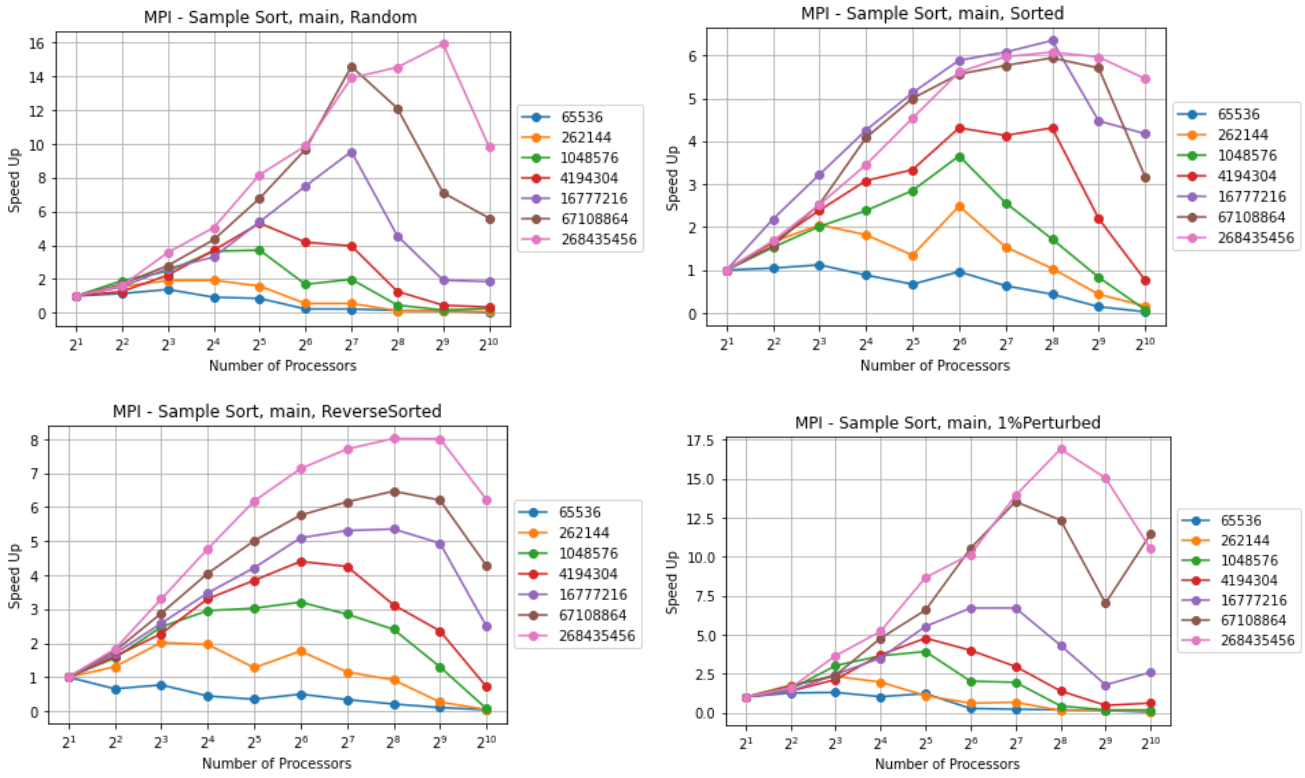


Speed up

MPI

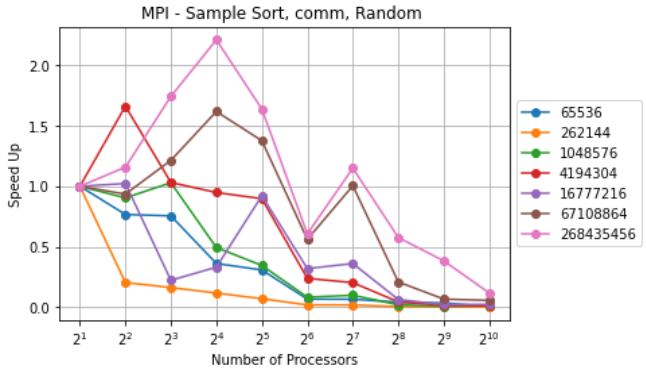
main

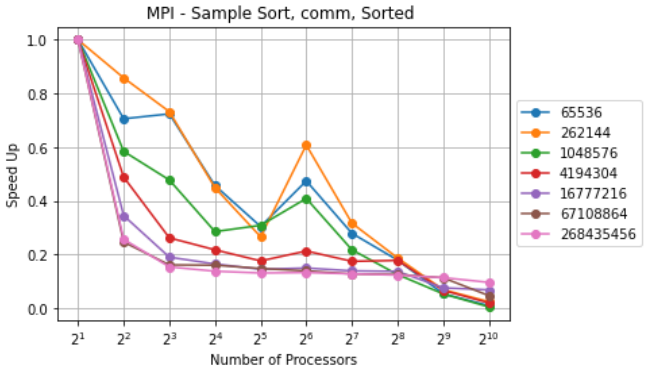
The MPI Speedup graphs for main have a similar pattern of switching what they are bounded by as seen in the strong scaling graphs, but reversed. Each one starts with a fairly good speedup shape as it is dominated by the speedup of computation in the early process counts, then starts to fall off as the communication overhead becomes greater and greater. As expected, the larger the input size, generally the better the speedup the algorithm sees.

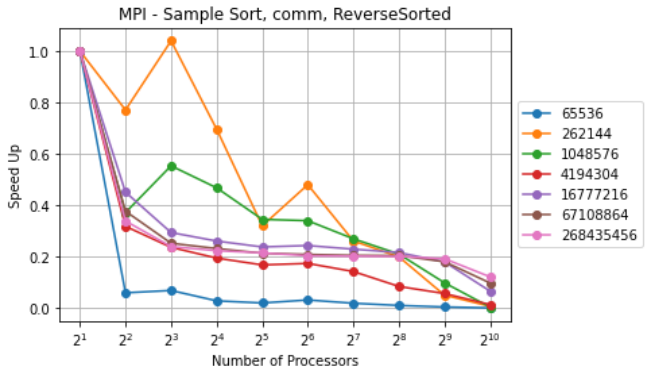


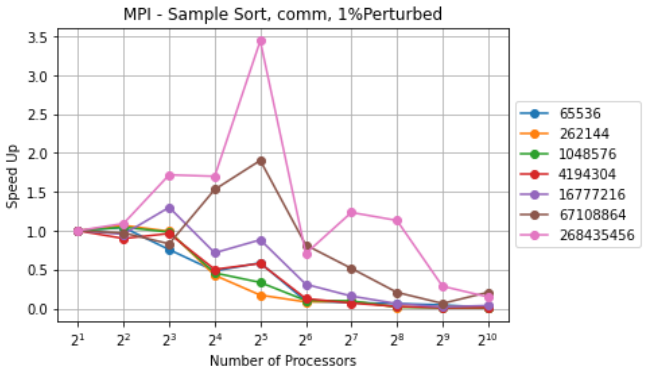
comm

Communication speedup graphs tell an expected story when considering the region has poor strong scaling performance. It starts off around 1.0 for the smaller input sizes but always ends up nearing 0 by the time the largest input size is reached.



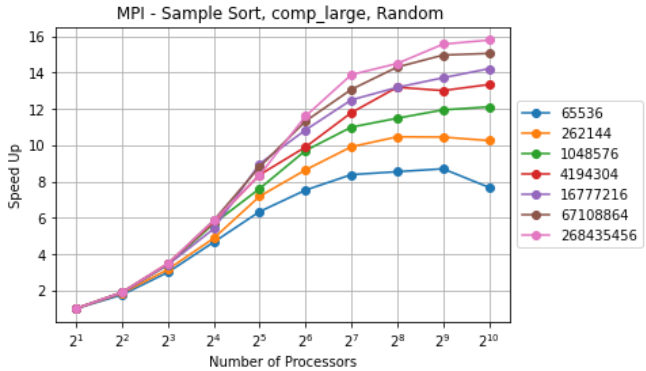


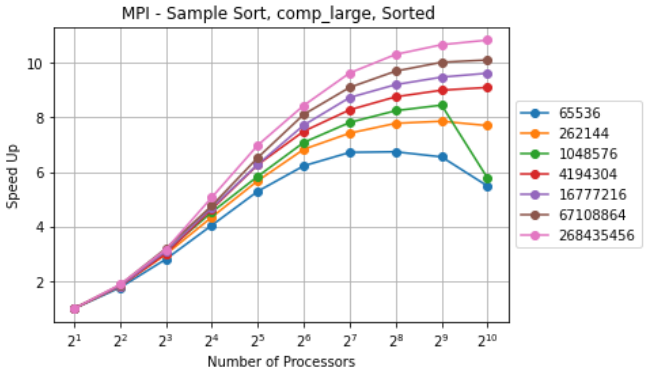


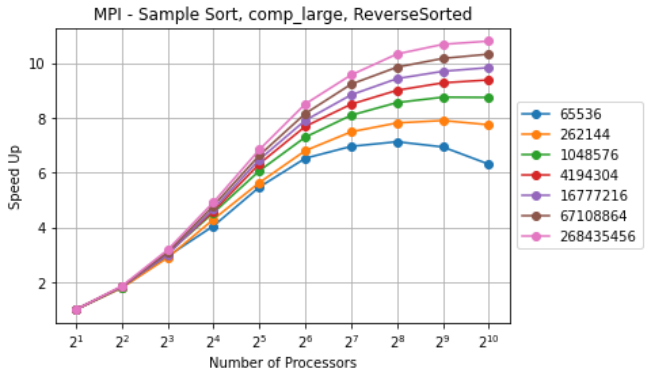


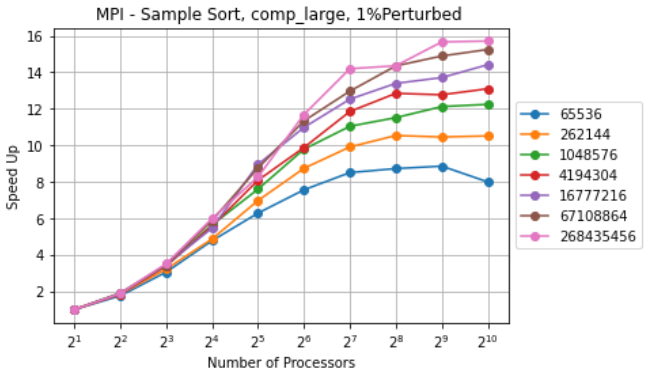
comp_large

The comp_large speedup graphs are almost picture-perfect when talking about speedups found in the real world. All input types see large speedups in the beginning as more processes are added for small numbers, but then plateau to various degrees once it reaches the point of diminishing returns at around 2^7 processes. As expected, the larger input sizes tend to achieve better speedups as they allow the algorithm to better utilize the maximum allowed resources.





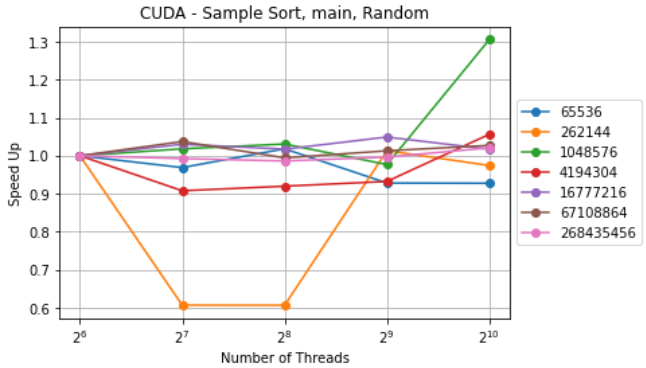


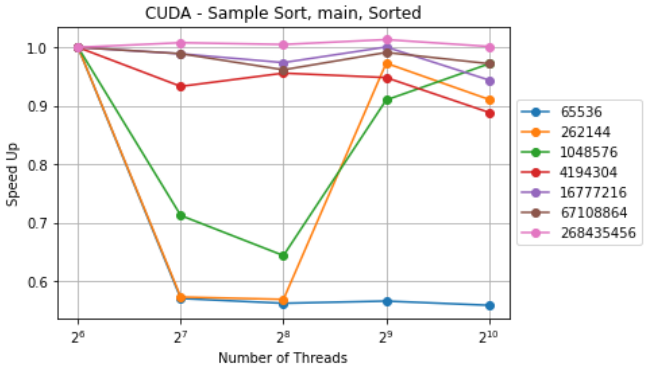


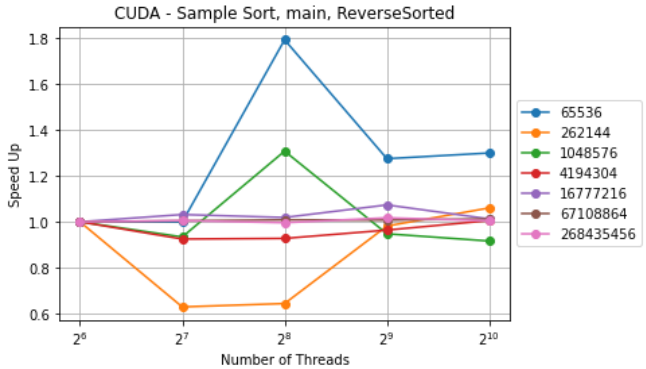
CUDA

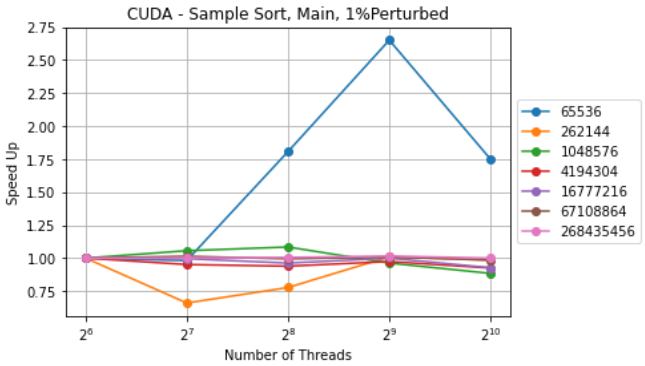
main

The CUDA speedup graphs for main show an interesting pattern of the larger input sizes hugging the 1.0 line and the smaller inputs bouncing around to a greater degree. This most likely means there are some small variations in the runtime no matter the input size and it is just more noticeable on the smaller sizes because their overall shorter times make them more susceptible to change.



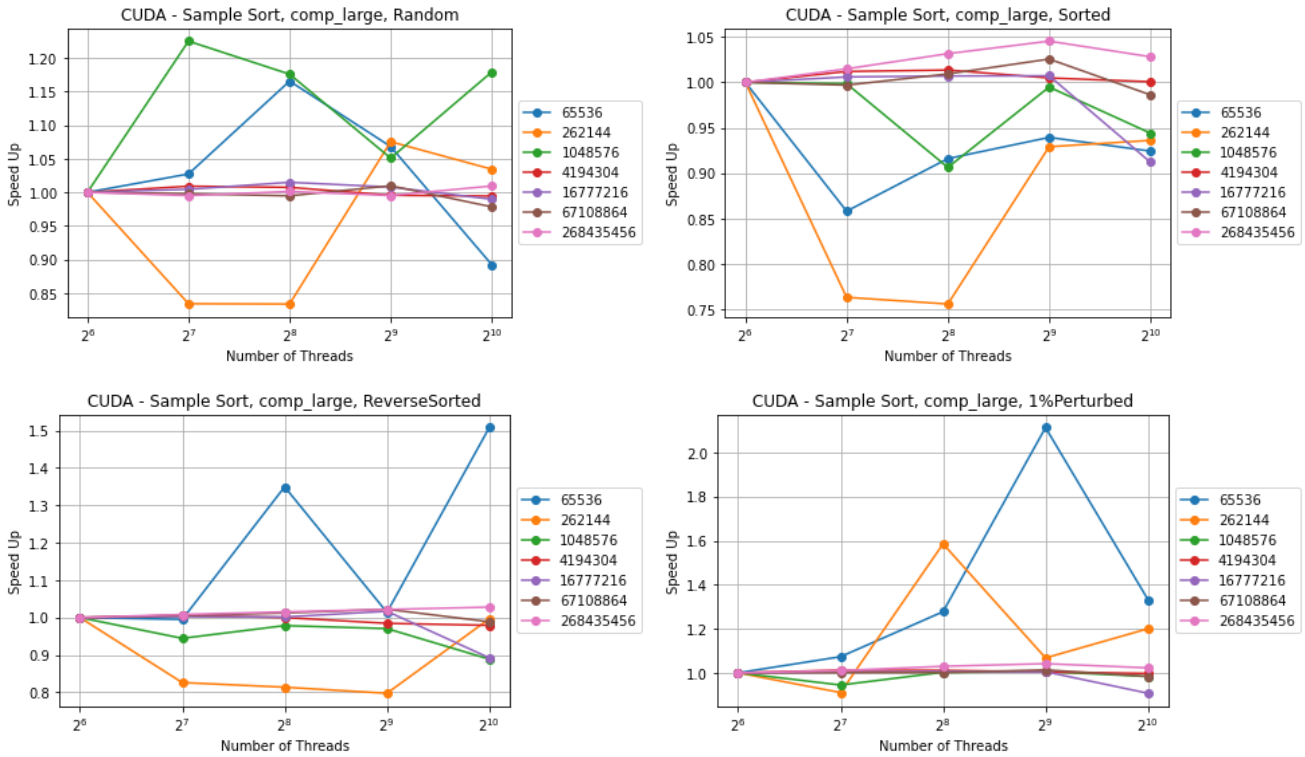






comp_large

The comp_large graphs are similar to the main graphs, but there seems to be a little bit more randomness. This most likely means the data initialization and correctness check sections of the program also have unclear runtime trends, therefore stabilizing the overall times a little bit, leading to the calmer graphs for main shown above.



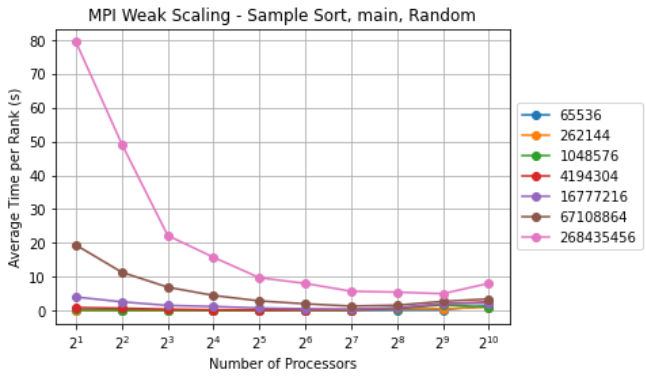
Weak Scaling

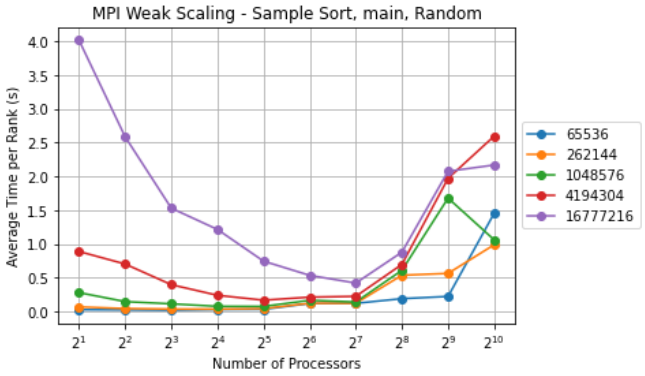
Note: Due to the longer runtimes of larger input sizes, the graph shape for the smaller inputs gets lost. To combat this, the left column of graphs has all input sizes and the right column has the same graph, but with the larger input sizes removed to enable analysis of smaller input sizes.

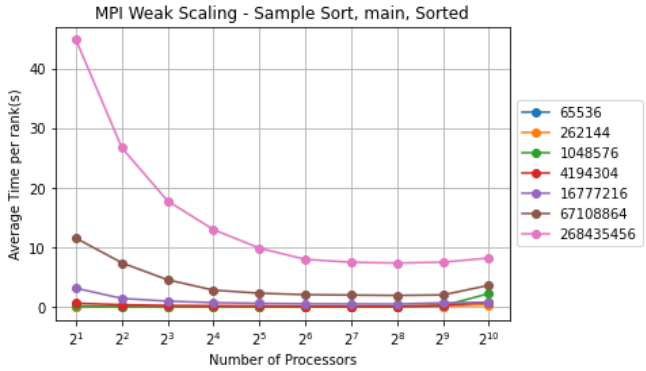
MPI

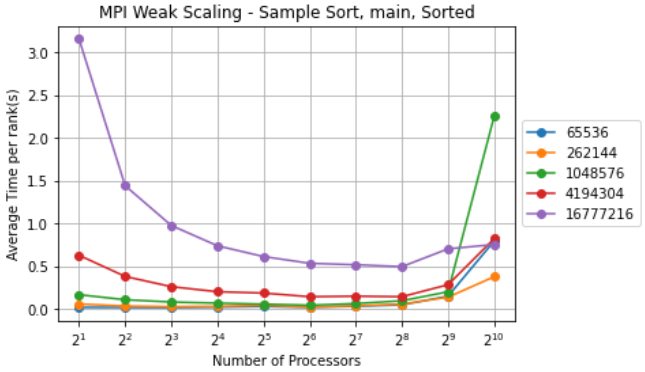
main

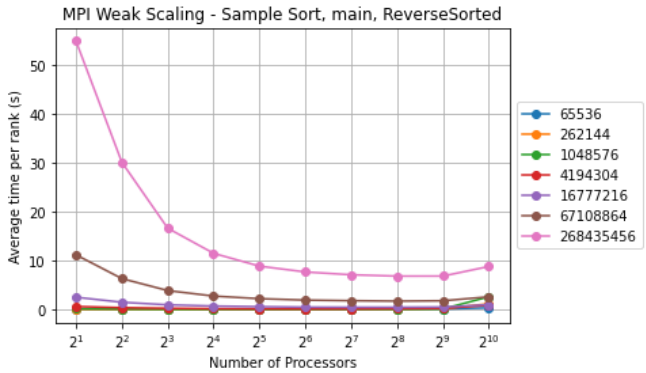
Weak scaling in main for the MPI implementation seems pretty good when looking at all input sizes. It is not until just the smaller input sizes are graphed that the upward spike near the largest number of processes can be seen. This is due to the communication overhead for larger number of processes that can be seen the comm graphs below. Prior to that, the runtime is dominated by the good weak scaling of the comp_large region.

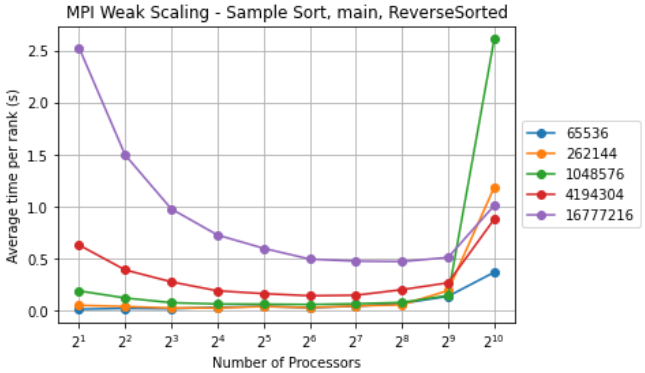


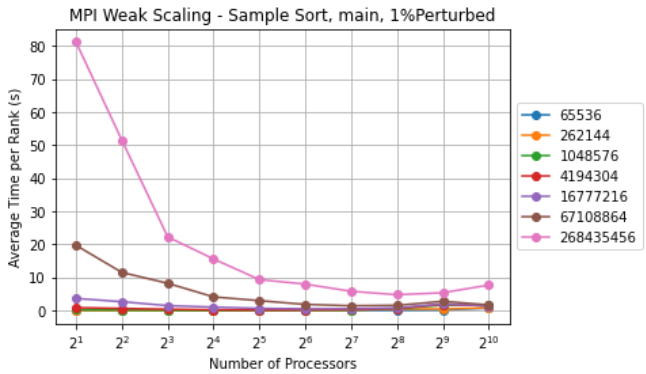


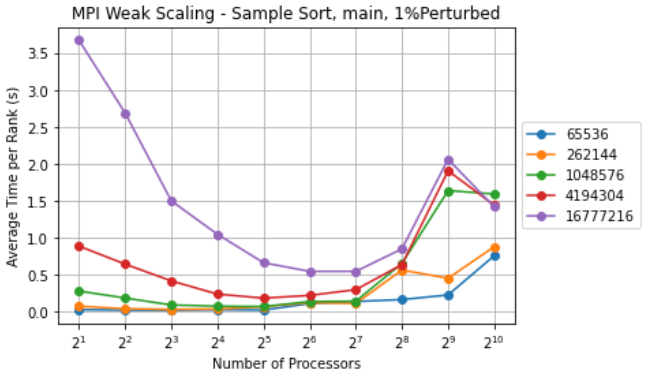






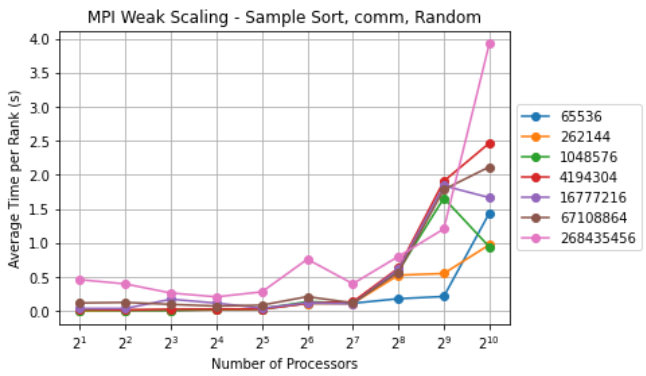


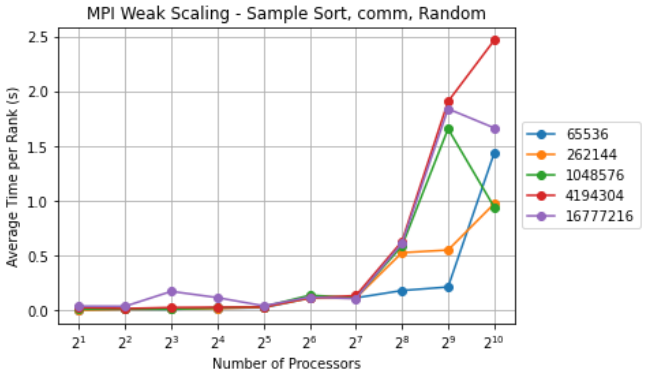


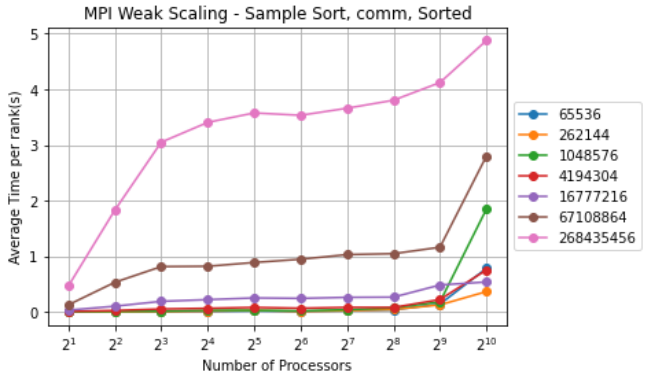


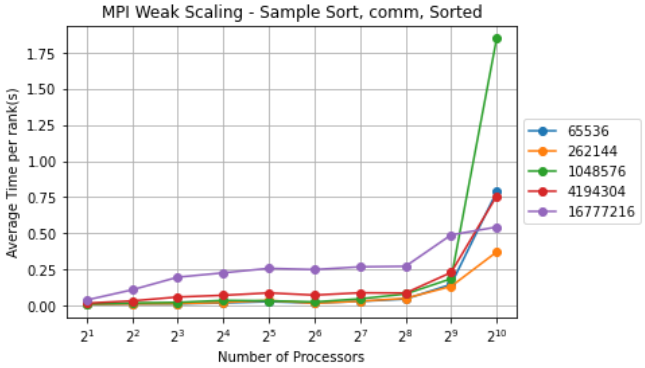
comm

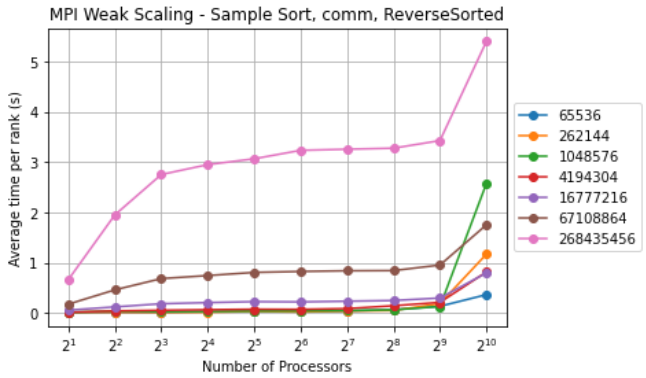
Graphs for the comm region, with the exception of the largest input size, seem to show decent weak scaling for lower numbers of processes. It is not until 2^6 or 2^7, when more than one node and therefore network communication is added, do the comm graphs start to show poor weak scaling and bleed into the main graphs.

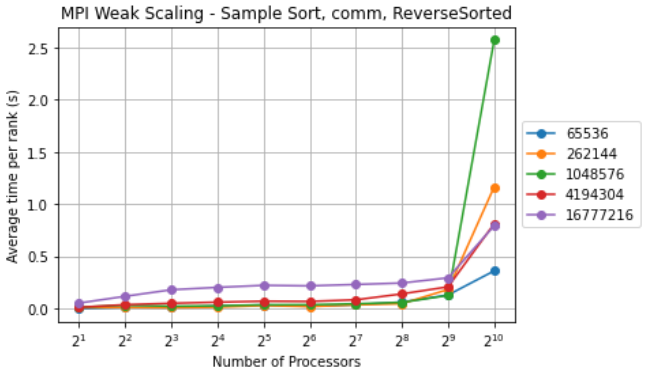


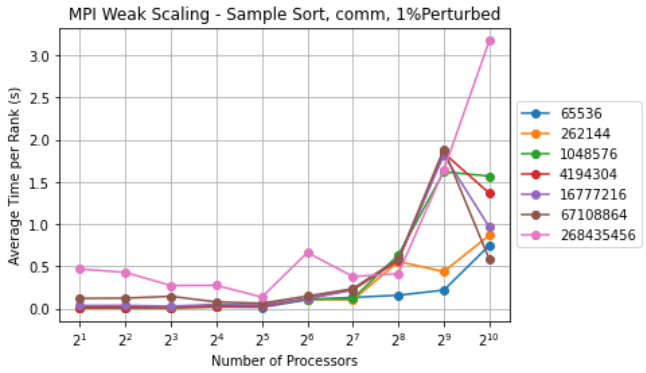


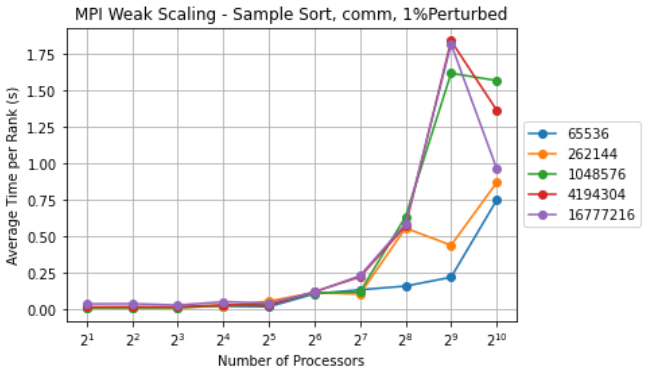






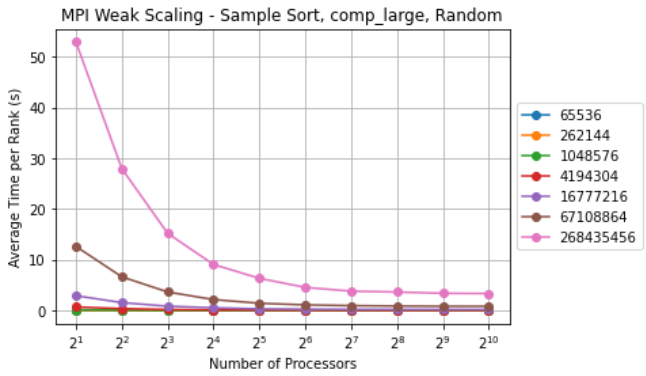


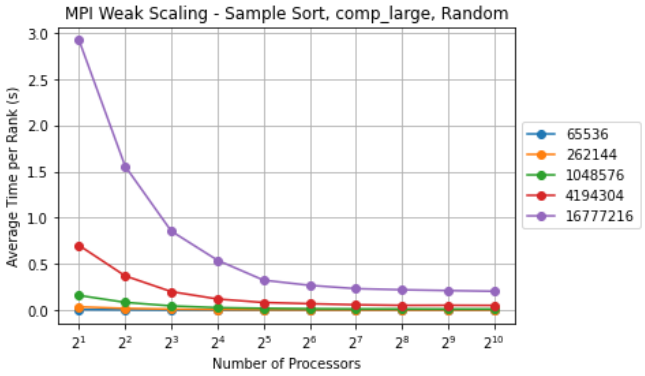


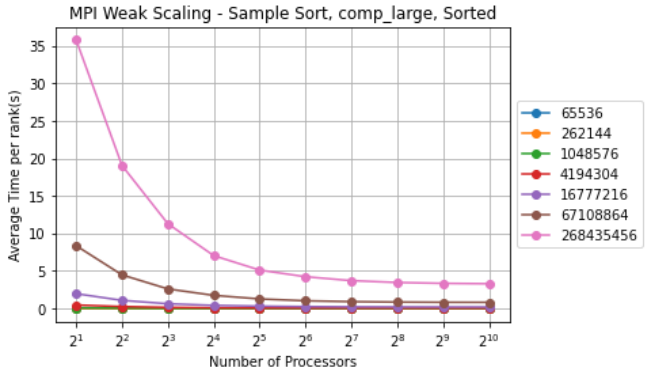


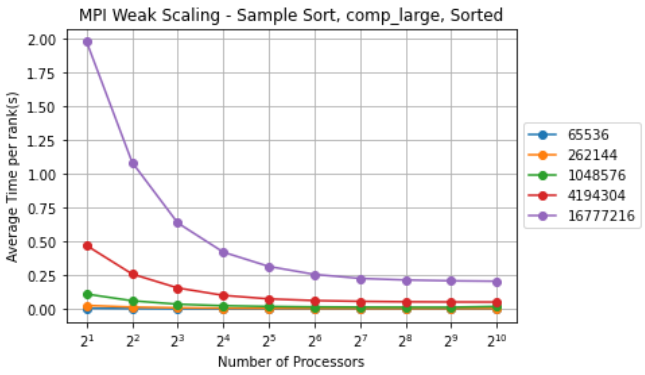
comp_large

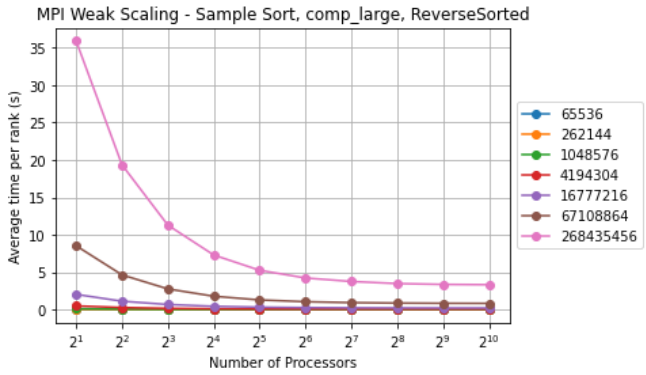
The graphs for comp_large show good weak scaling across the board for all input sizes as they have a steep drop at the beginning and then flatten out at the end. An interesting observation that can be seen on the graph of smaller input sizes is that sizes of 2^22 and under do not see as steep of an initial dropoff as the larger sizes do.

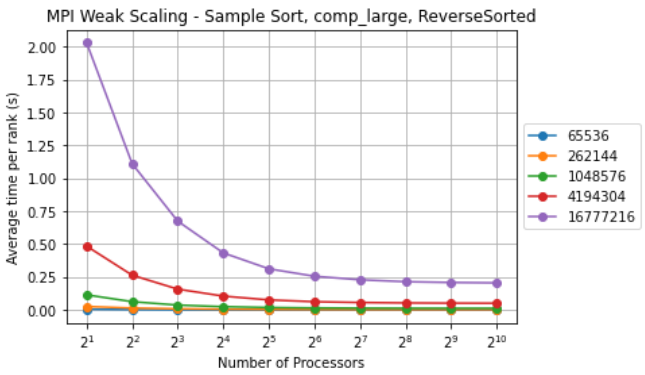


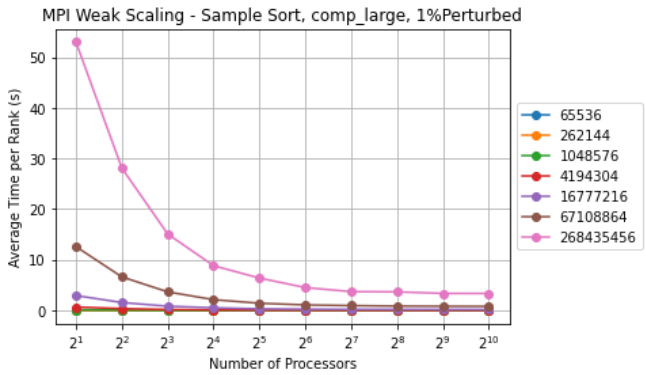


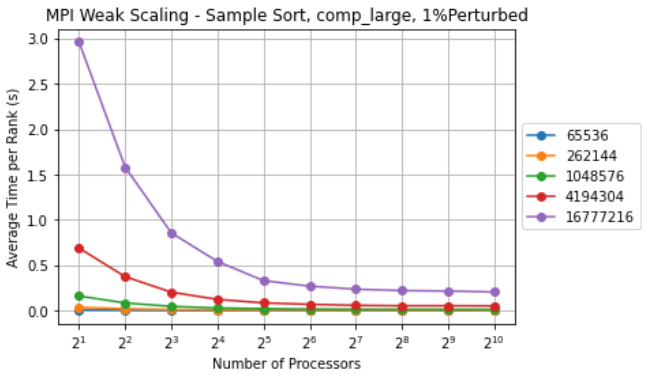








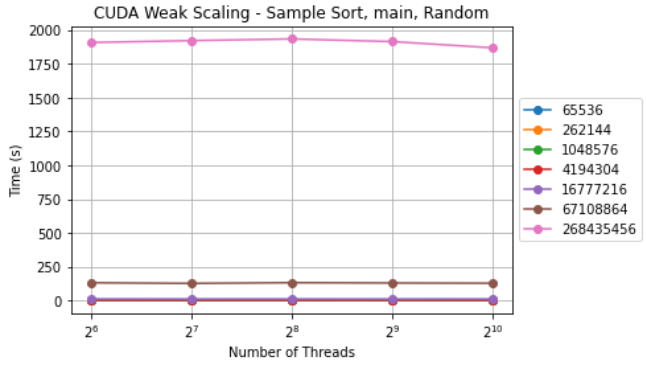


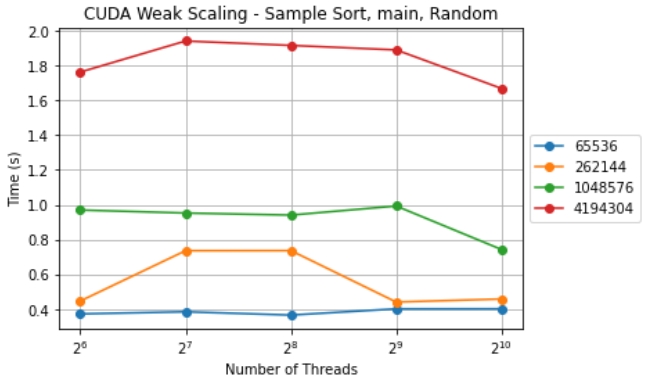


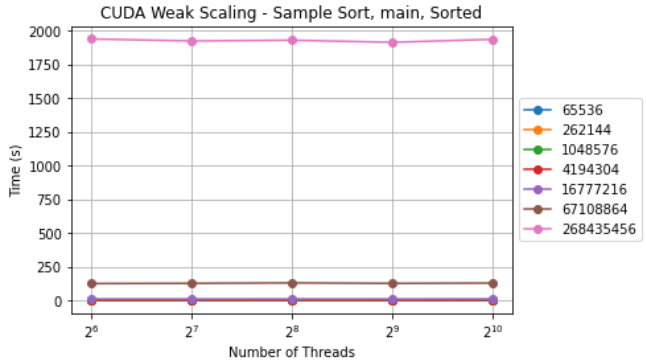
CUDA

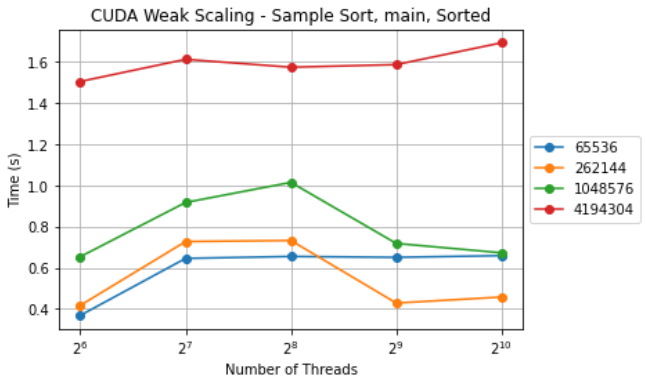
main

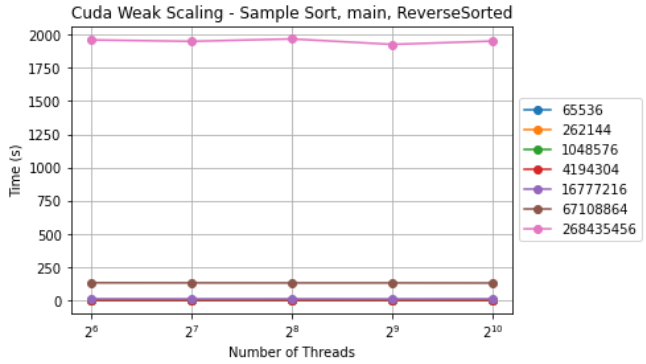
The CUDA weak scaling graphs for main show fairly flat lines overall on the graphs with larger input sizes implying good weak scaling performance. When looking at just the smaller input sizes, while a bit more variation can be seen, the Y-axis scale is much more narrow so small ups and downs are not surprising.

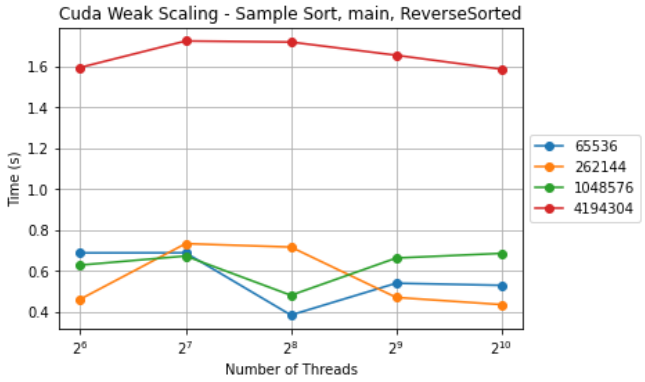


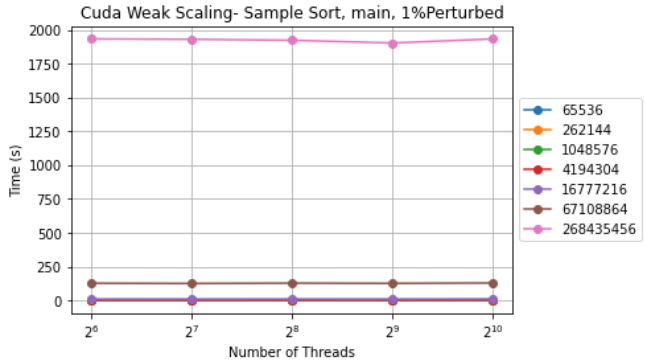


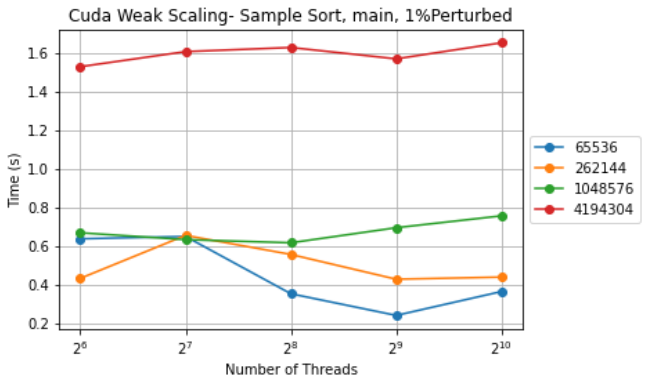






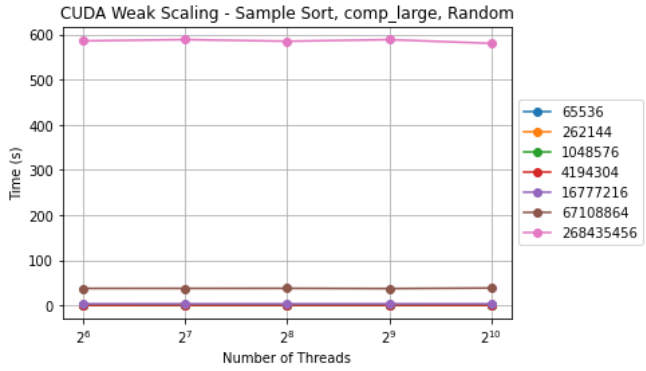


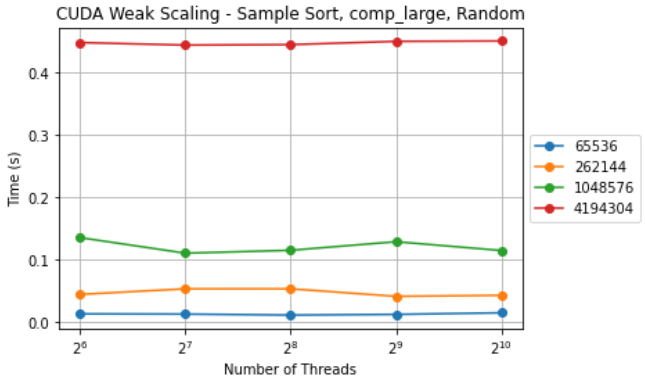


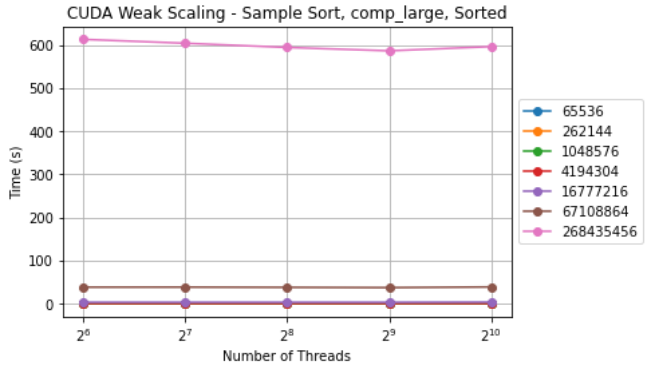


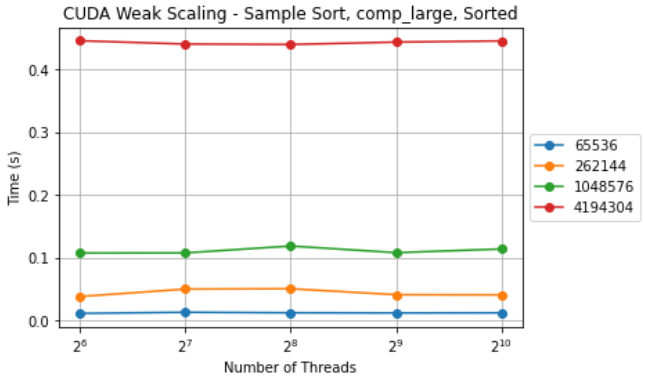
comp_large

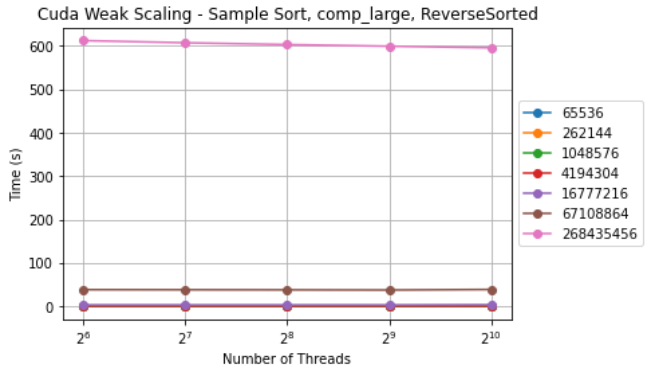
Weak scaling comp_large graphs echo the story told by the main graphs with the exception of the smaller input sizes. These maintain much more of a flat line which means the variation found in the graphs above is due to either data initialization or correctness checking runtimes fluctuating, not the actual sorting computation time.

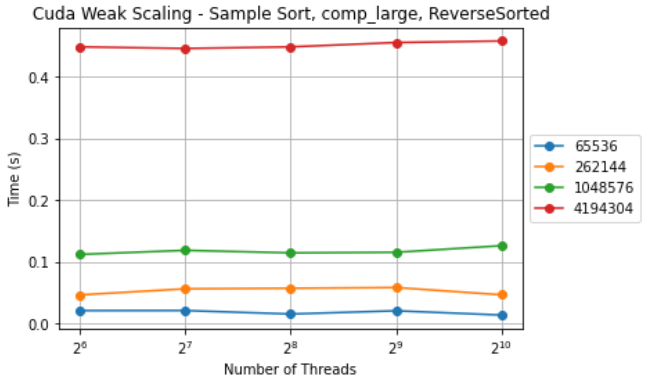


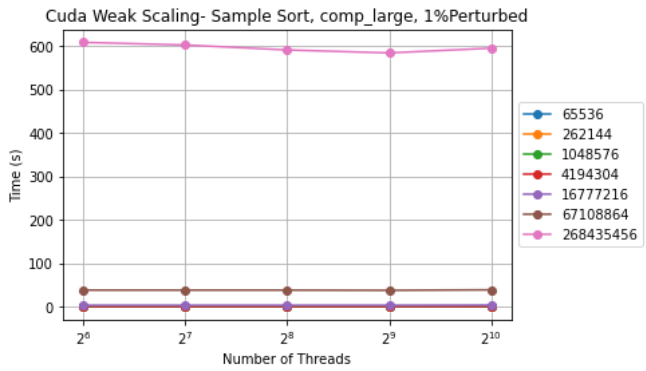


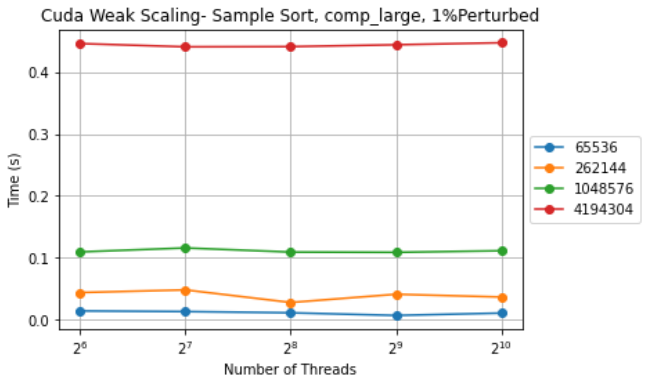












Merge Sort

Weak Scaling

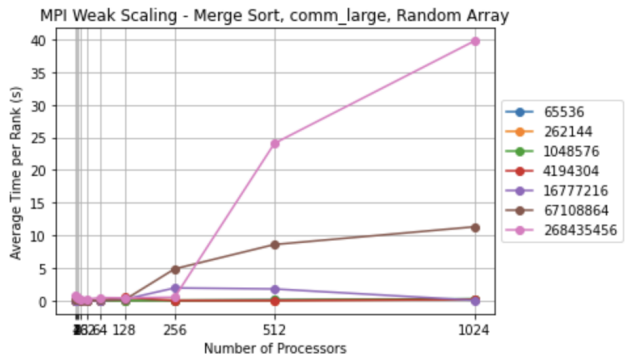
MPI

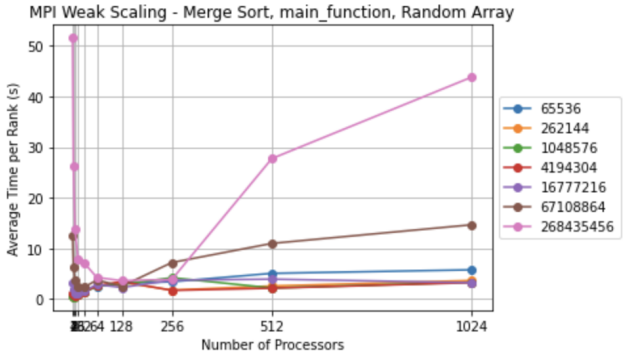
One thing to note when I did my weak scaling is that I chose to measure the average time per rank over number of processors. The reason I chose average time over total time is that disussing with the TA about how these sorts work, total time is always going to grow as you increase processors because its an aggregate

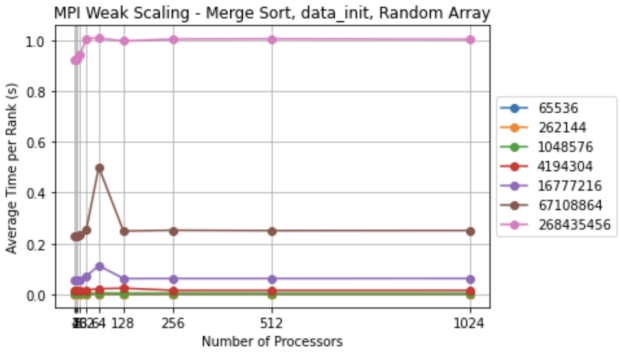
of all times over all processors. Additionally, I wanted to begin by focusing on the main function times for everything because I thought a good introduction to the analysis is how the program as a whole ran on average.

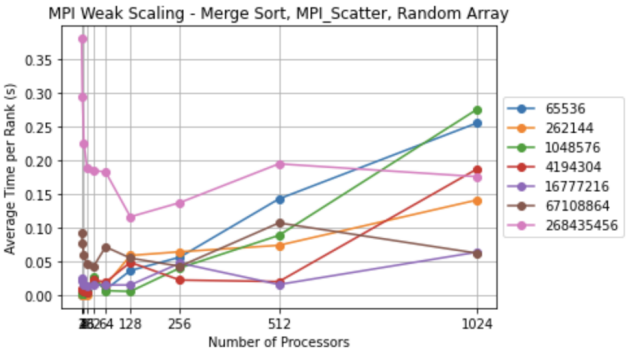
RANDOM INPUT ARRAY

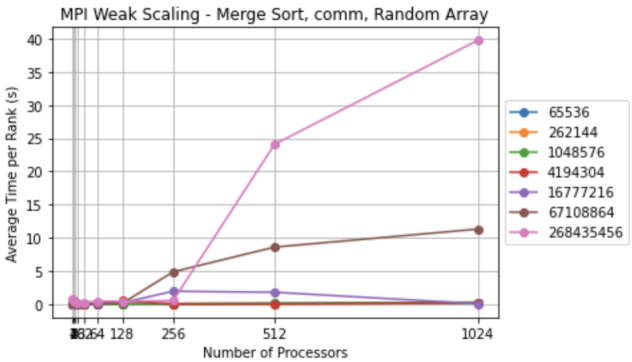
The analysis begins with weak scaling, specifically with MPI. For weak scaling, I began with the input type of a random array, where each element is a randomly generated number from 0 to n, where n is the size of the array. The most obvious trend between all of the graphs in this section is that the as you increase the number of elements, the algorithm performs more slowly. This is shown best with the largest input size of 2^28 having the largest graph, as signified by the pink line. It is interesting to note that the computation time for all the times looks relatively the same as we parallelize, and that most of the variation in times comes from the communication. I meausred both the MPI_Scatter and MPI_Gather functions, and the MPI_Gather function takes much longer than the scatter. In the sorted array input type, we see very similar behavior, where larger input sizes seemed to have a much longer time to complete. The same behavior is present in the reverse sorted array as well. Finally, in the data initialized with 1% of the data perturbed, we see the similar data. We can assume this due to merge sort typically always breaking the arrays down into a single element subarray and merging them back together, so we should expect relatively consistent behavior.

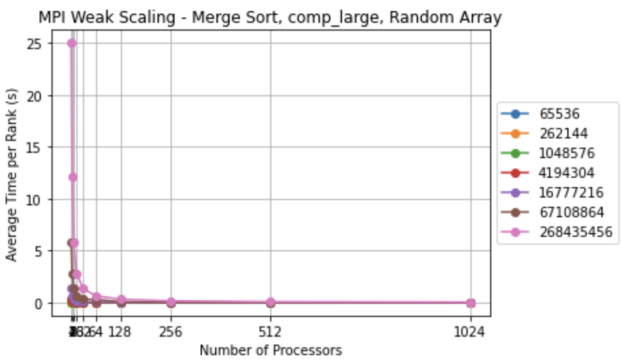


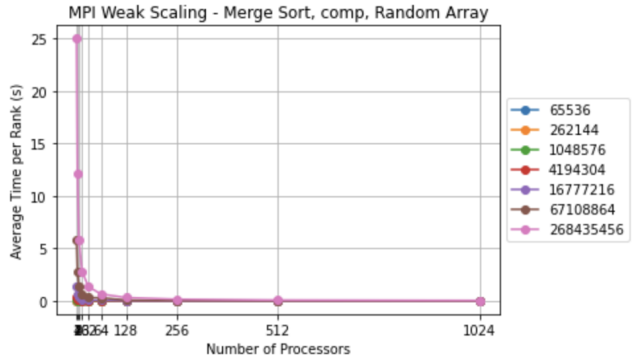


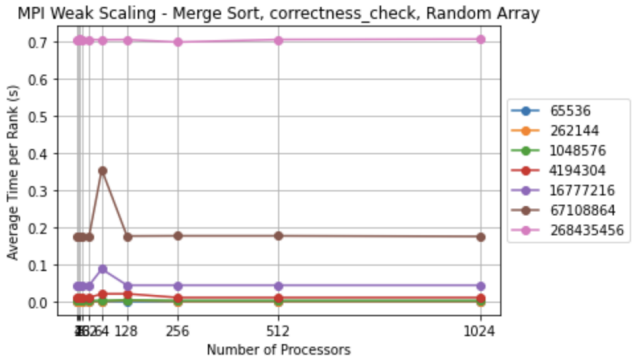


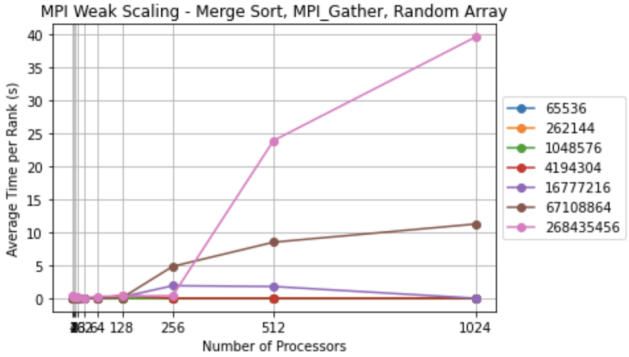




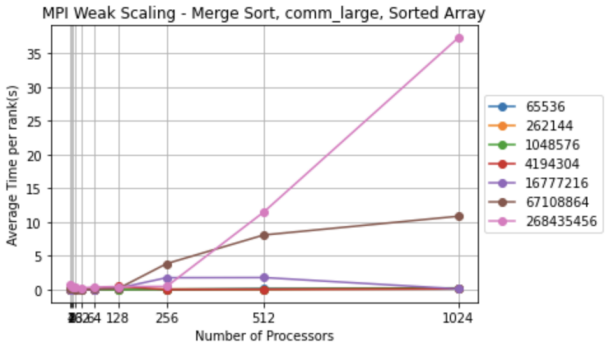


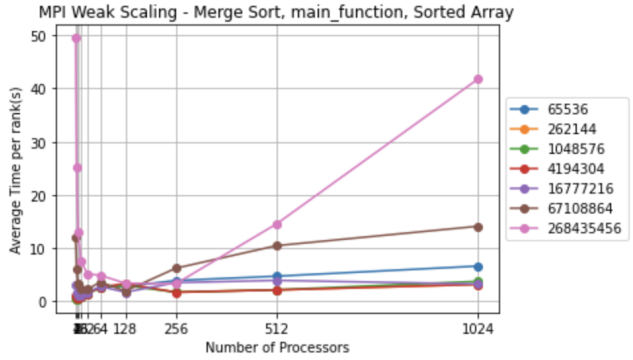


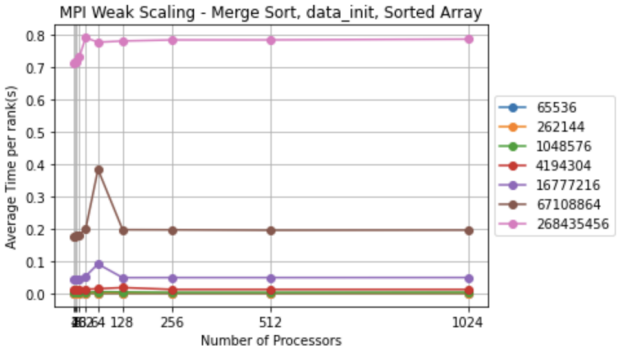


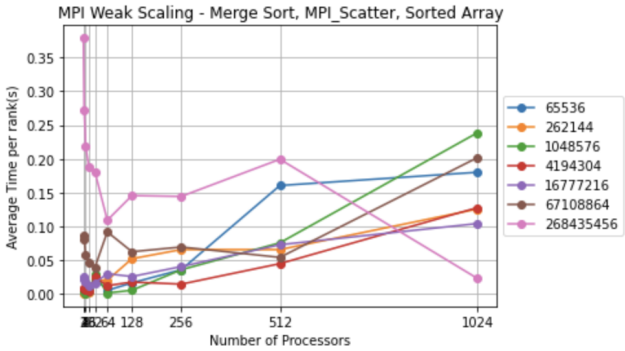


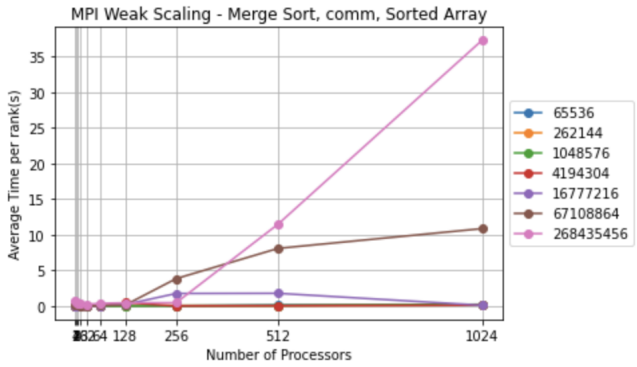
SORTED INPUT ARRAY

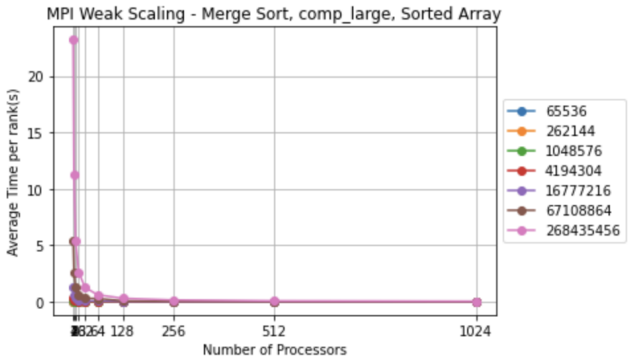


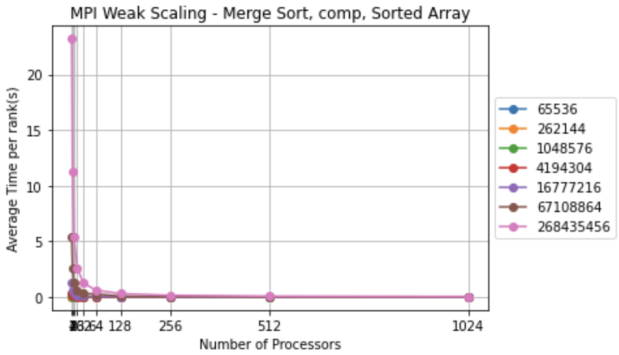


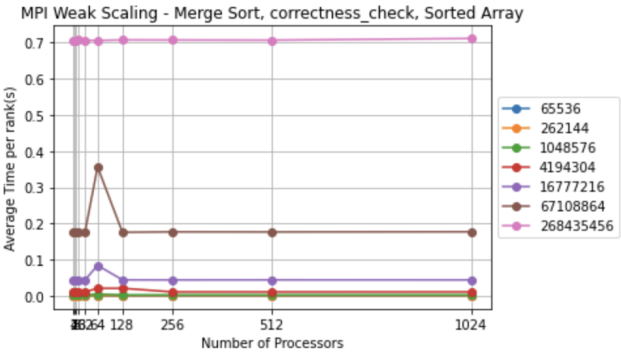


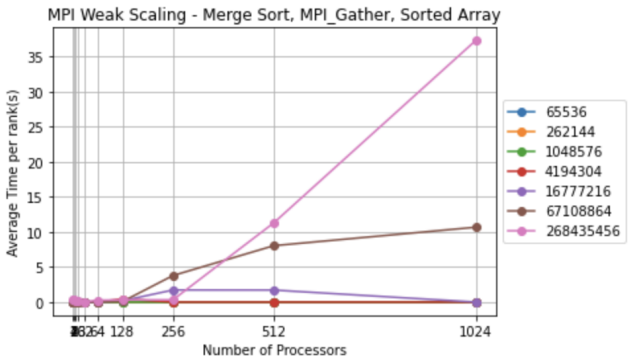




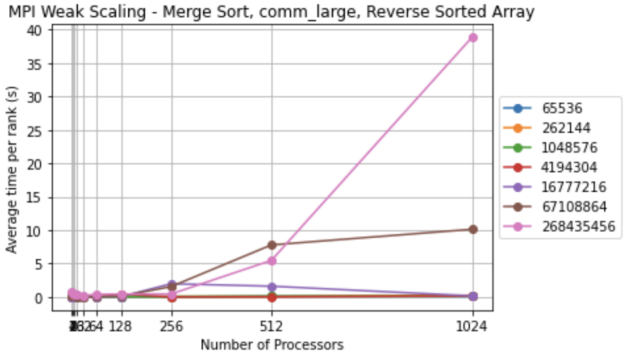


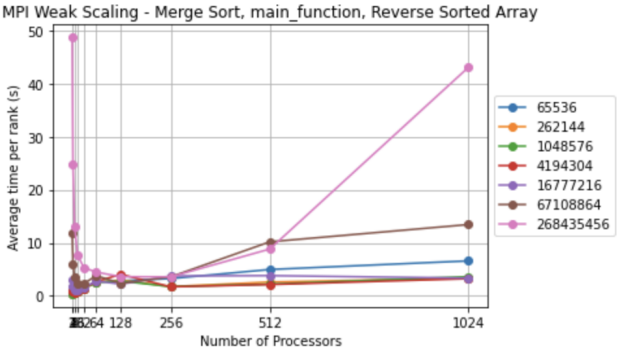


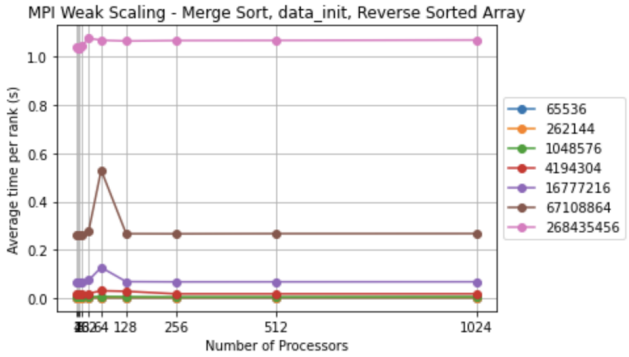


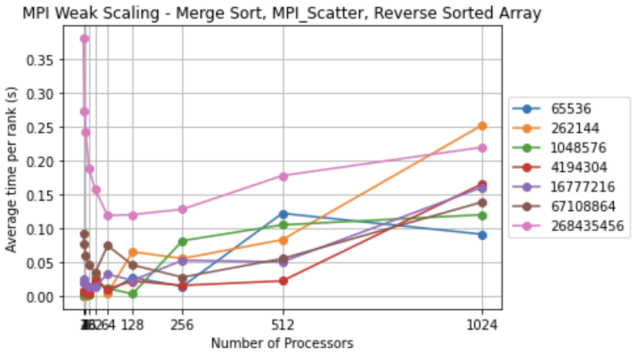


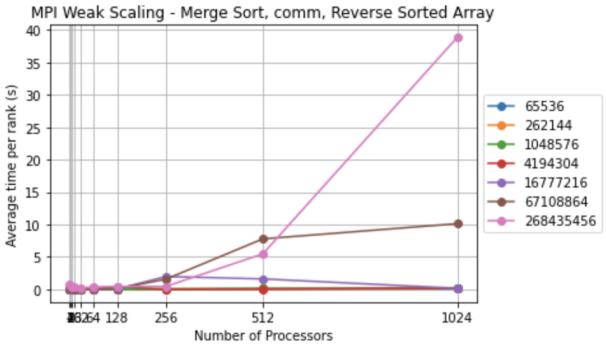
REVERSE SORTED INPUT ARRAY

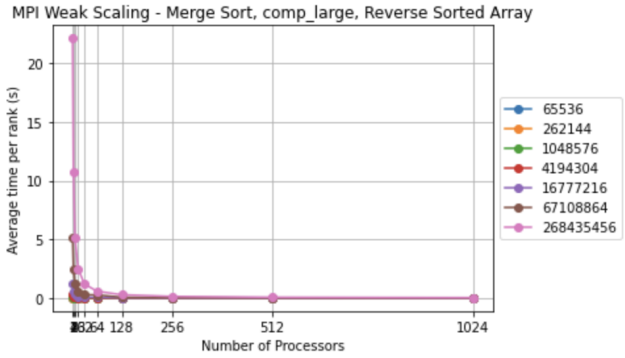


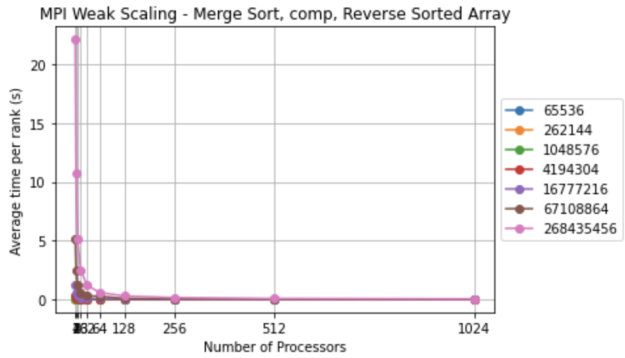


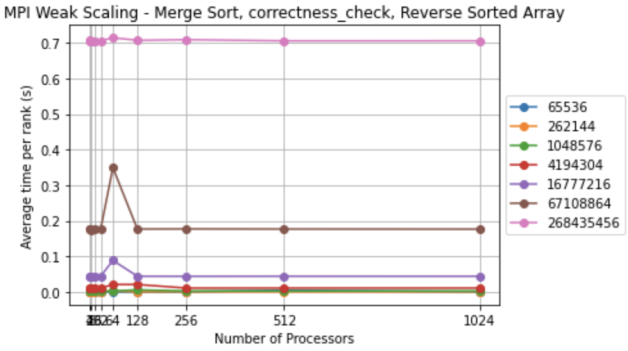


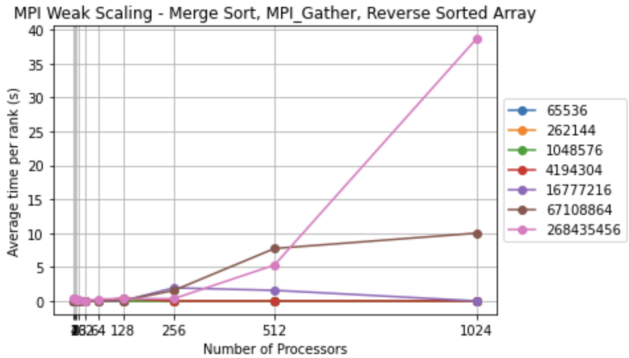




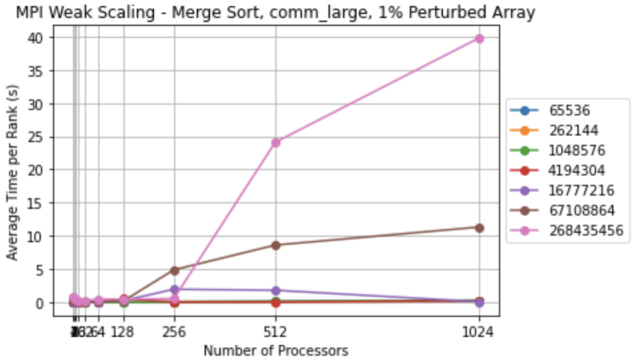


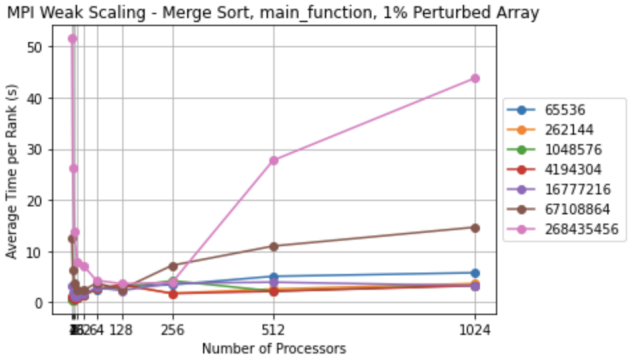


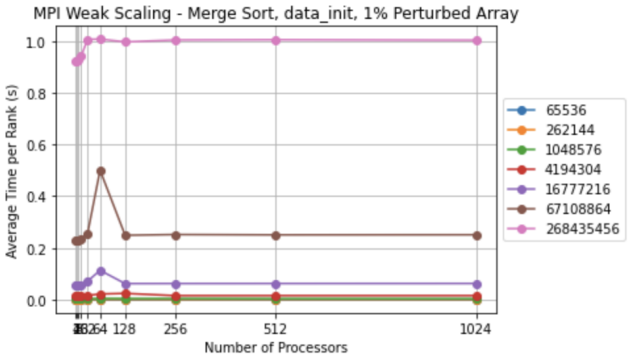


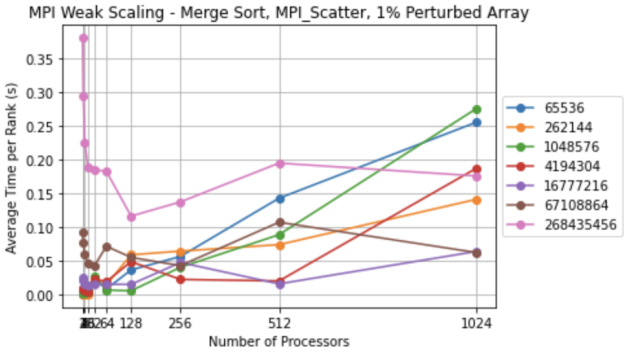


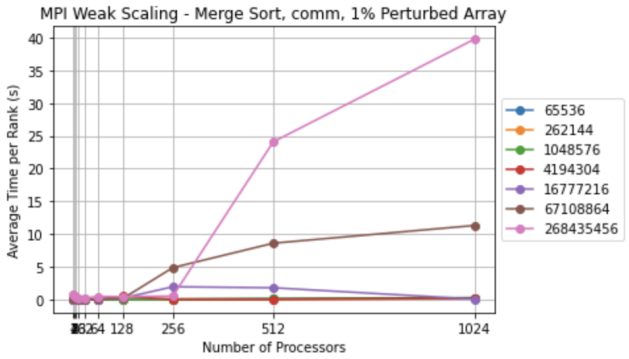
1% PERTURBED INPUT ARRAY

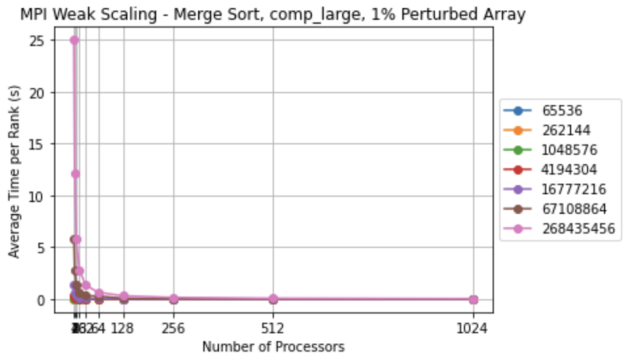

MPI Weak Scaling - Merge Sort, comm_large, 1% Perturbed Array
Average Time per Rank (s)
Number of Processors
65536
262144
1048576
4194304
16777216
67108864
268435456

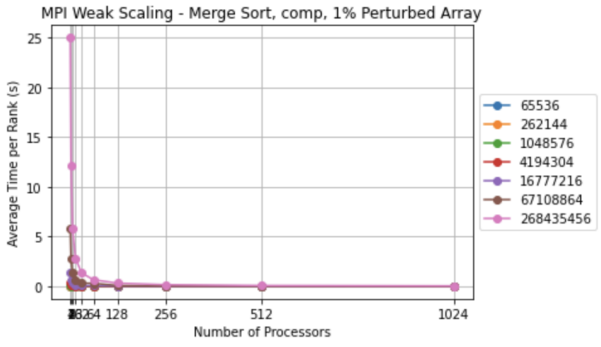

MPI Weak Scaling - Merge Sort, main_function, 1% Perturbed Array
Average Time per Rank (s)
Number of Processors
65536
262144
1048576
4194304
16777216
67108864
268435456

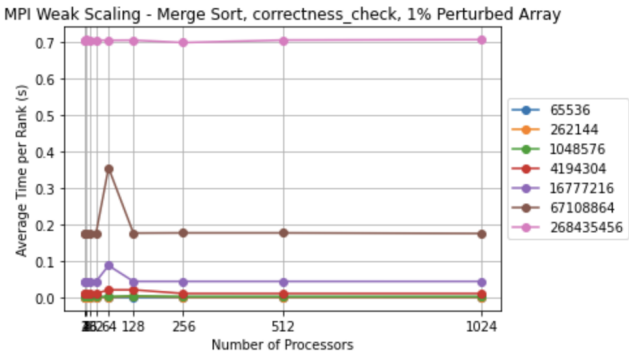

MPI Weak Scaling - Merge Sort, data_init, 1% Perturbed Array
Average Time per Rank (s)
Number of Processors
65536
262144
1048576
4194304
16777216
67108864
268435456

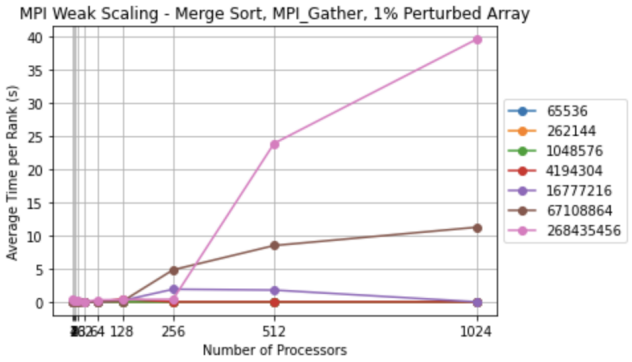

MPI Weak Scaling - Merge Sort, MPI_Scatter, 1% Perturbed Array
Average Time per Rank (s)
Number of Processors
65536
262144
1048576
4194304
16777216
67108864
268435456


MPI Weak Scaling - Merge Sort, comm, 1% Perturbed Array
Average Time per Rank (s)
Number of Processors
65536
262144
1048576
4194304
16777216
67108864
268435456


MPI Weak Scaling - Merge Sort, comp_large, 1% Perturbed Array
Average Time per Rank (s)
Number of Processors
65536
262144
1048576
4194304
16777216
67108864
268435456


MPI Weak Scaling - Merge Sort, comp, 1% Perturbed Array
Average Time per Rank (s)
Number of Processors
65536
262144
1048576
4194304
16777216
67108864
268435456

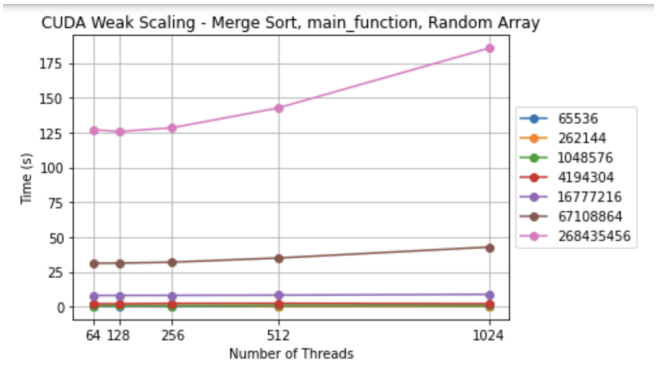

MPI Weak Scaling - Merge Sort, correctness_check, 1% Perturbed Array
Average Time per Rank (s)
Number of Processors
65536
262144
1048576
4194304
16777216
67108864
268435456

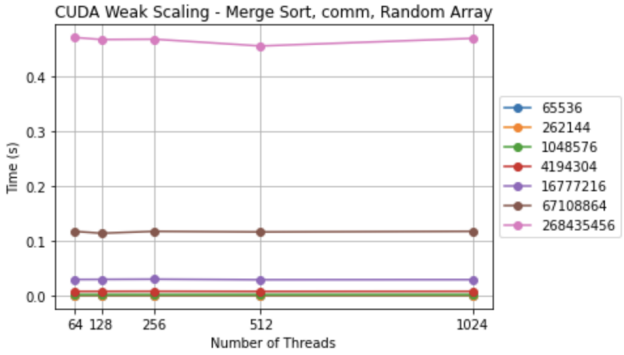

MPI Weak Scaling - Merge Sort, MPI_Gather, 1% Perturbed Array
Average Time per Rank (s)
Number of Processors
65536
262144
1048576
4194304
16777216
67108864
268435456

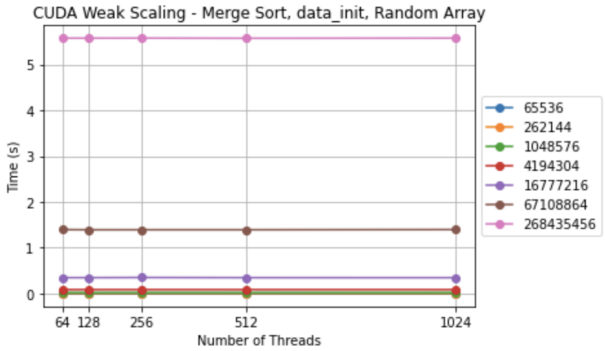
CUDA

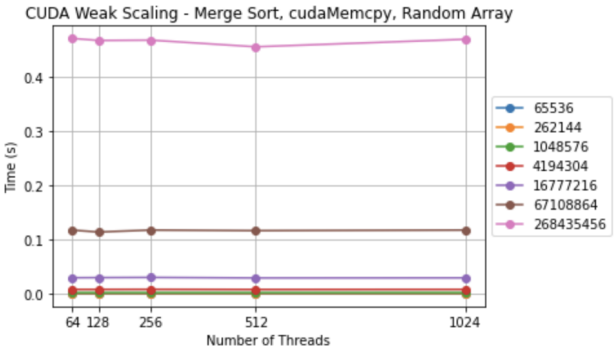
For the CUDA weak scaling of the different input types, we had a lot of the same activity as the MPI implementation. As we increased the number of elements in the input array, there was more time needed for the merge sort algorithm to complete. It also seems thats the CUDA implementation takes a bit longer to complete than the MPI implementation. We can see this most notably with the 2^28th line (the pink), where it takes over 120 seconds to complete the main_function, whereas the MPI implementation had a maximum of 40 seconds to complete. We also sea a lot steadier increase in time for the CUDA implementation. It is interesting to see that all of the different performance regions increase quite steadily. Another big difference with CUDA is how it handles communication. Rather than speaking to multiple processors like in MPI, CUDA handles only cudamemcpy, so we see rather constant communciation times between all of the different input sizes.

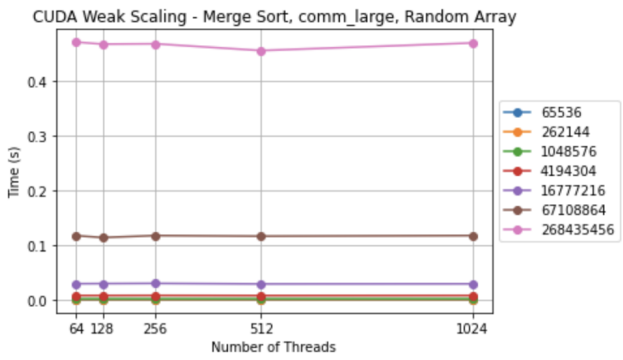
Random Array

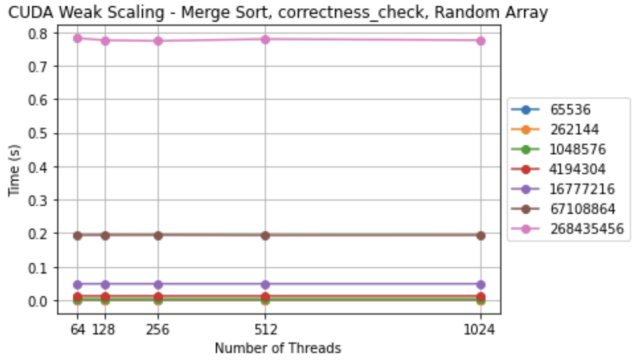


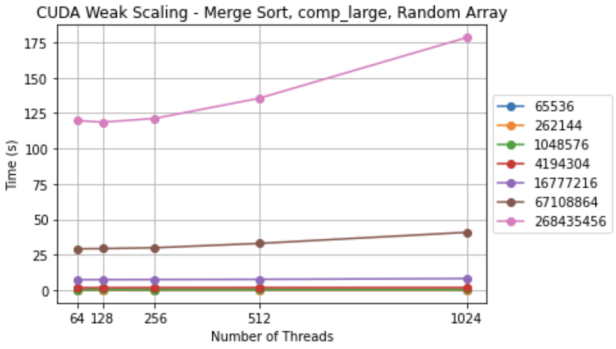


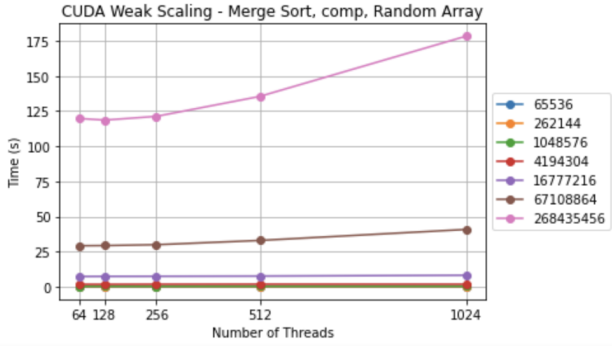




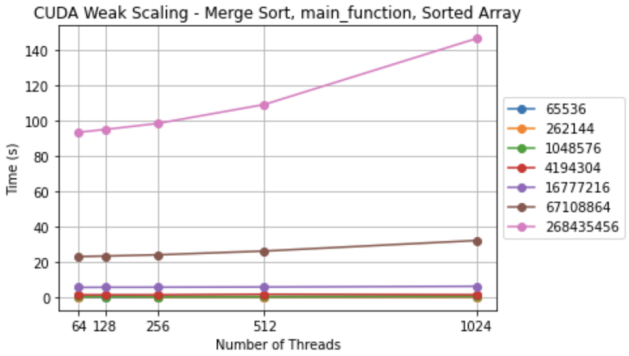


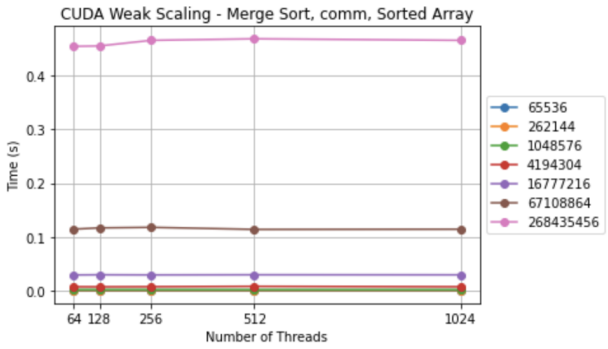


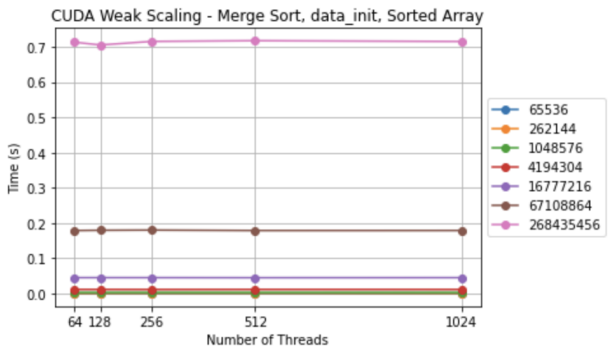


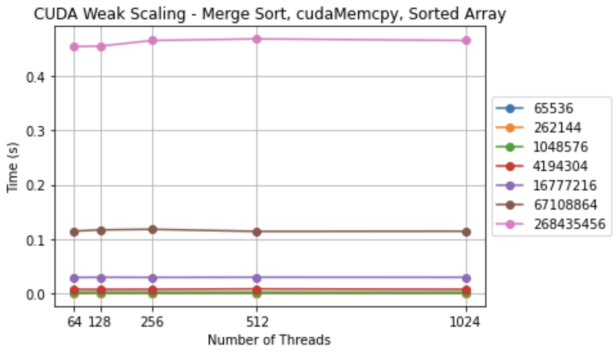


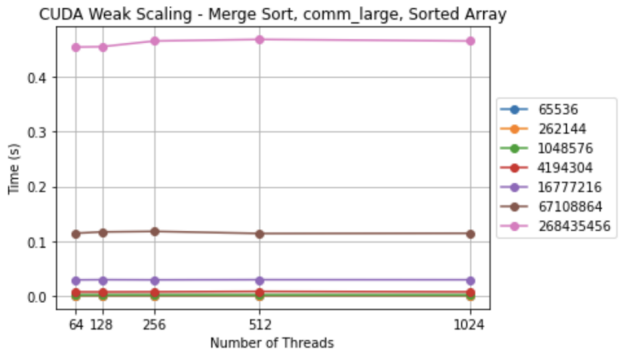
Sorted Array

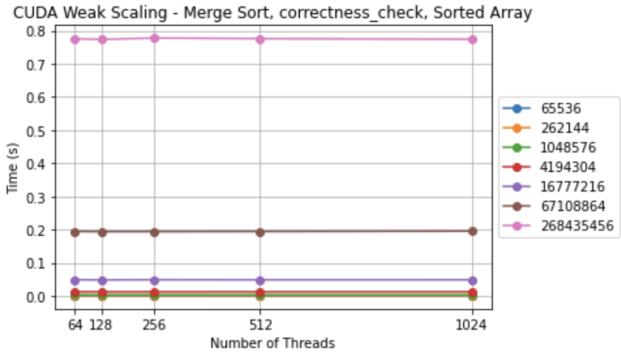


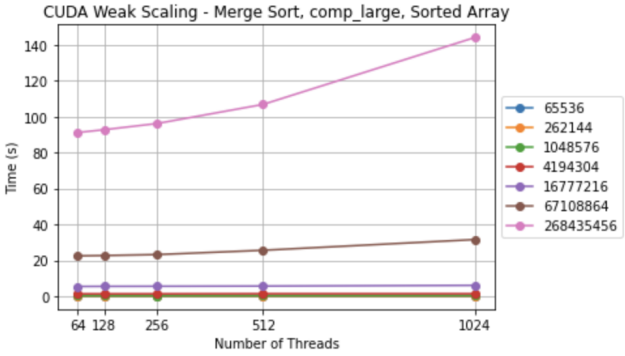


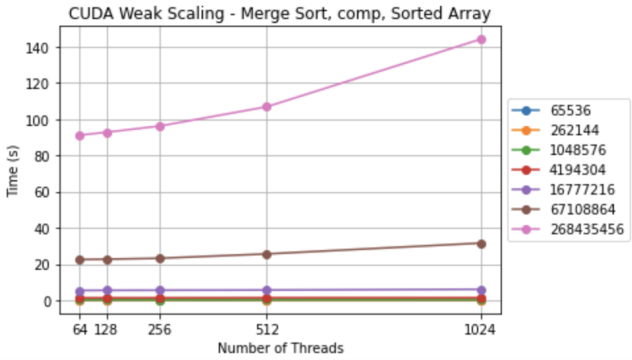




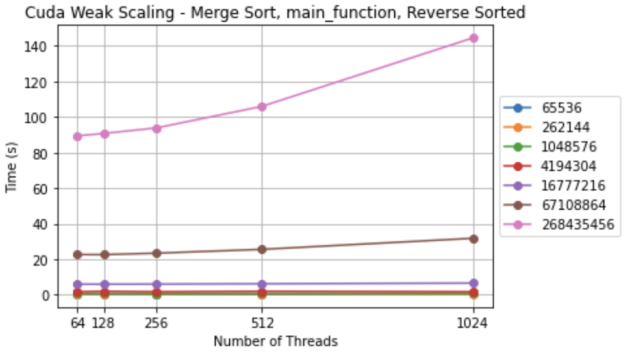


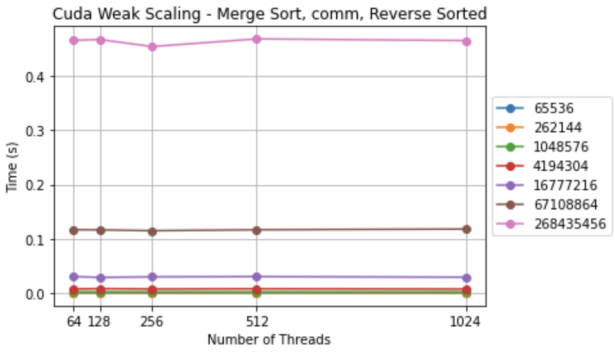


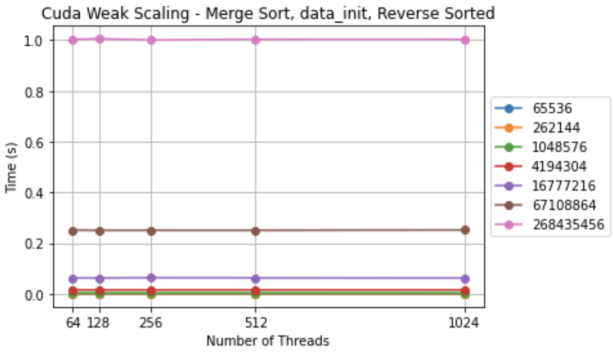


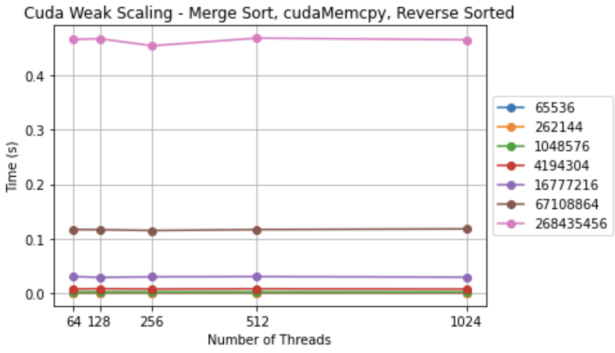


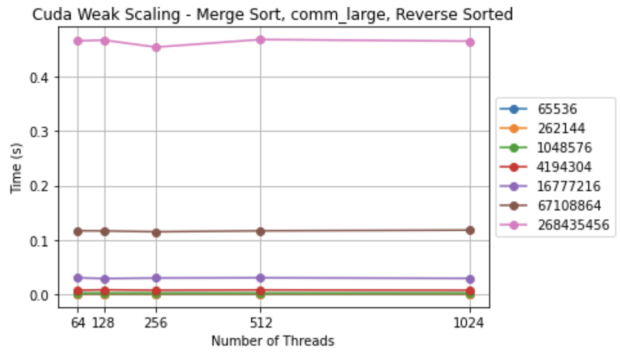
Reverse Sorted Array

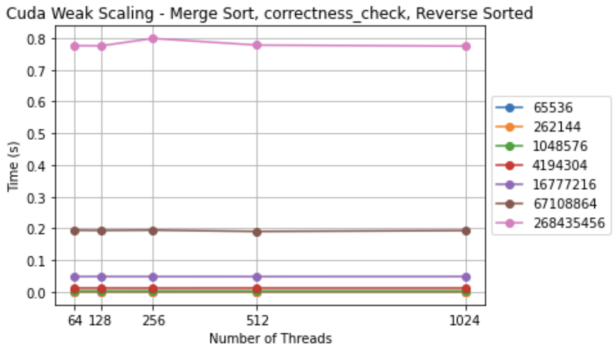


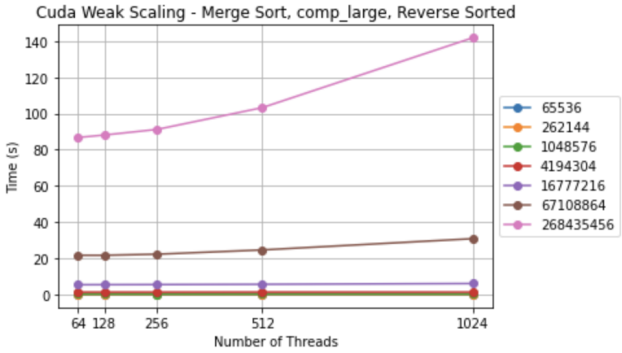


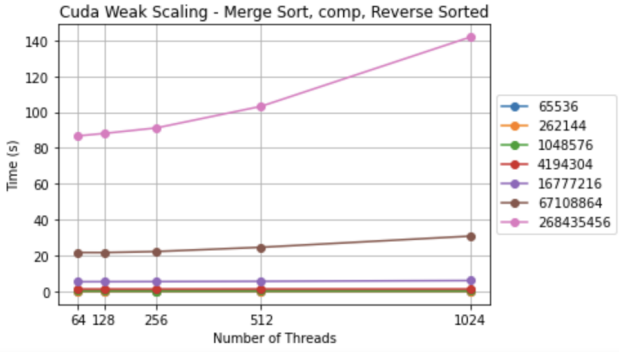




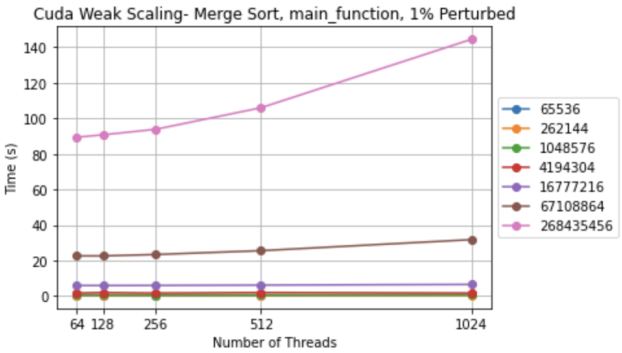


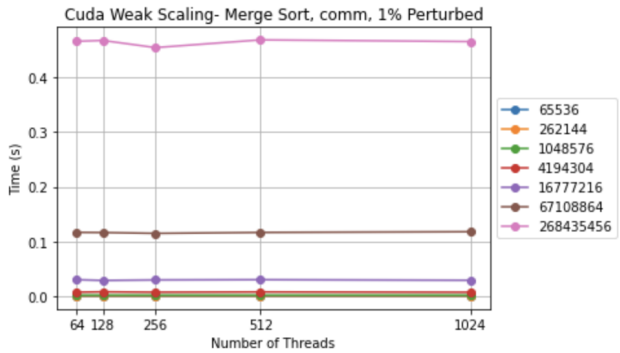


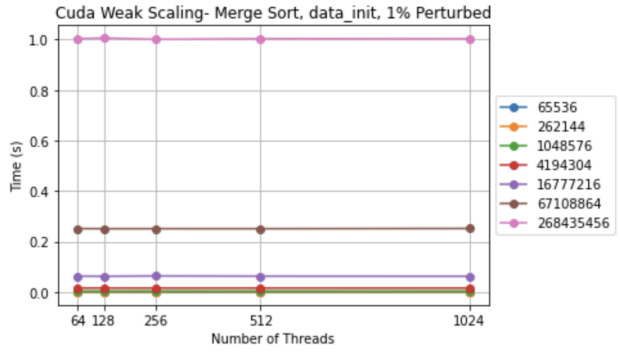


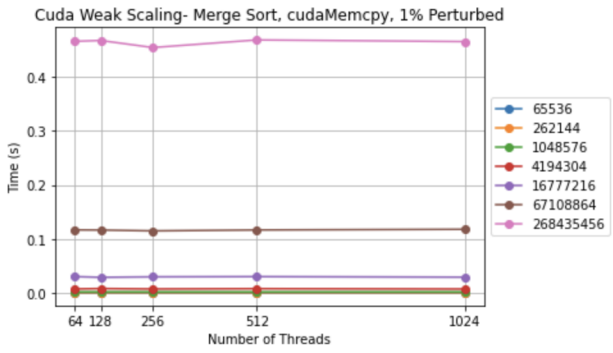


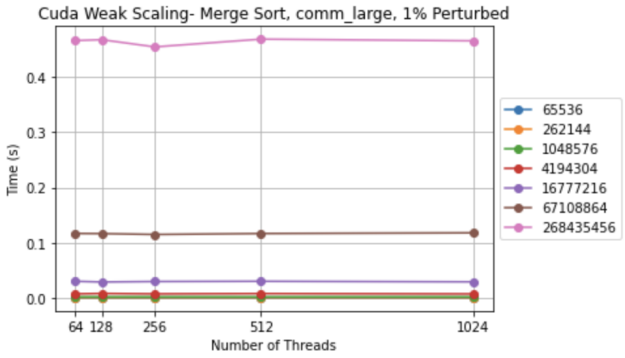
1% Perturbed Array

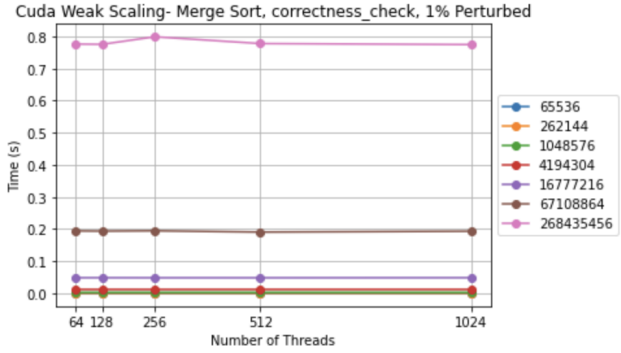


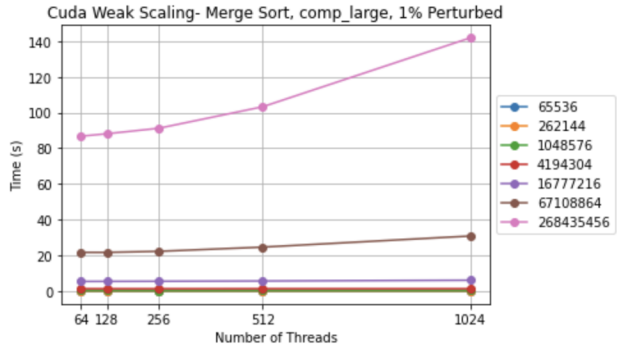


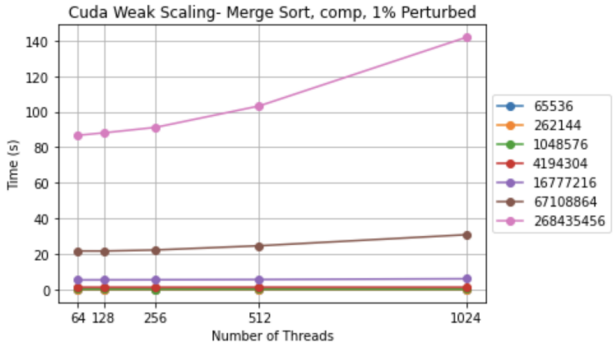










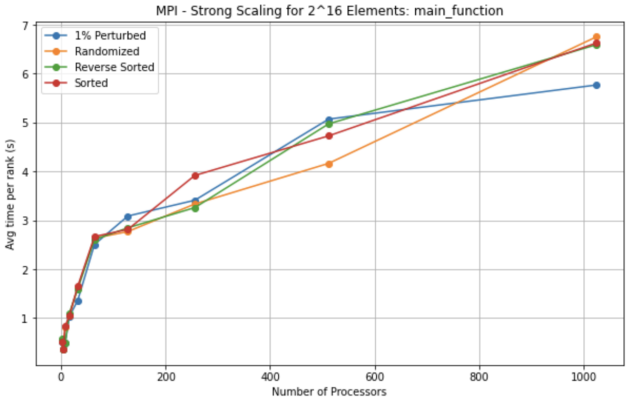


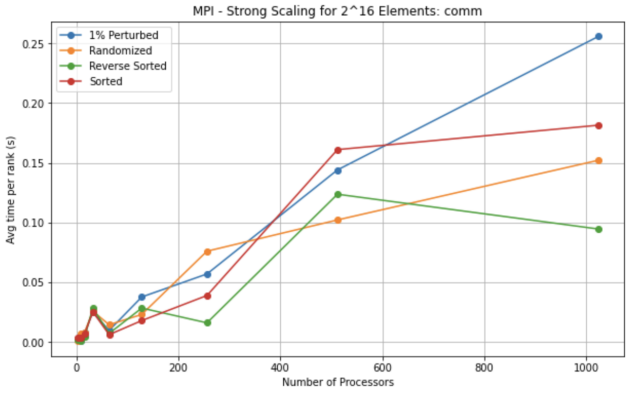
Strong Scaling

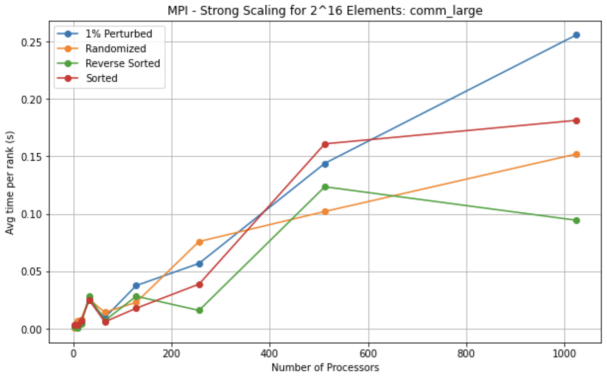
MPI

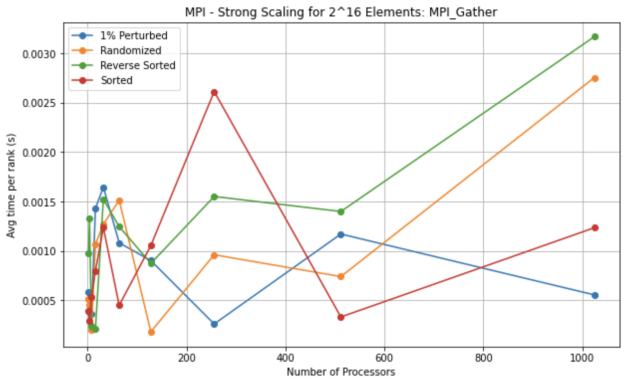
We see some similar behavior as the weak scaling here in terms of shape of the graph. The most notable one here is the comp and comp_large regions, which have the same exponentially decreasing graph, where each input type has relatively the same behavior. This is more or less expected given the nature of merge sort, where each input is eventually recursively broken down into single input subarrays. Especially considering that each graph has the same number of elements, we can understand why the computation is so much closer between the lines for this strong scaling. It is interesting because this strong scaling shows that MPI_Scatter is much slower than MPI_Gather, which is the opposite than from the weak scaling. We also see that as we increase the number of elements, our behavior becomes more erratic, which is most easily seen in the main_function graph of each input size. We're also seeing that the time peaks around 128 processors before decreasing and then further increasing again. For communication, we are seeing some steady increase in time as well, but overall pretty fast across the ranks.

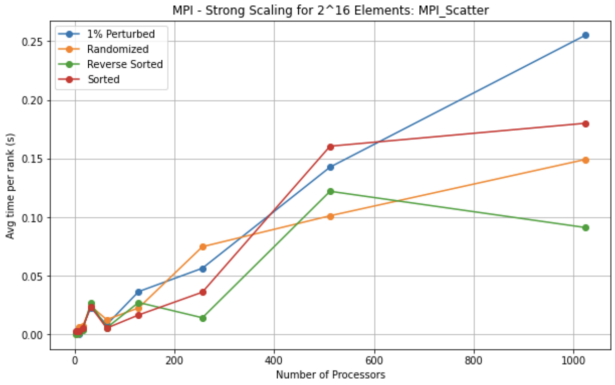
2^16 Elements

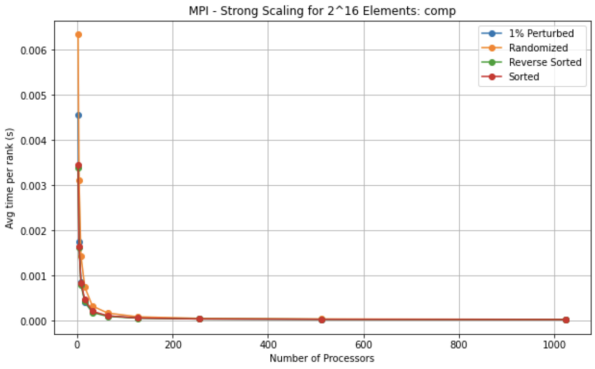

MPI - Strong Scaling for 2^16 Elements: main_function
1% Perturbed
Randomized
Reverse Sorted
Sorted
Avg time per rank (s)
Number of Processors

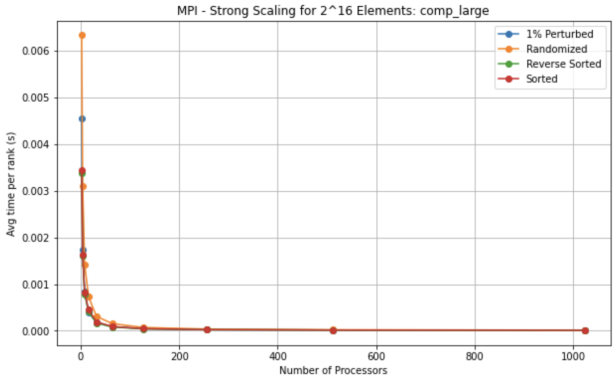

MPI - Strong Scaling for 2^16 Elements: comm
1% Perturbed
Randomized
Reverse Sorted
Sorted
Avg time per rank (s)
Number of Processors

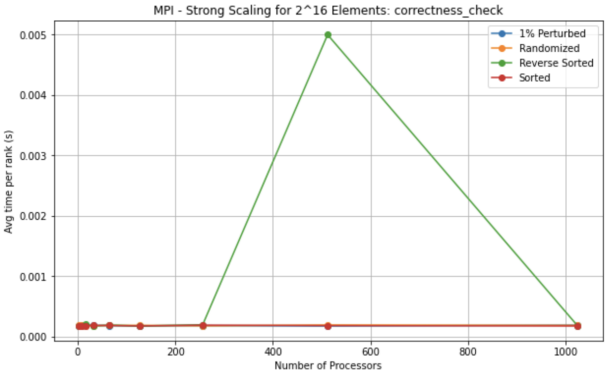

MPI - Strong Scaling for 2^16 Elements: comm_large
1% Perturbed
Randomized
Reverse Sorted
Sorted
Avg time per rank (s)
Number of Processors

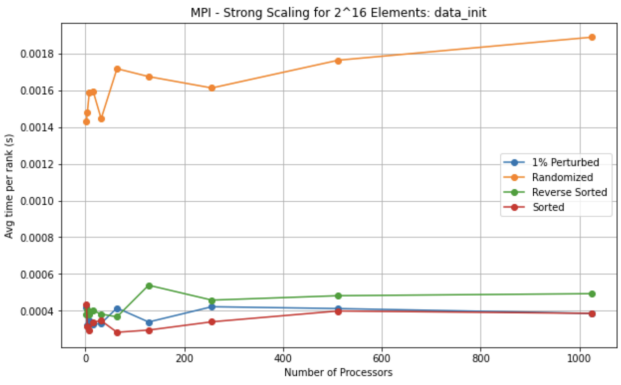

MPI - Strong Scaling for 2^16 Elements: MPI_Gather
1% Perturbed
Randomized
Reverse Sorted
Sorted
Avg time per rank (s)
Number of Processors


MPI - Strong Scaling for 2^16 Elements: MPI_Scatter
1% Perturbed
Randomized
Reverse Sorted
Sorted
Avg time per rank (s)
Number of Processors

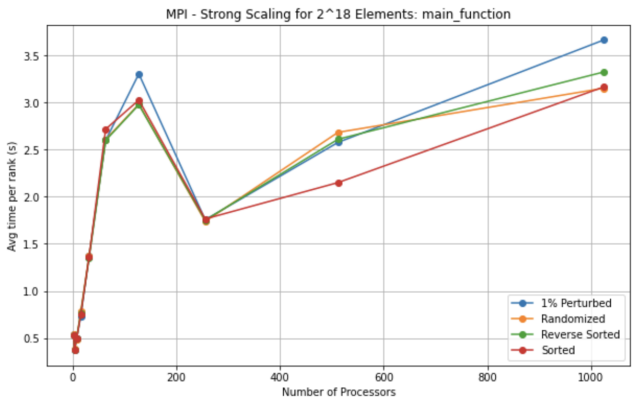

MPI - Strong Scaling for 2^16 Elements: comp
1% Perturbed
Randomized
Reverse Sorted
Sorted
Avg time per rank (s)
Number of Processors

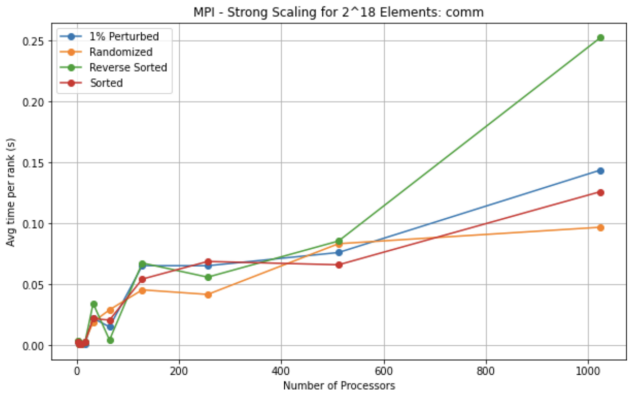

MPI - Strong Scaling for 2^16 Elements: comp_large
1% Perturbed
Randomized
Reverse Sorted
Sorted
Avg time per rank (s)
Number of Processors

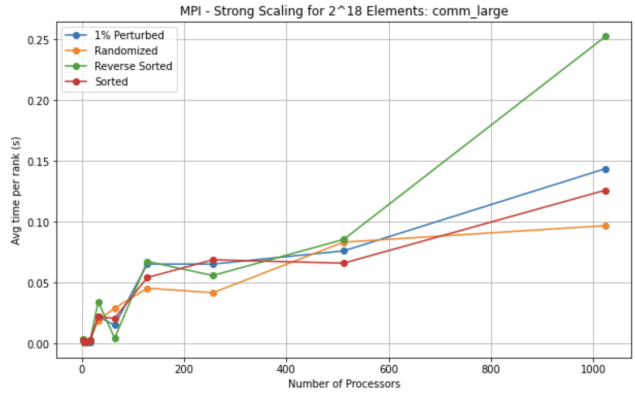

MPI - Strong Scaling for 2^16 Elements: correctness_check
1% Perturbed
Randomized
Reverse Sorted
Sorted
Avg time per rank (s)
Number of Processors

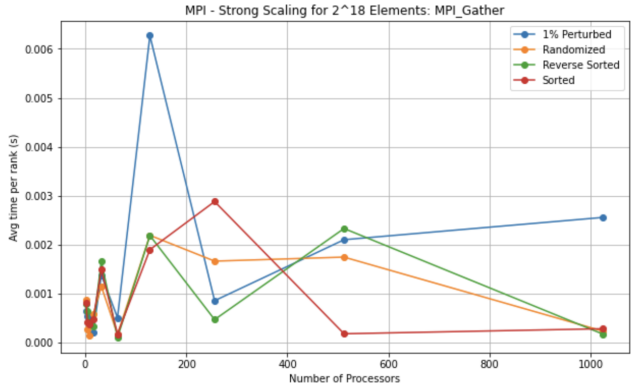

MPI - Strong Scaling for 2^16 Elements: data_init
1% Perturbed
Randomized
Reverse Sorted
Sorted
Avg time per rank (s)
Number of Processors

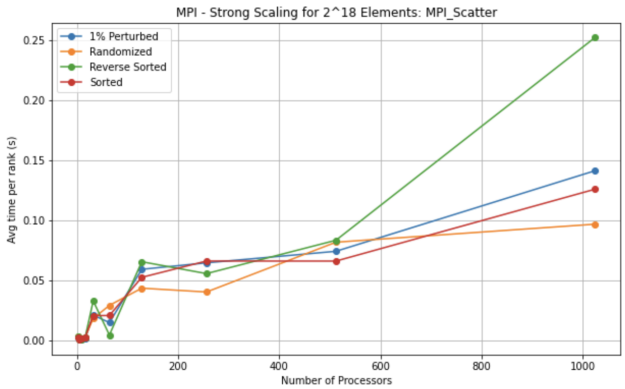
2 ^18 Elements

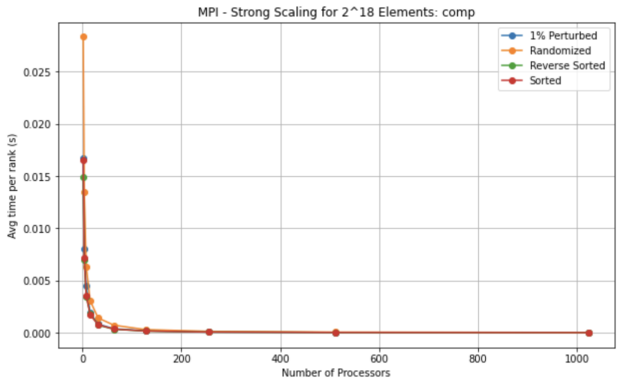


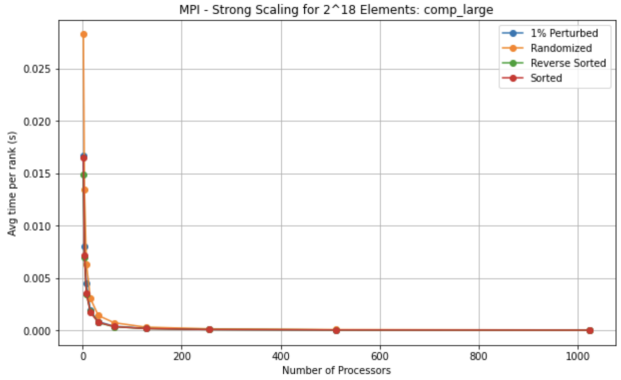


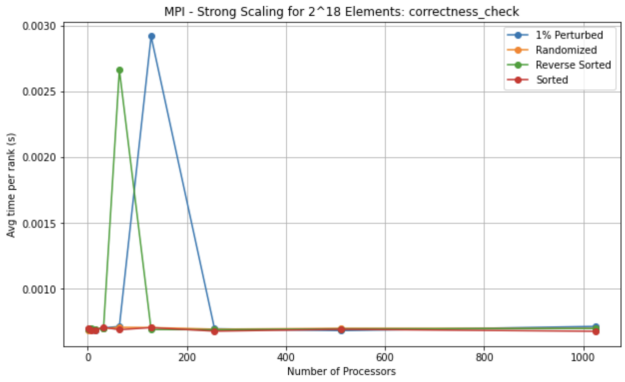


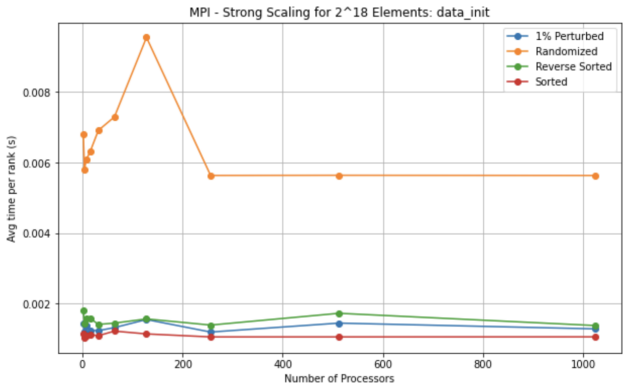




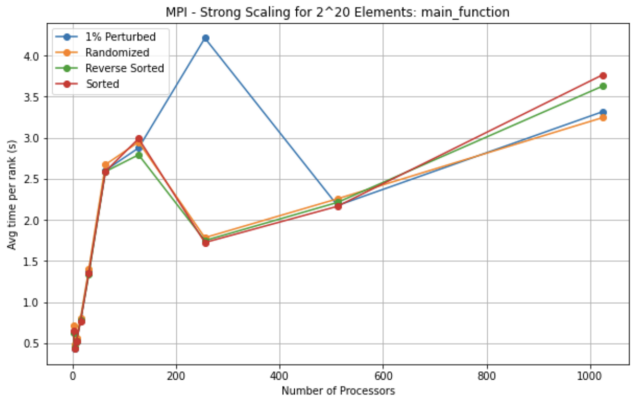


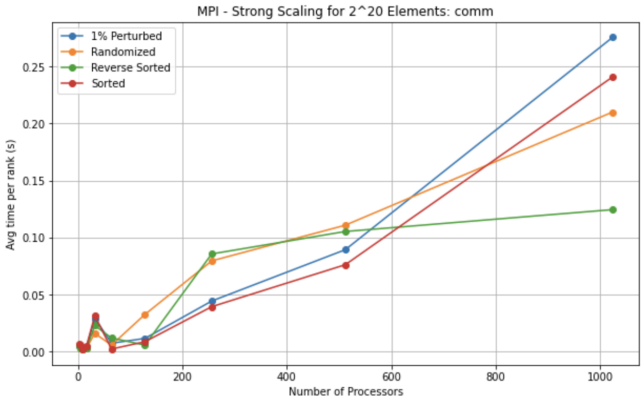


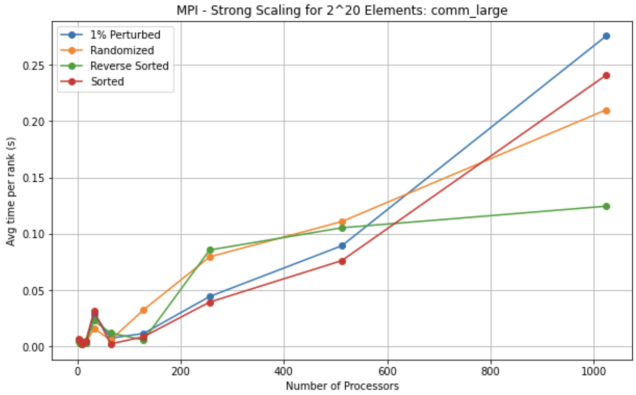


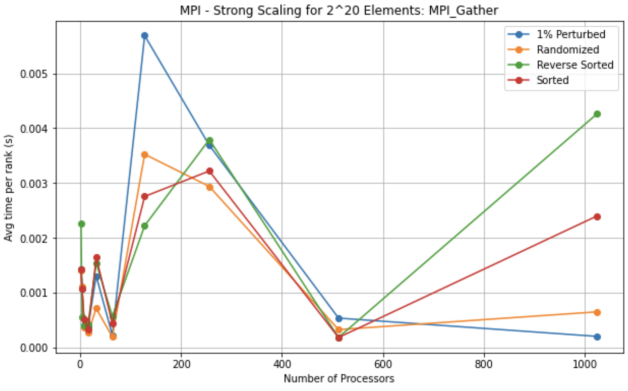


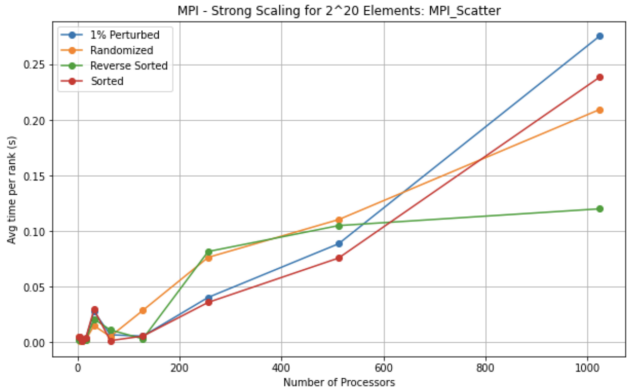
2^20 Elements

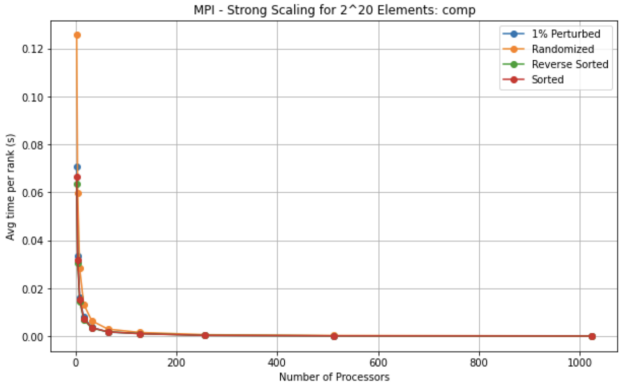


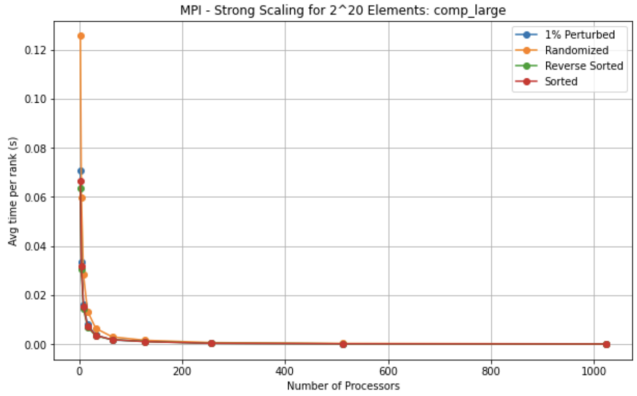


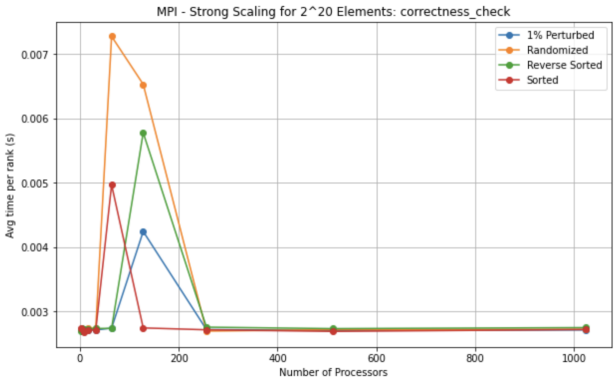


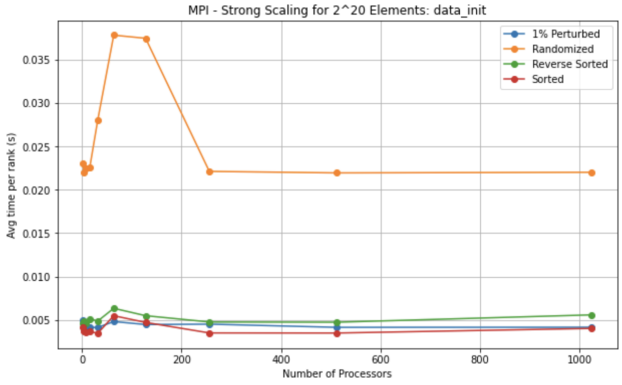




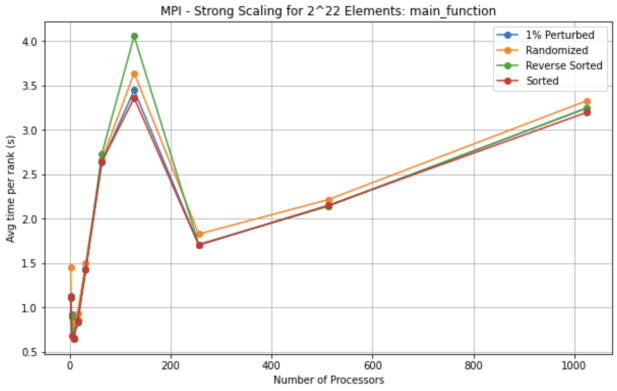


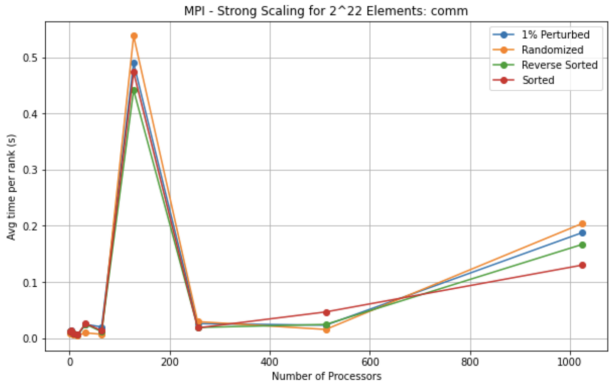


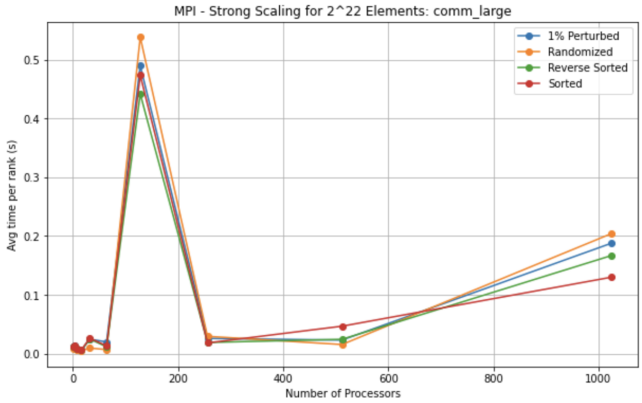


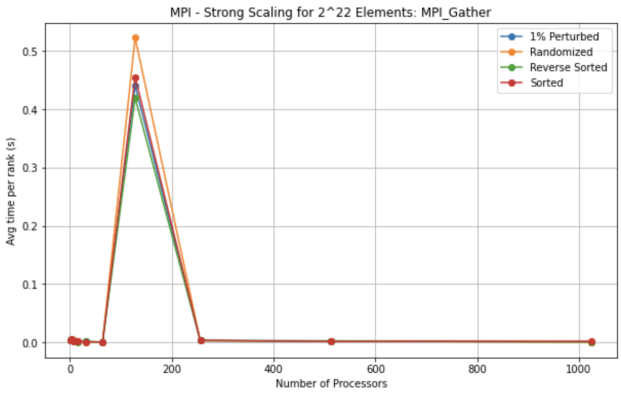


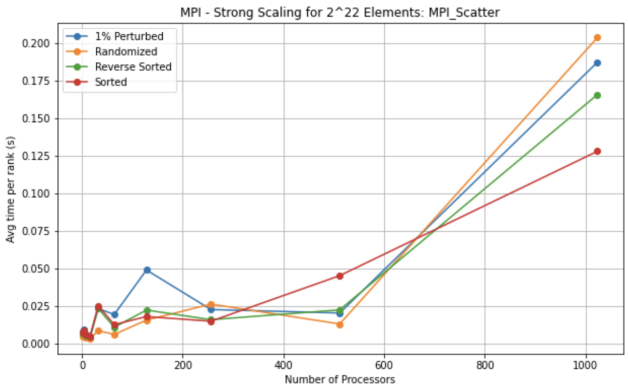
2^22 Elements

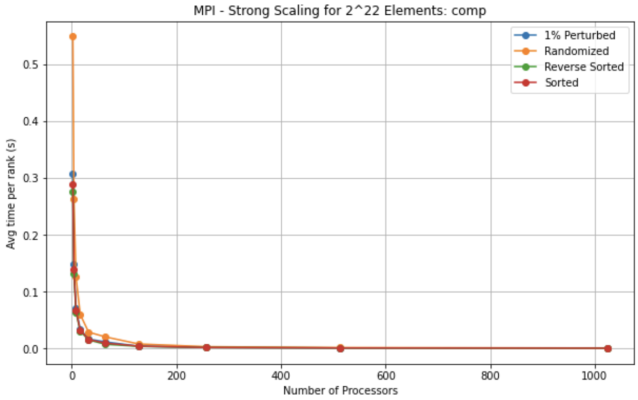

MPI - Strong Scaling for 2^22 Elements: main_function
1% Perturbed
Randomized
Reverse Sorted
Sorted
Avg time per rank (s)
Number of Processors

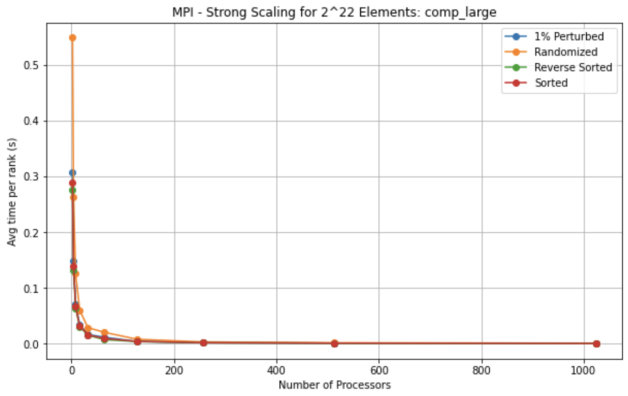

MPI - Strong Scaling for 2^22 Elements: comm
1% Perturbed
Randomized
Reverse Sorted
Sorted
Avg time per rank (s)
Number of Processors

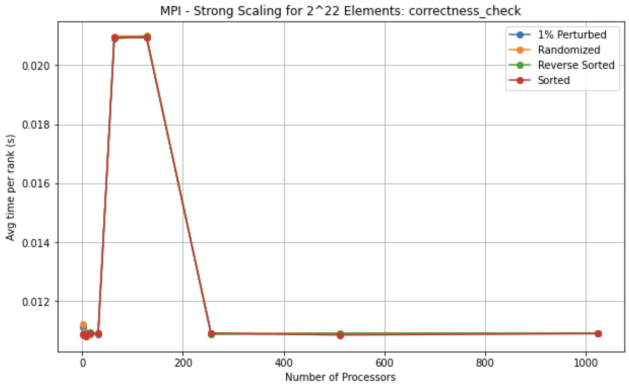

MPI - Strong Scaling for 2^22 Elements: comm_large
1% Perturbed
Randomized
Reverse Sorted
Sorted
Avg time per rank (s)
Number of Processors

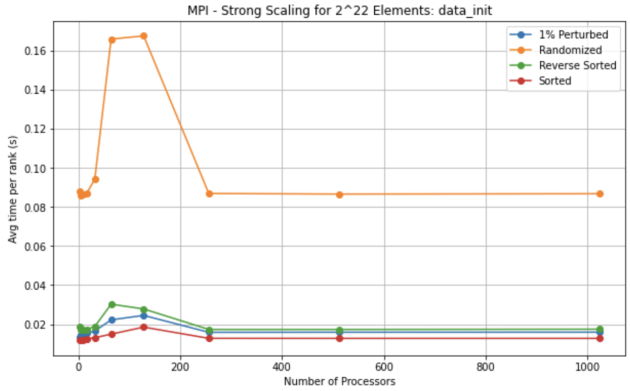

MPI - Strong Scaling for 2^22 Elements: MPI_Gather
1% Perturbed
Randomized
Reverse Sorted
Sorted
Avg time per rank (s)
Number of Processors


MPI - Strong Scaling for 2^22 Elements: MPI_Scatter
1% Perturbed
Randomized
Reverse Sorted
Sorted
Avg time per rank (s)
Number of Processors

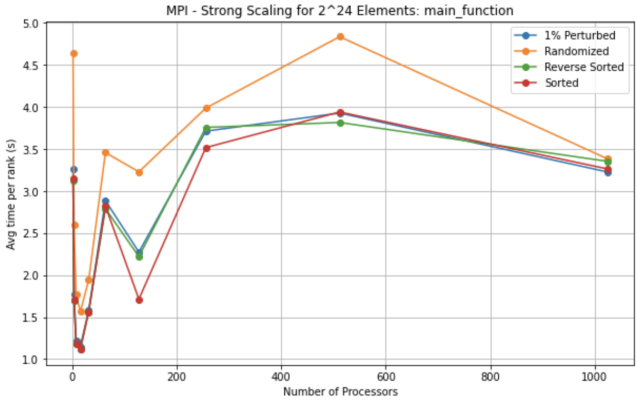

MPI - Strong Scaling for 2^22 Elements: comp
1% Perturbed
Randomized
Reverse Sorted
Sorted
Avg time per rank (s)
Number of Processors

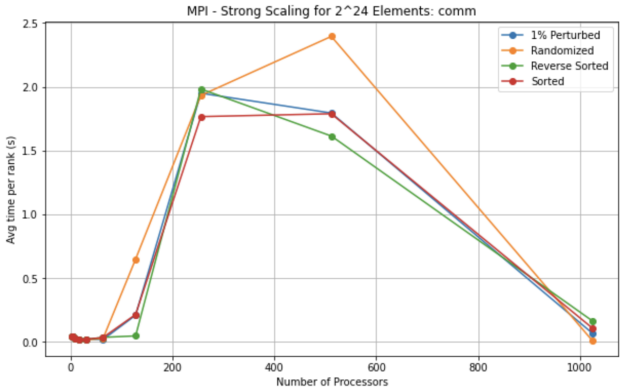


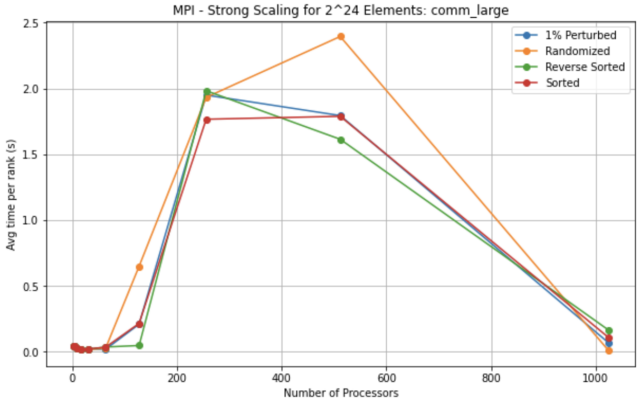


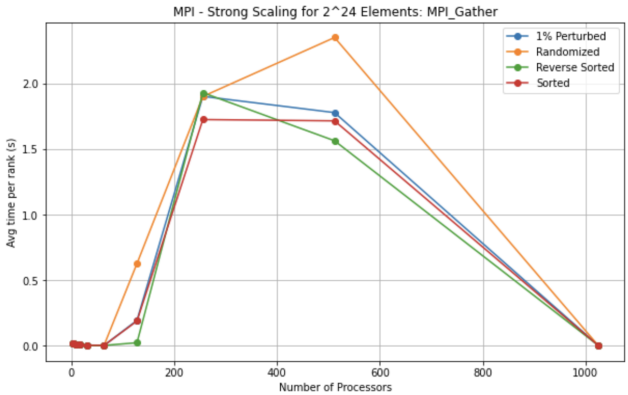


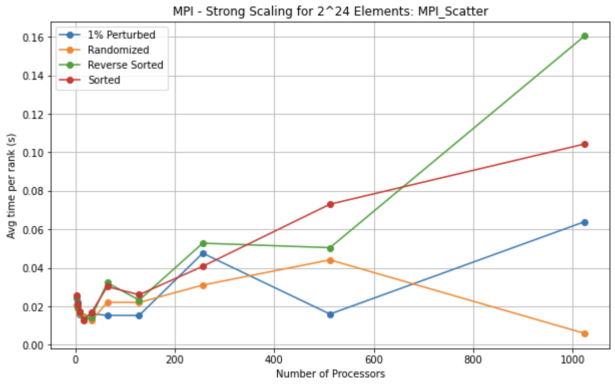
2^24 Elements

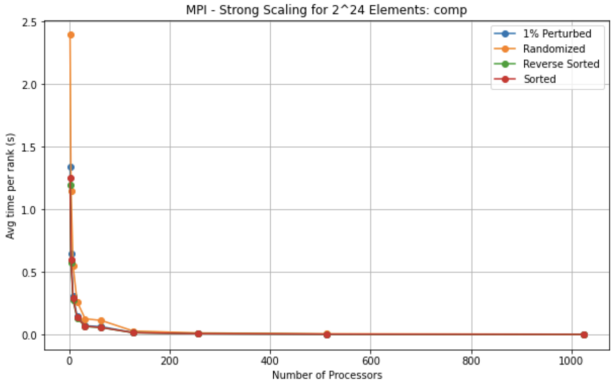


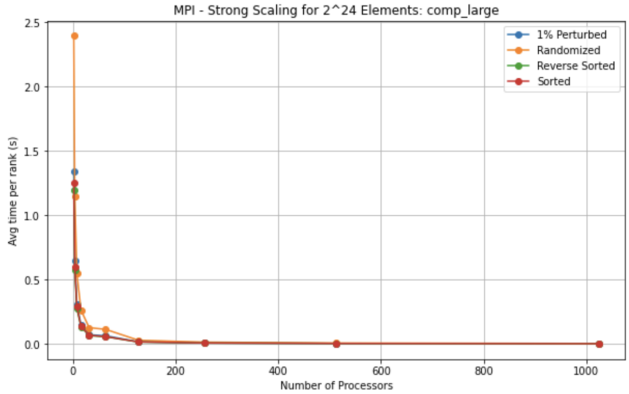


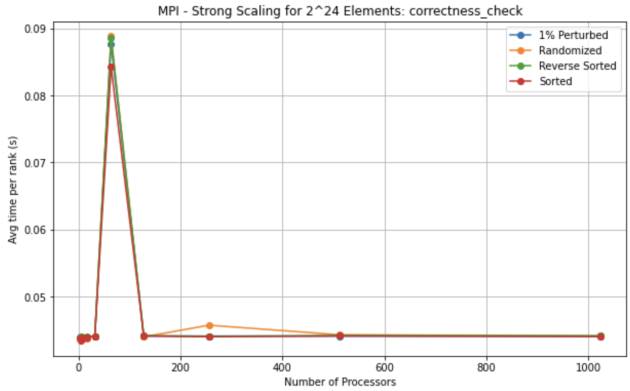


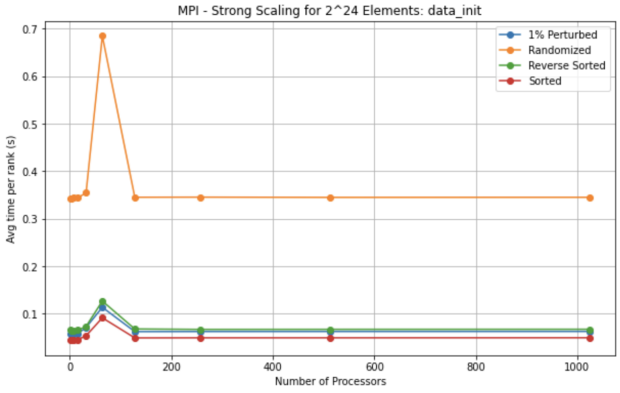




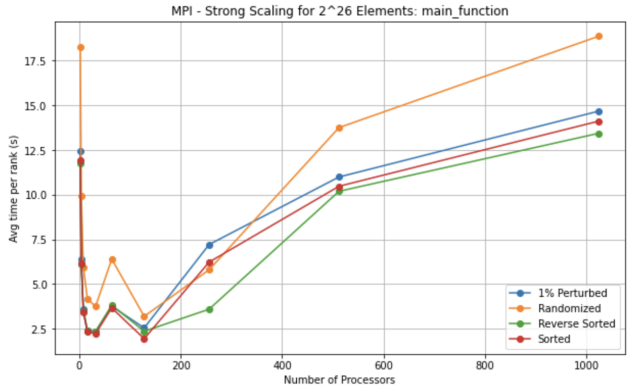


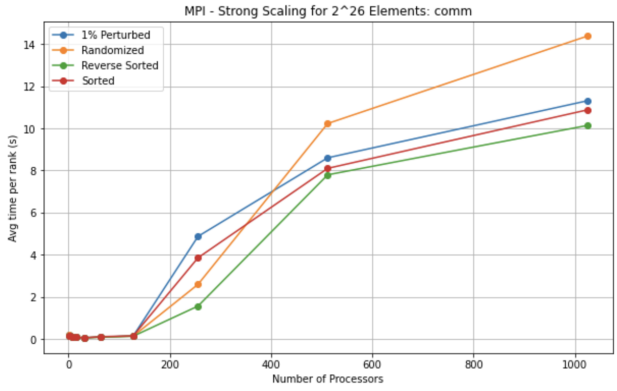


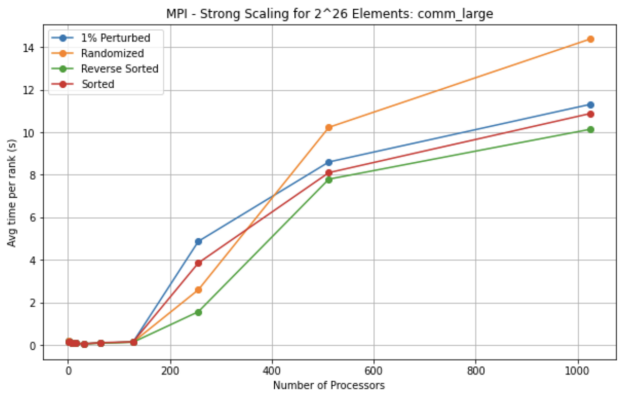


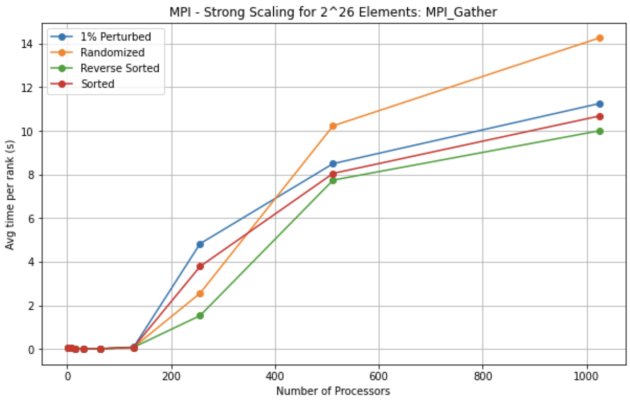


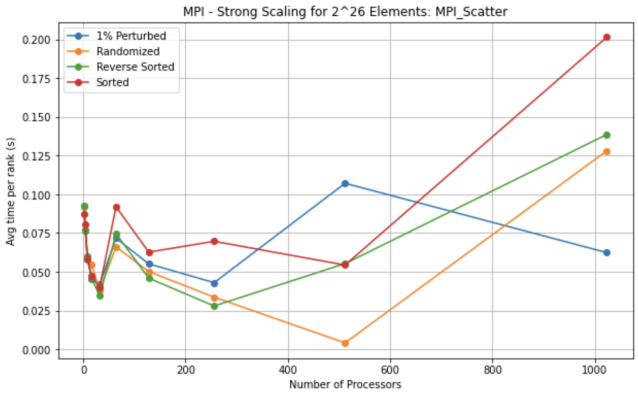
2^26 Elements

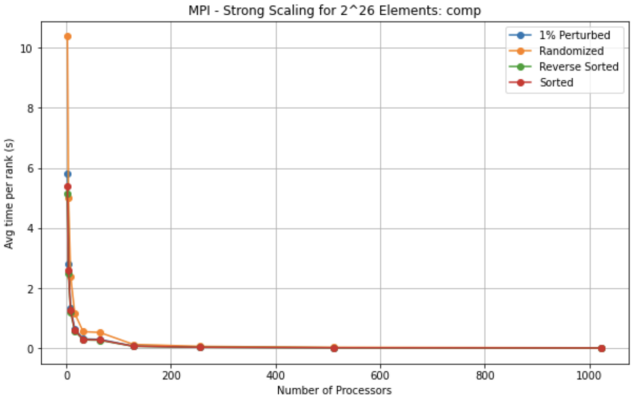

MPI - Strong Scaling for 2^26 Elements: main_function
Avg time per rank (s)
Number of Processors
1% Perturbed
Randomized
Reverse Sorted
Sorted

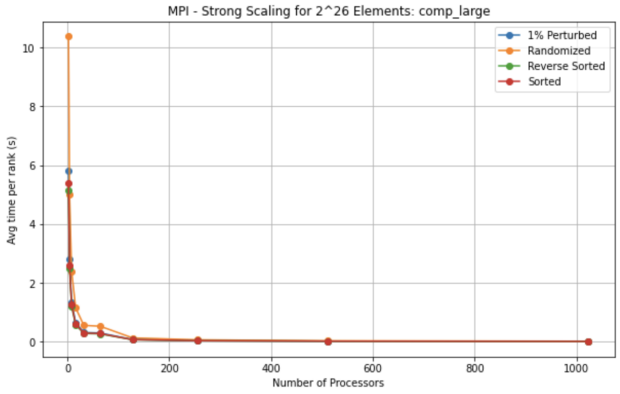

MPI - Strong Scaling for 2^26 Elements: comm
Avg time per rank (s)
Number of Processors
1% Perturbed
Randomized
Reverse Sorted
Sorted

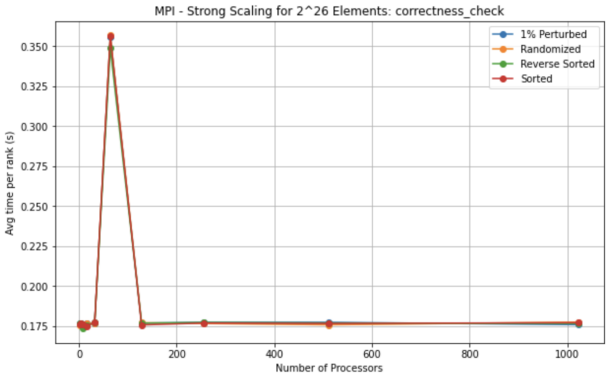

MPI - Strong Scaling for 2^26 Elements: comm_large
Avg time per rank (s)
Number of Processors
1% Perturbed
Randomized
Reverse Sorted
Sorted

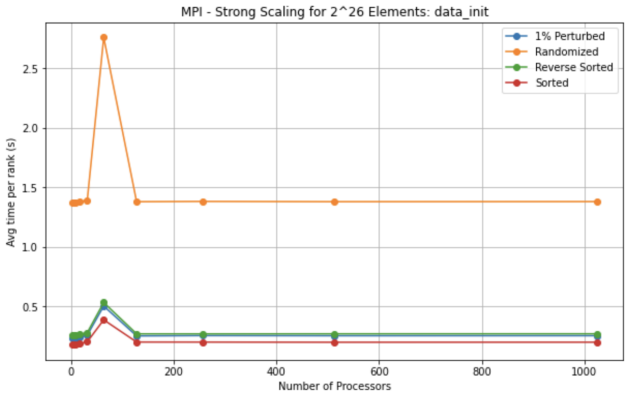

MPI - Strong Scaling for 2^26 Elements: MPI_Gather
Avg time per rank (s)
Number of Processors
1% Perturbed
Randomized
Reverse Sorted
Sorted


MPI - Strong Scaling for 2^26 Elements: MPI_Scatter
Avg time per rank (s)
Number of Processors
1% Perturbed
Randomized
Reverse Sorted
Sorted

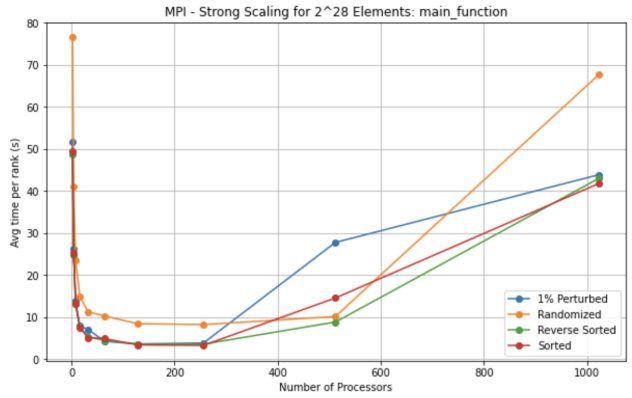

MPI - Strong Scaling for 2^26 Elements: comp
Avg time per rank (s)
Number of Processors
1% Perturbed
Randomized
Reverse Sorted
Sorted

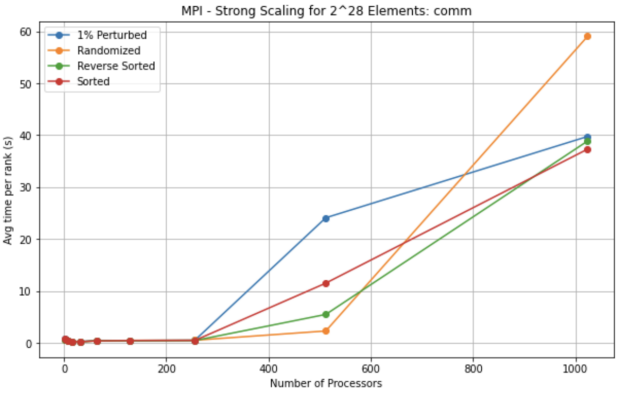


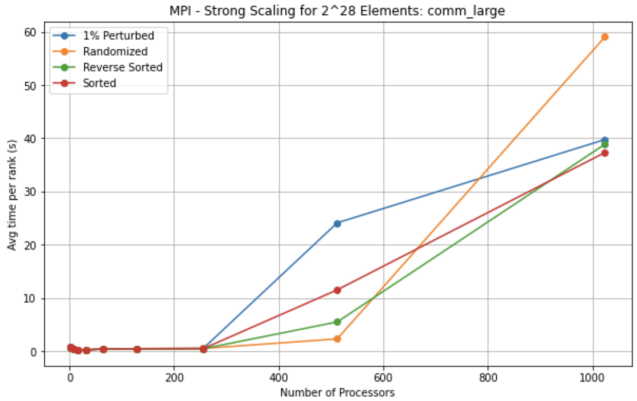


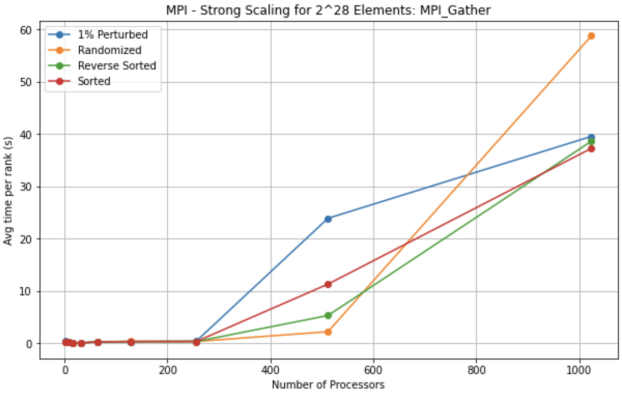


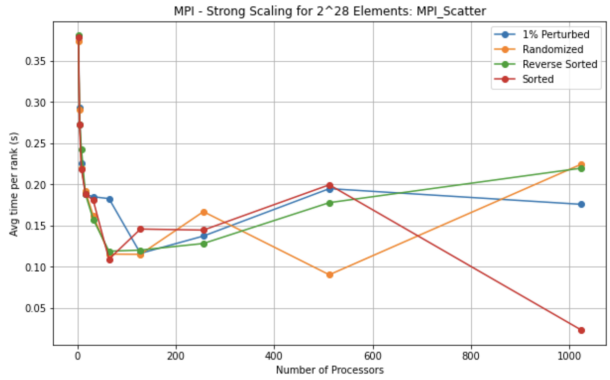
2^28 Elements

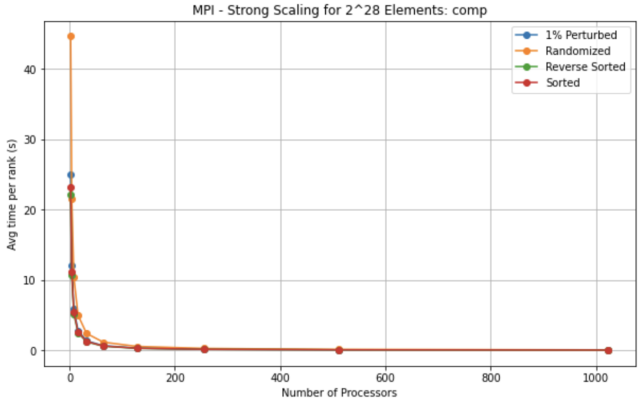


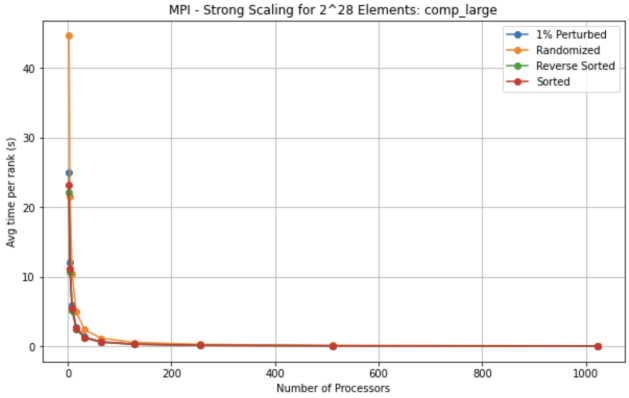


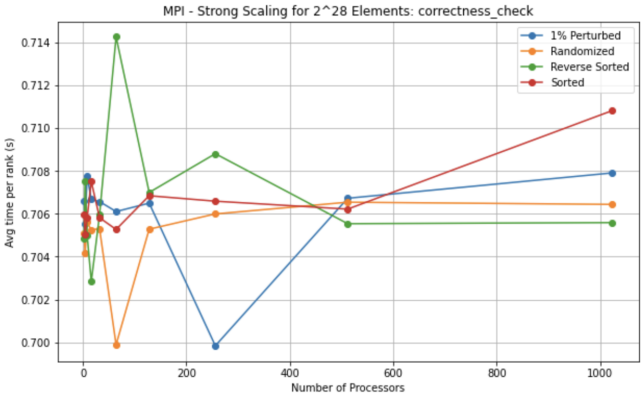


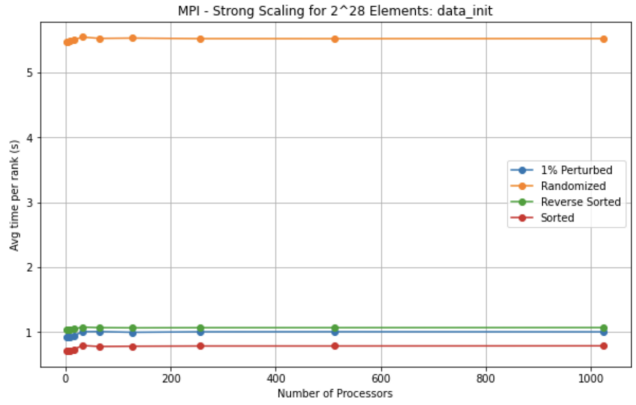










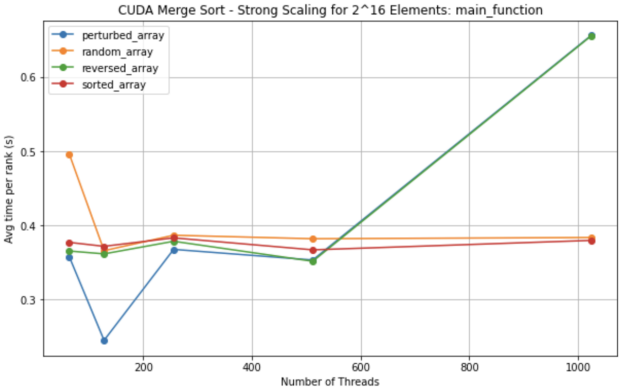


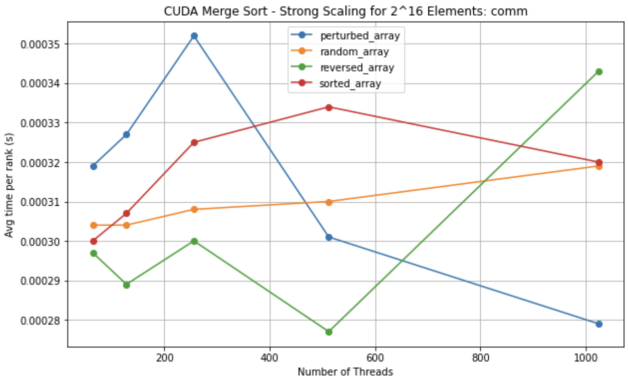
CUDA

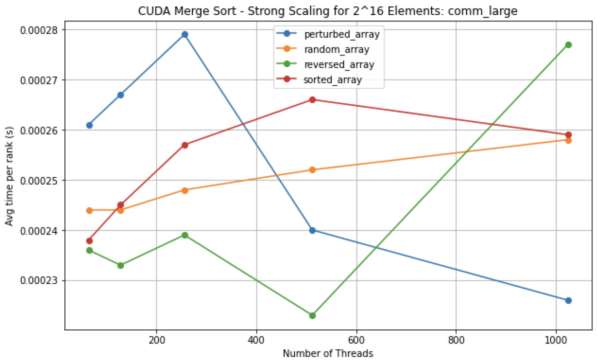
Looking at the graphs for CUDA strong scaling before, the data seems to be much more erratic. Rather than steady increases, we have many fluctuations as we increase the number of threads at a certain size, regardless of the input type. There doesn't seem to be an input type that outperforms the others. This is rather expected because merge sort will continue to break down the elements, regardless of the input type. It does look like the random array input does take a little more time than the other inputs, as it generally sits above the rest of the input types. This is followed by the 1% perturbed array. However, it is important to note that the scale of these graphs tend to be very close together, which can signify that there is no real tradeoff here based on input type. We definietly see that the comm region of the code, which again is around the CUDAMemcpys, fluctuates quite a bit, regardless of input type. There is no definite winner here. We get the main differences in computation, but again due to scale, isn't very significant. It is interesting to see though what could potentially cause this. Random taking the longest makes sense because there are a lot of comparisons, but it may be concerning that the reverse sorted array is one of the quicker ones, since this should theoretically have the maximum number of comparisons to make. One thing to note is that these two slowest, the random and perturbed array, have the slowest data initialization timees as well, so this could be a cause. We also see that

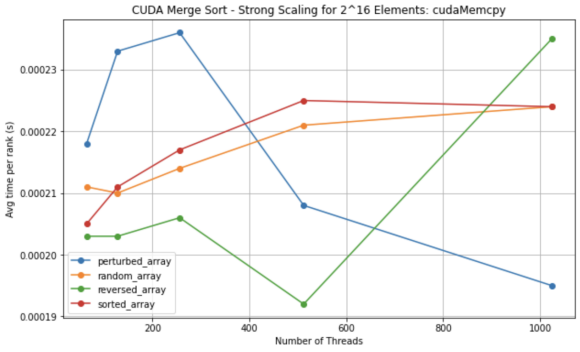
the merge step was done on the CPU, which introduces a bit of a bottleneck as we increase the problem size and number of threads.

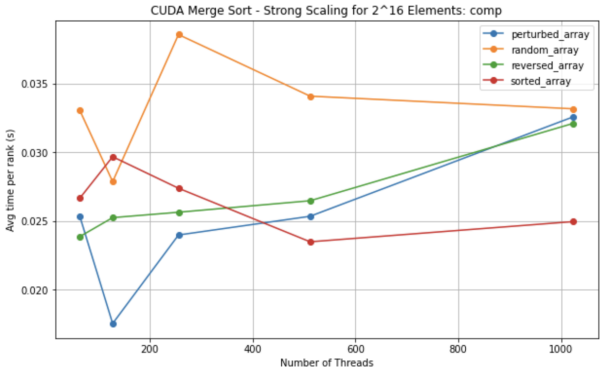
2^16 Elements

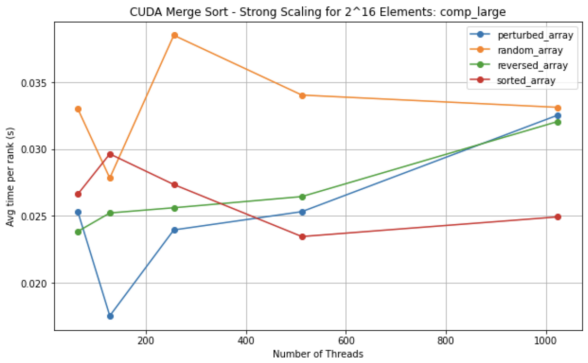


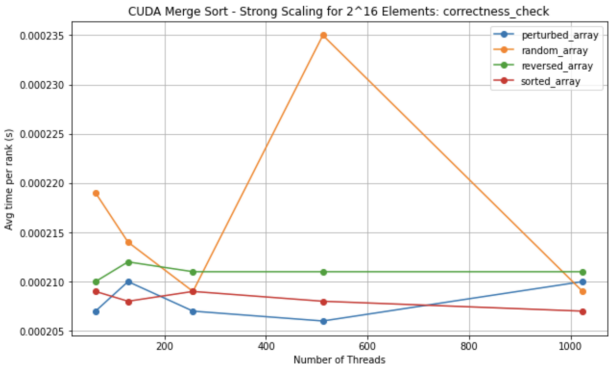


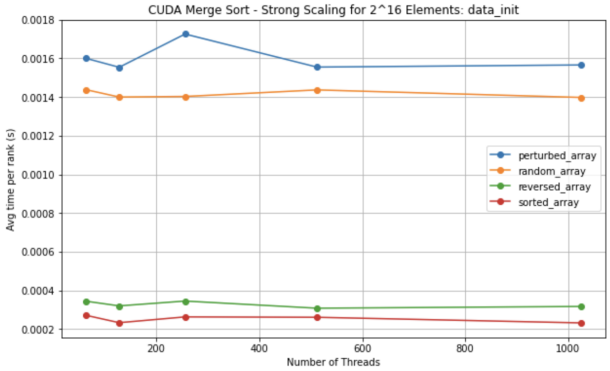




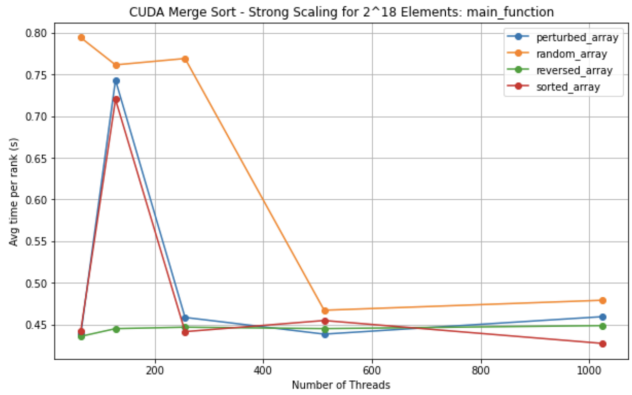


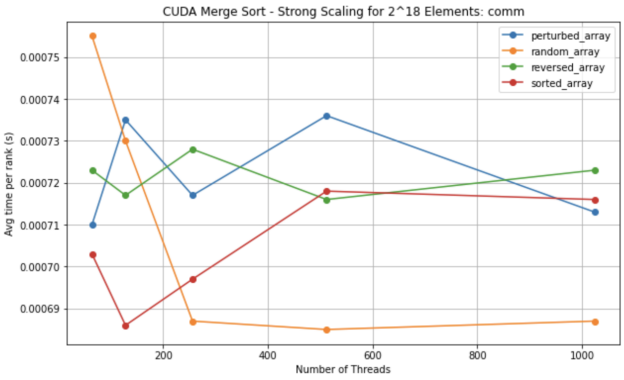


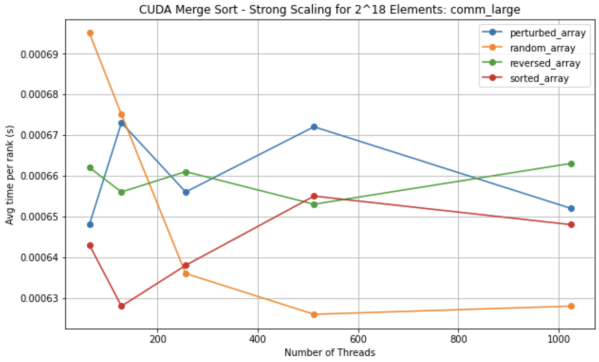


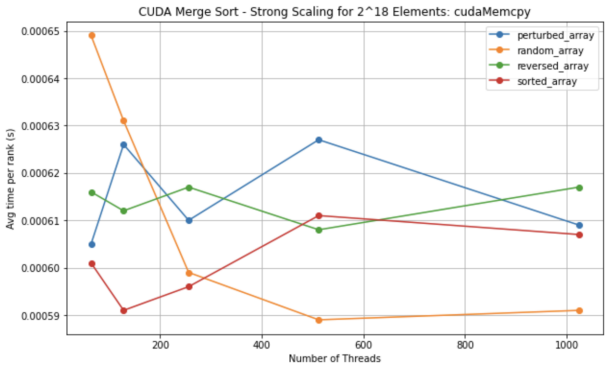


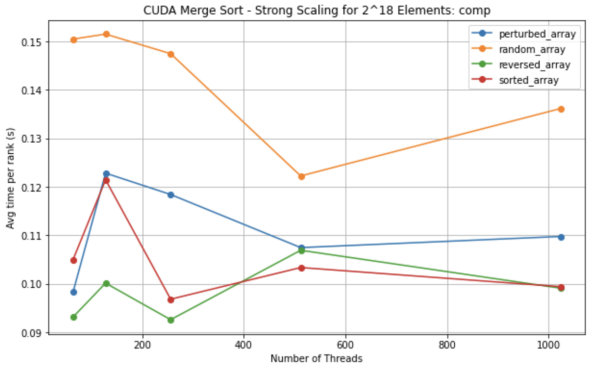
2 ^18 Elements

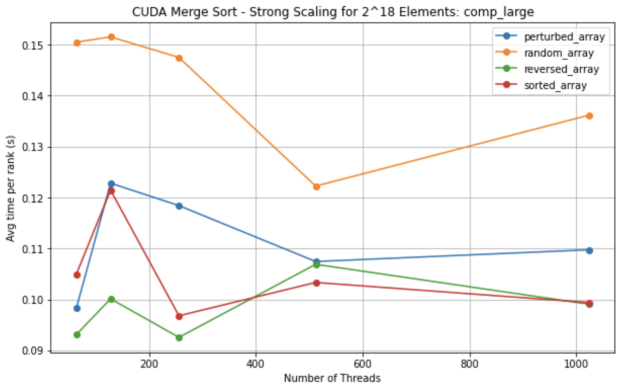


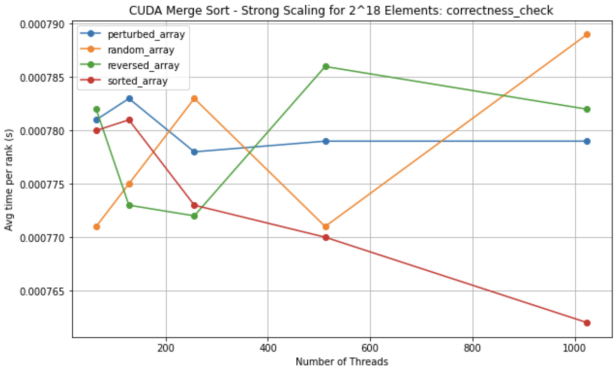


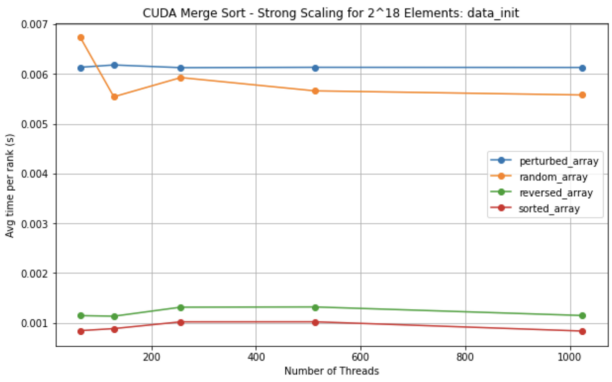




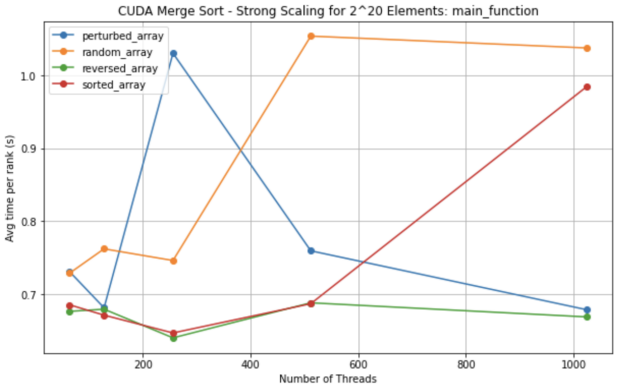


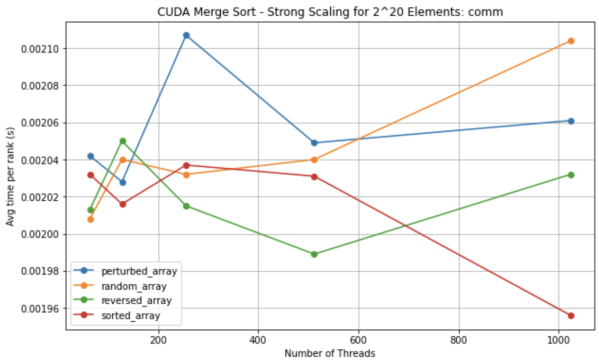


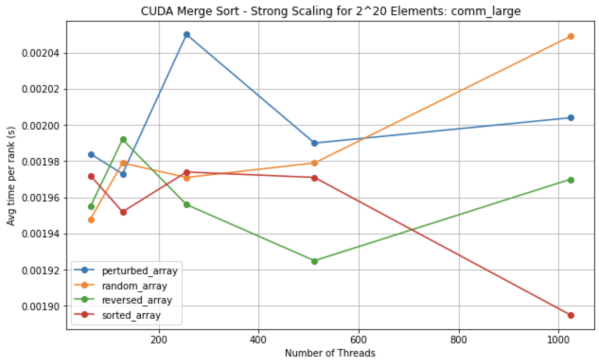


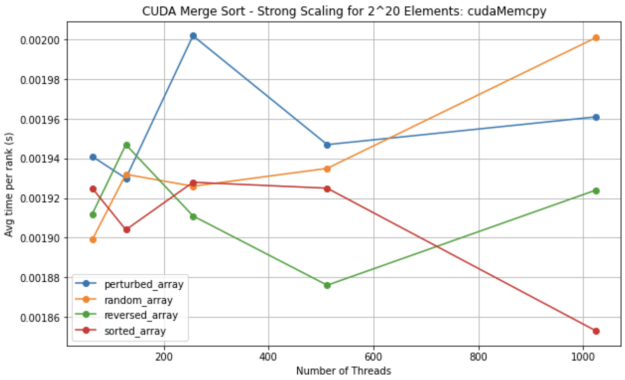


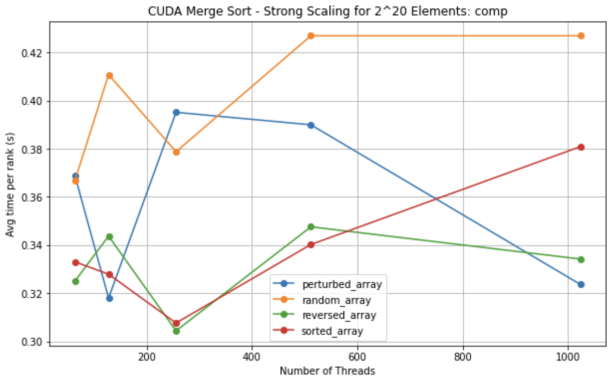
2^20 Elements

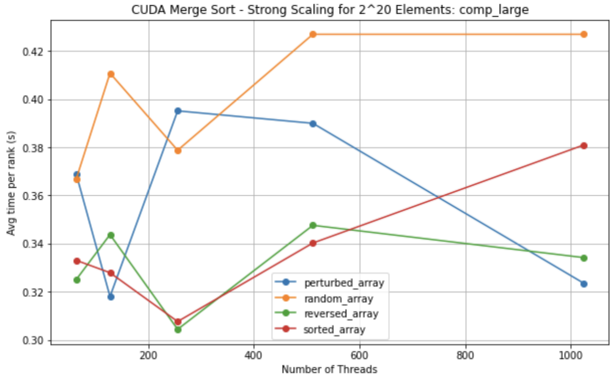


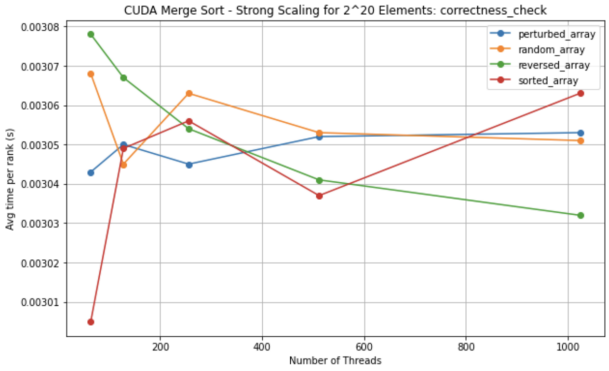


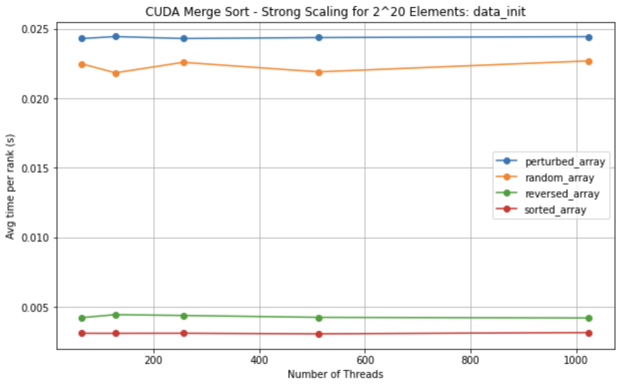




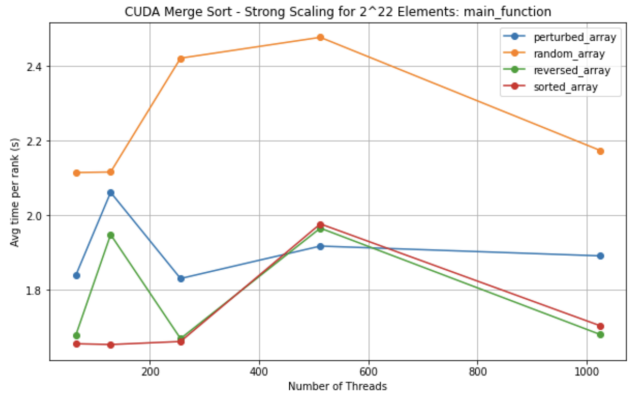


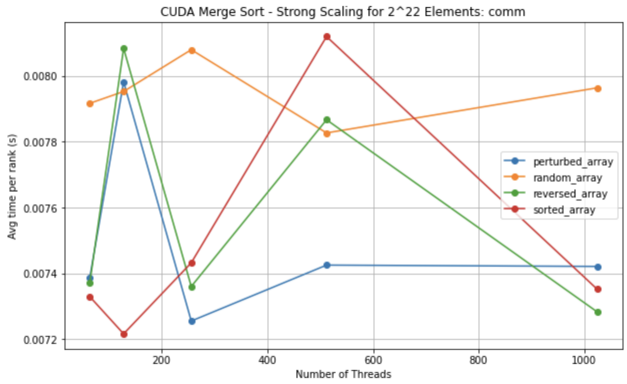


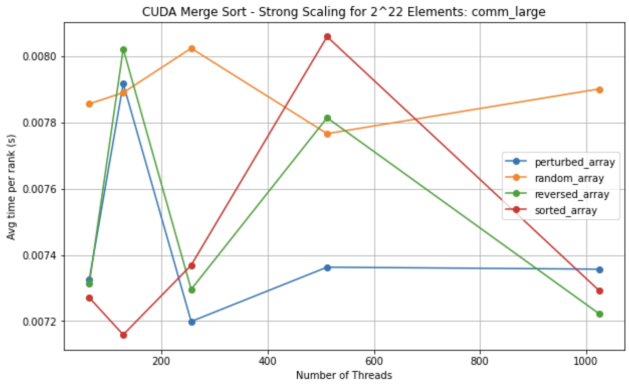


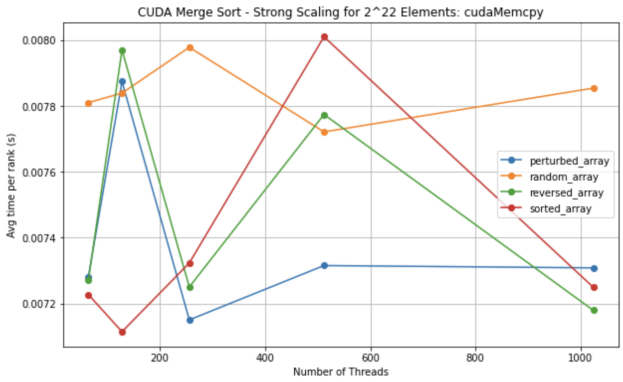


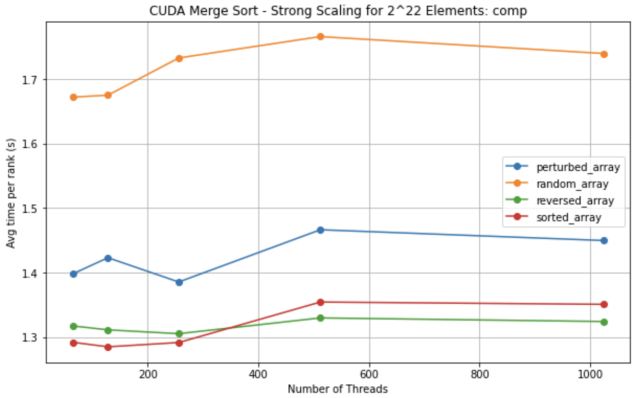
2^22 Elements

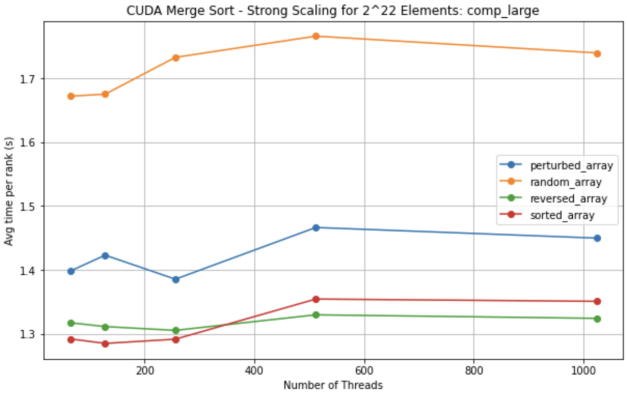


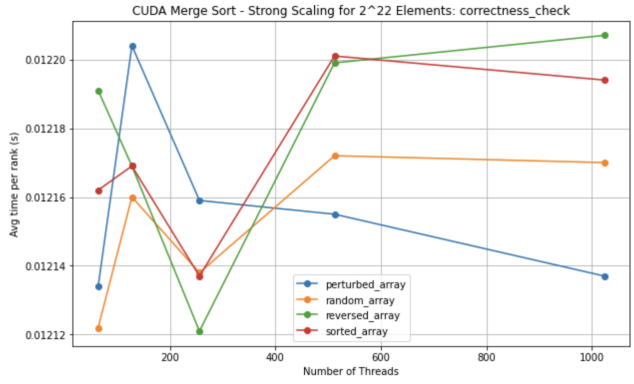


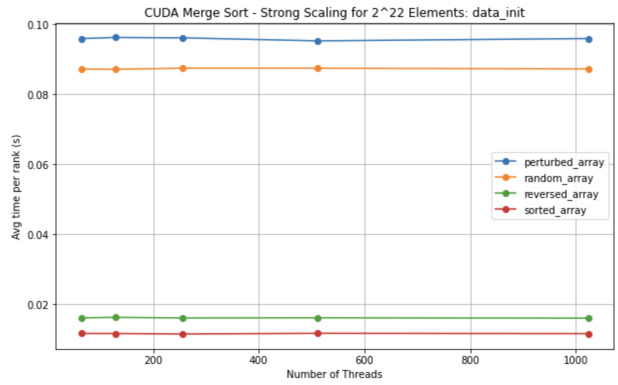




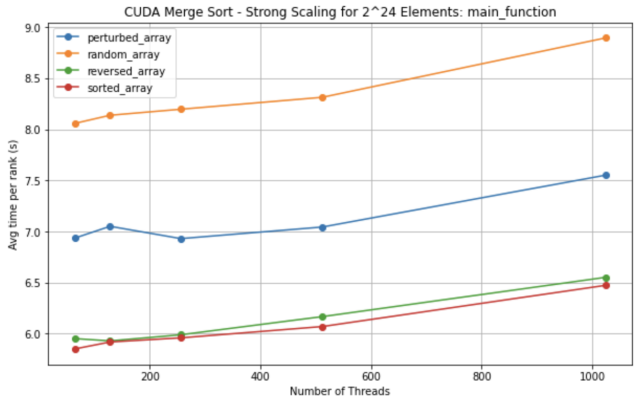


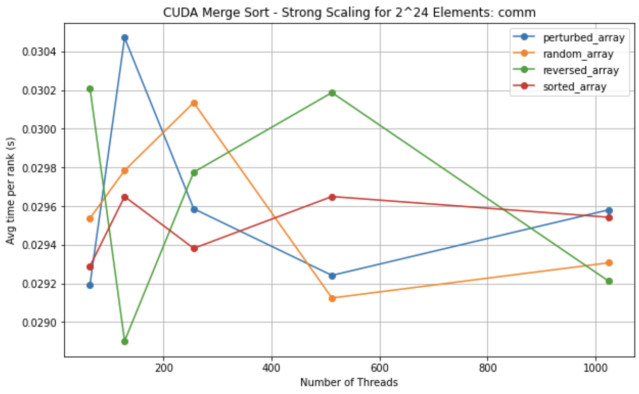


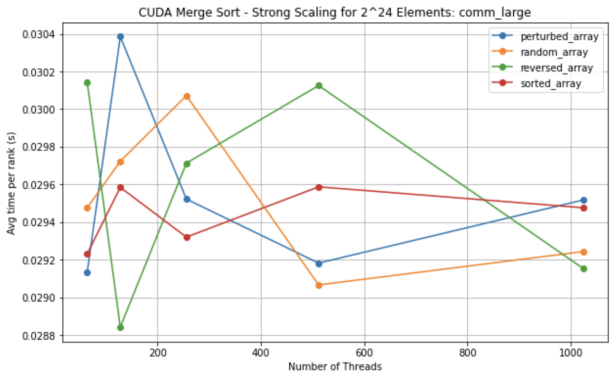


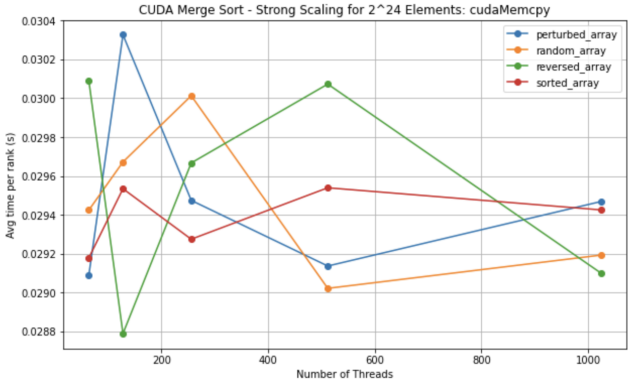


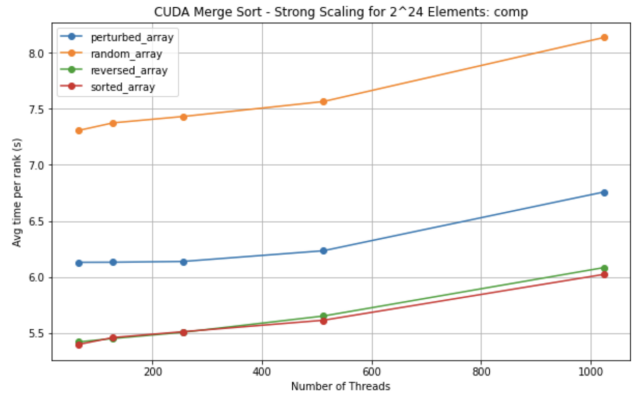
2^24 Elements

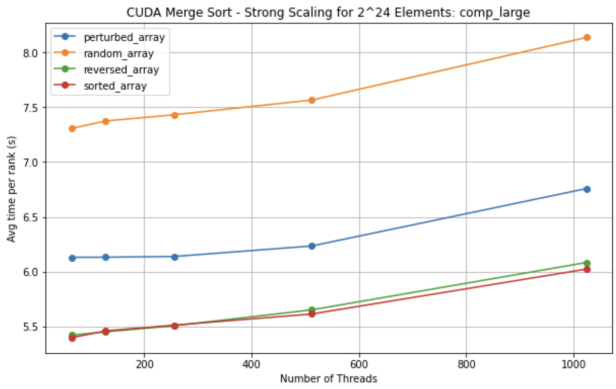


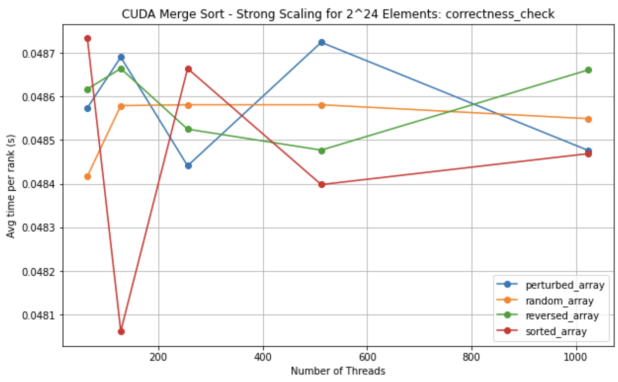


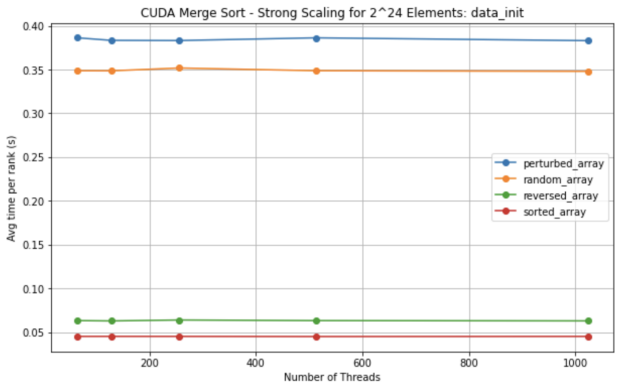




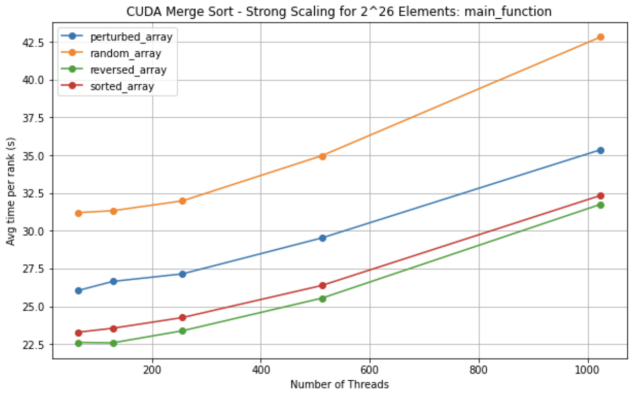


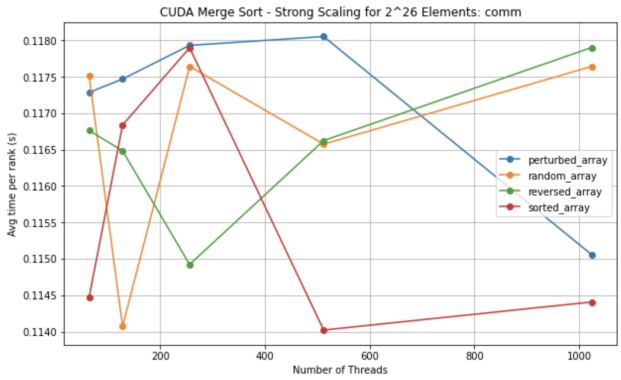


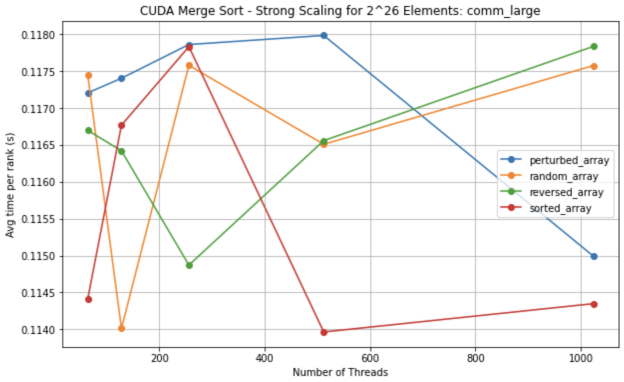


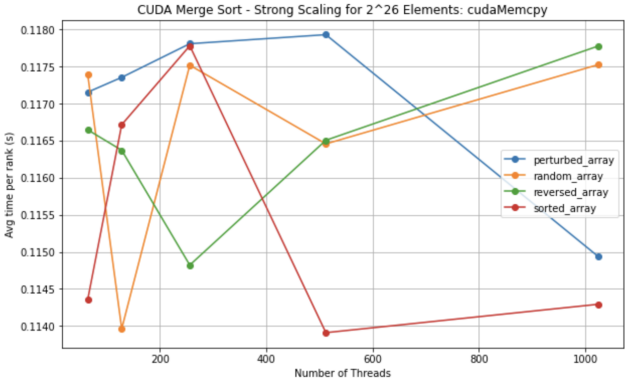


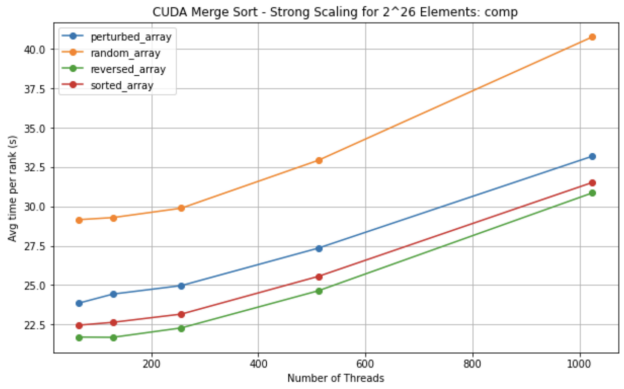
2^26 Elements

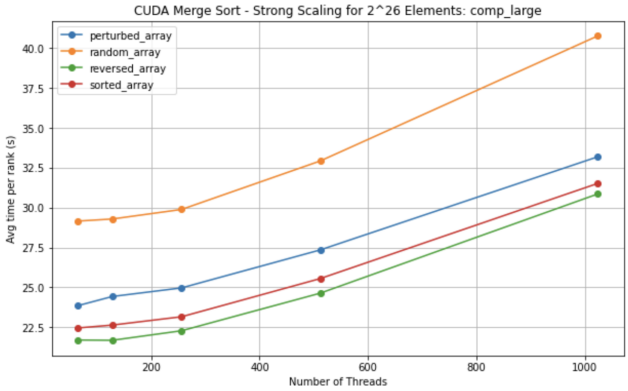


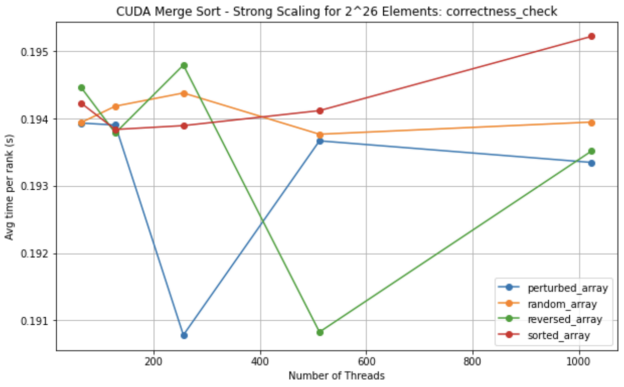


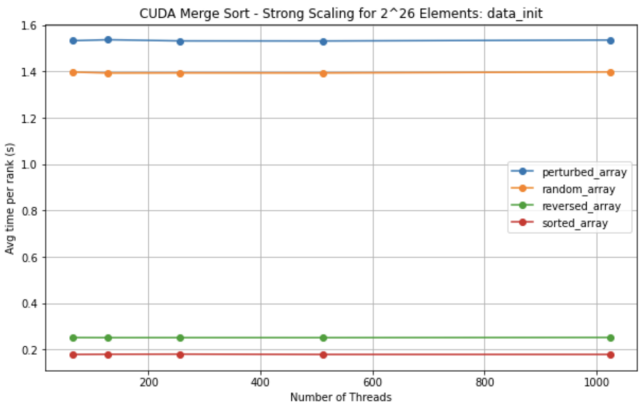




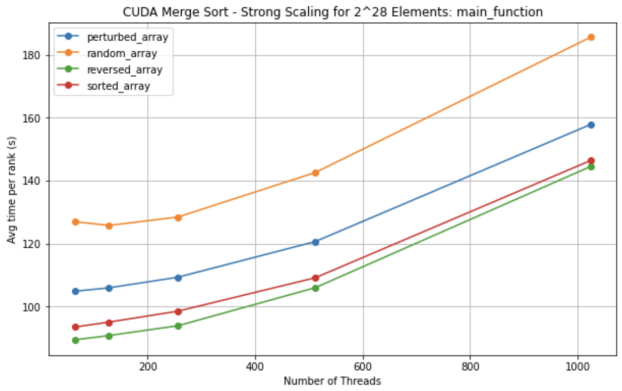


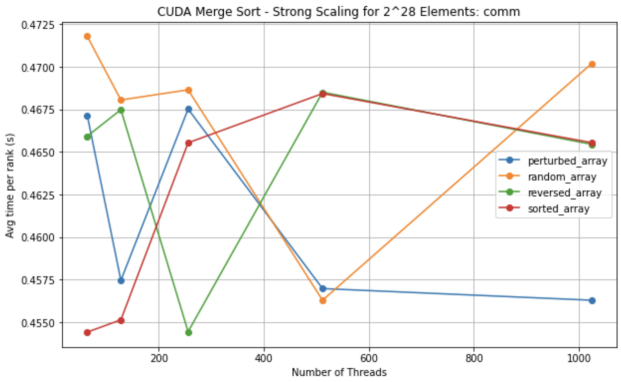


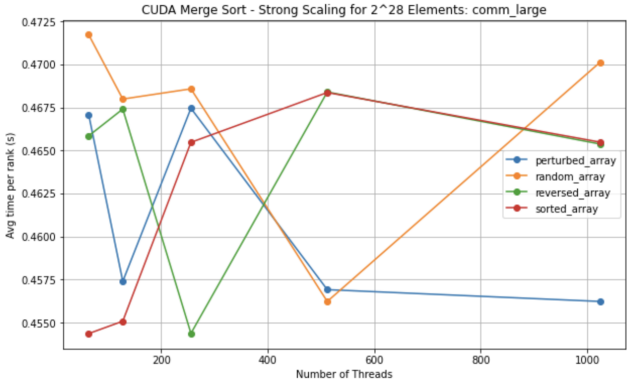


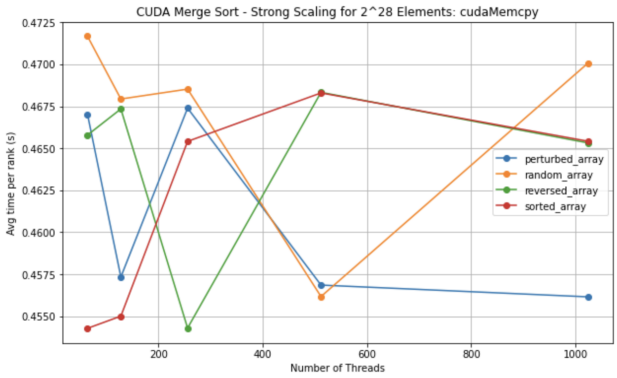


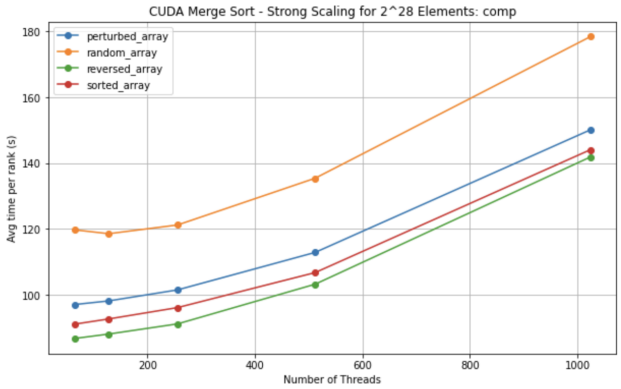
2^28 Elements

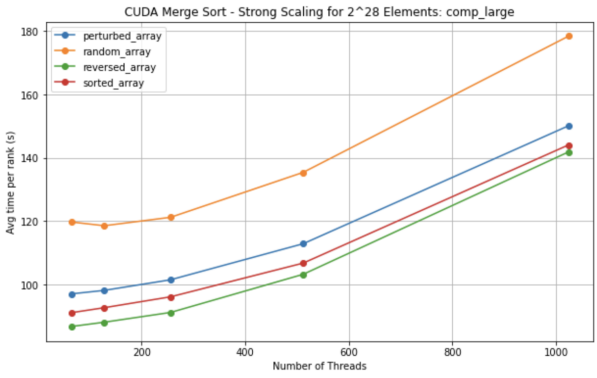


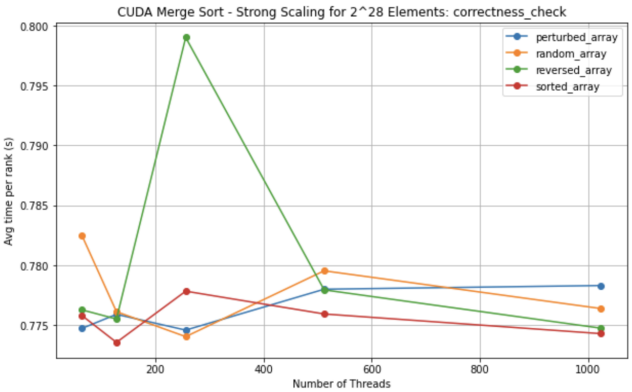


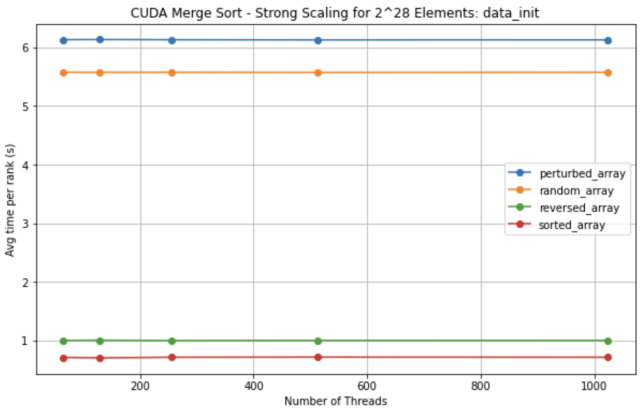












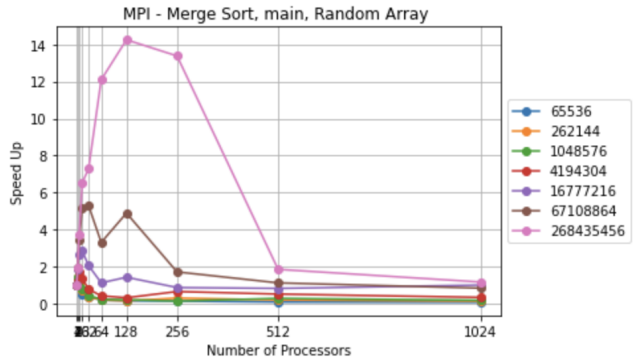
Speedup

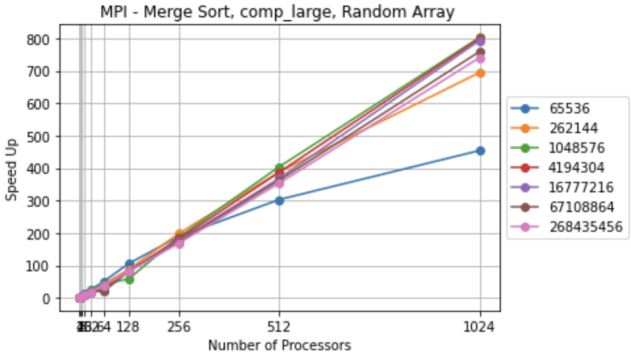
MPI

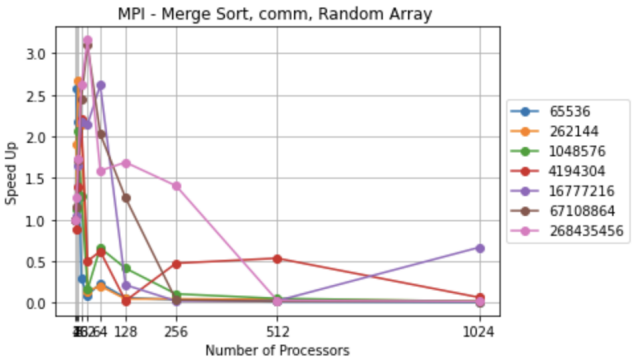
When looking at the computation speedups, we see a linear trend between the input sizes. This shows the importance of parallelism. For all sizes, as we increase the number of processors, the pure computation time gets better and better, this showing the higher speedup. We do see that speedup for the 2^16 elements begins to taper off for the comp regions, indiating that the parallelism is good for very large sizes, but smaller sizes aren't as impacts by increasing processors. This makes sense because you would spend more percentage

of your time increasing your processors and have more communication and face more overhead for the smaller values. If we look at the entire main region as a whole, we can see that the larger the input the size, the more speedup there is. This further reinforces that there is a need for parallelism, and a definite benefit. For all of the input sizes, there seems to be a peak speedup at 256 processors before decreasing again. After 256 processors, there may be so much overhead that we lose overall performance. It is a little difficult to look at it easily for the smaller sizes, but the speedup graphs tend to all show that a bigger input size benefits from this parallelism. We also see that for communication, speedup decreases for more processsors, again reiterating that we are likely facing some signifcant overhead.

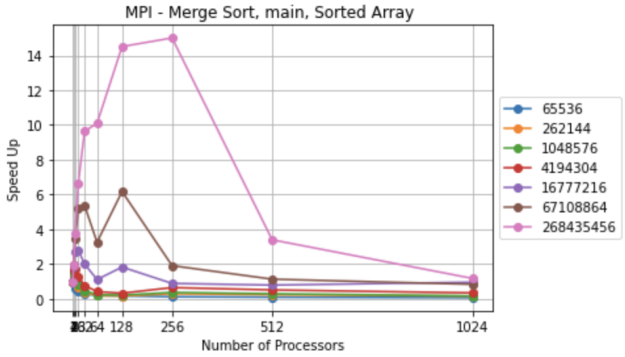
Random Array

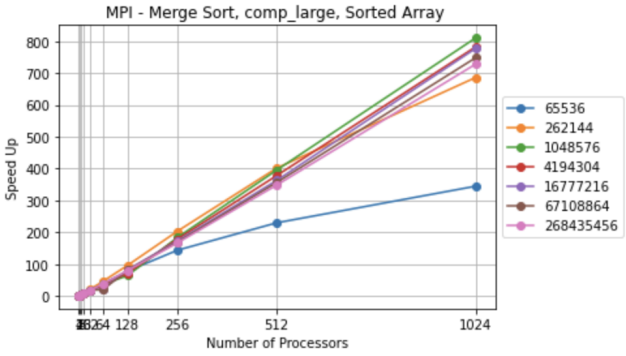


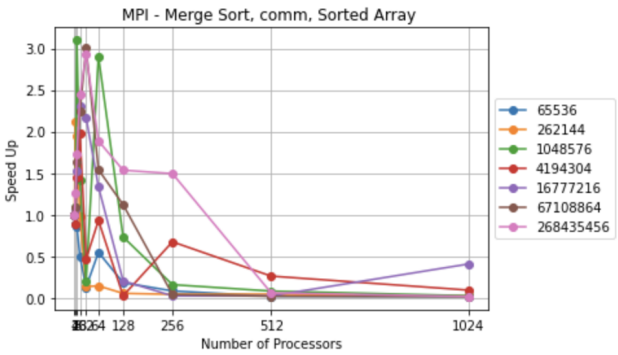




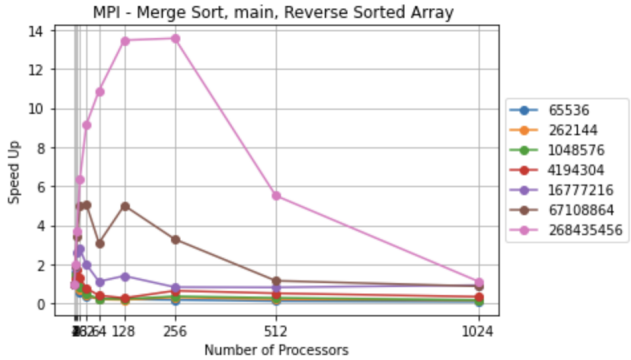
Sorted Array

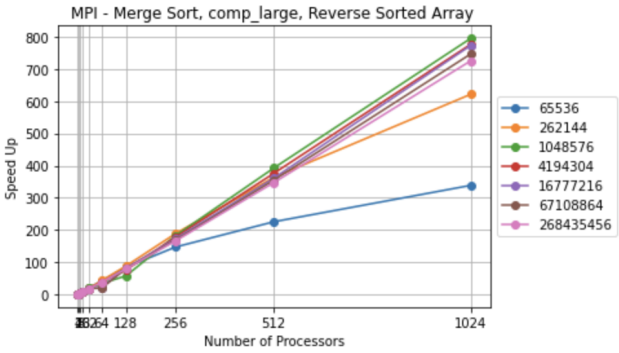


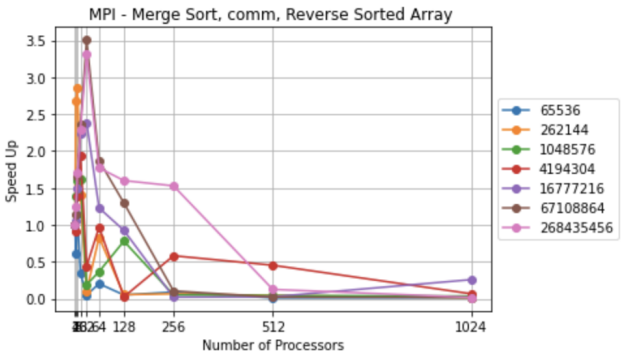




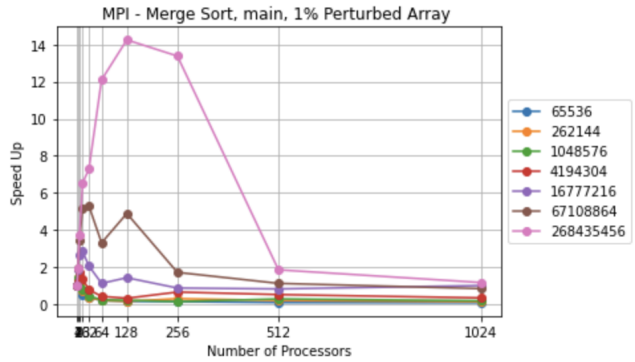
Reverse Sorted Array

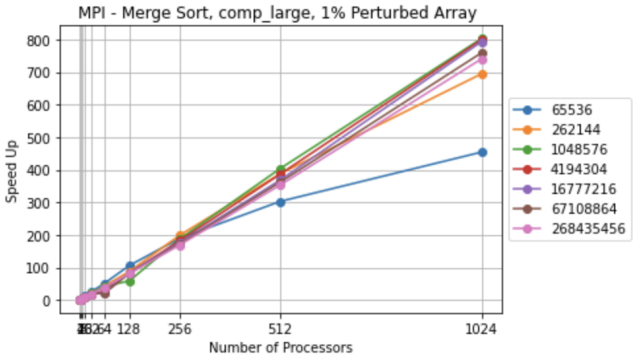


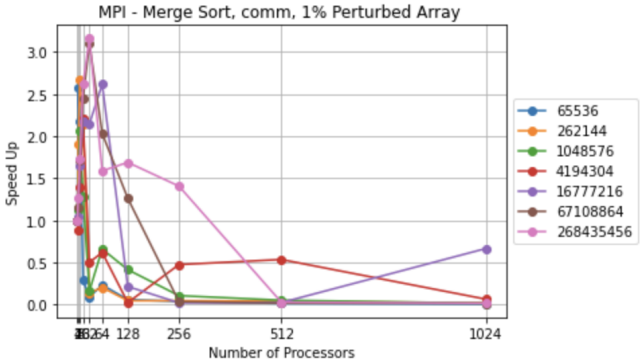




1% Perturbed Array



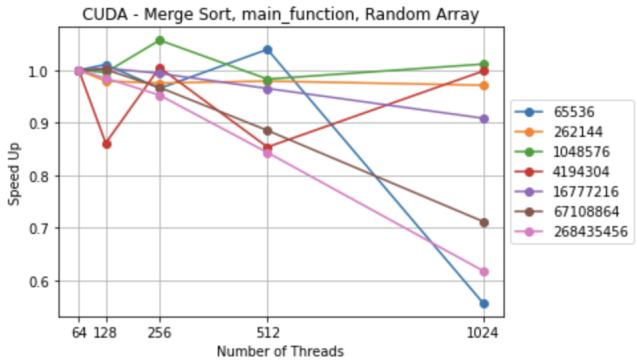


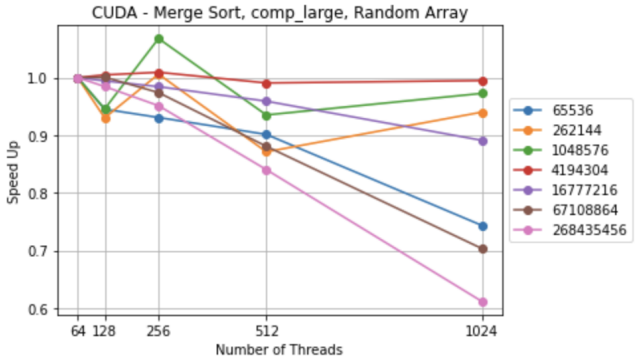


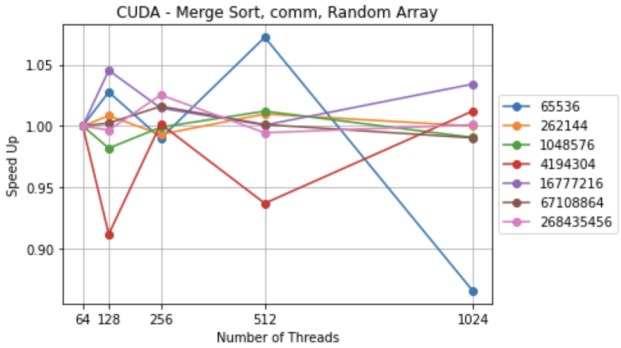
CUDA

In the CUDA speedup graphs, there doesn't seem to be any inherent benefit to paralleism. When we look at the main_function as a whole, we see that the speedup values tend to hover around 1.0. In fact, most of the lines tend to be below a value of 1, indicating that it is performing worse by being parallel. There also seems to be no pattern with the varying input size. We see that at the highest number of processors, the worst speedups are 2^16 elements and 2^28 elements, which are both the smallest and largest values respectively. It seems like as we increase processors, speedup for the communication is worst for the 2^16 element, but the computation is worst for 2^28 elements. Communication is hard to guage completely becuase there is quite a bit of fluctuation here, where computation speedup hovers around 1.0 and steadily decreases for most of the input sizes. This could also be because theres no real inter-process communication for CUDA like there is for MPI.

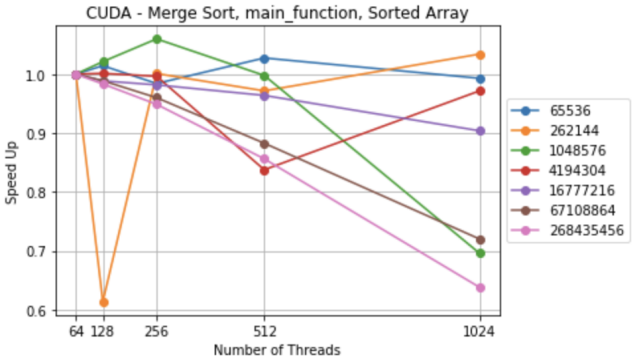
Random Array

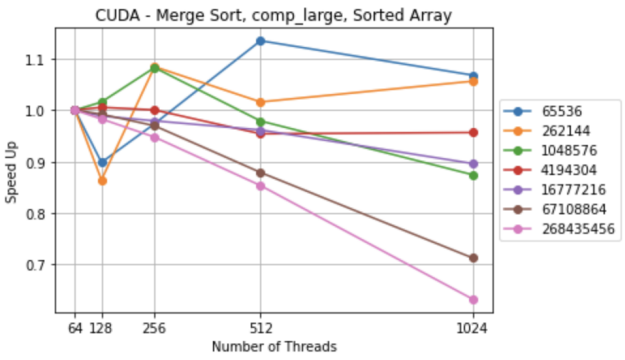


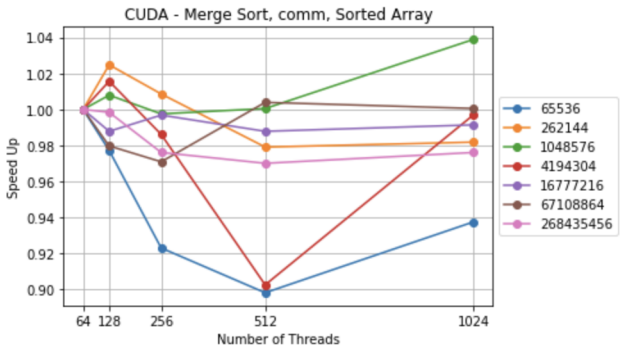




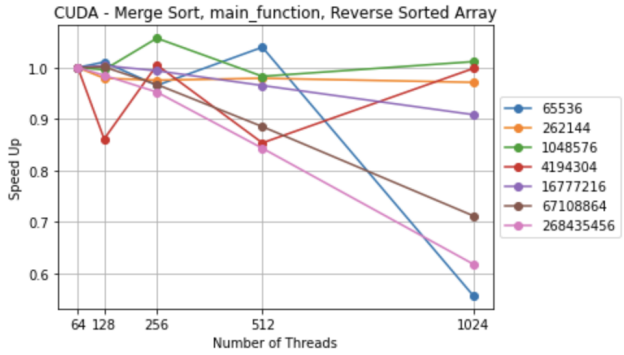
Sorted Array

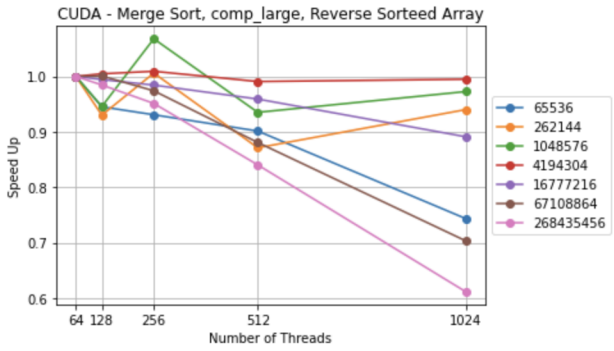


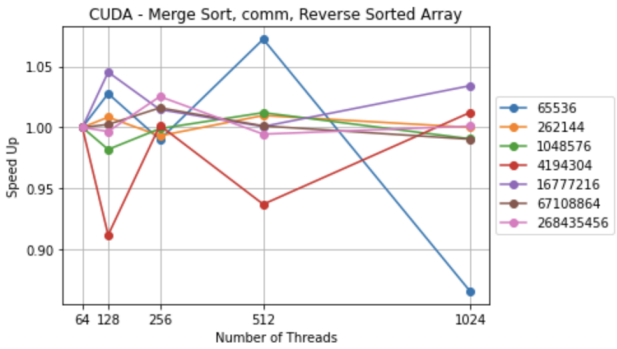




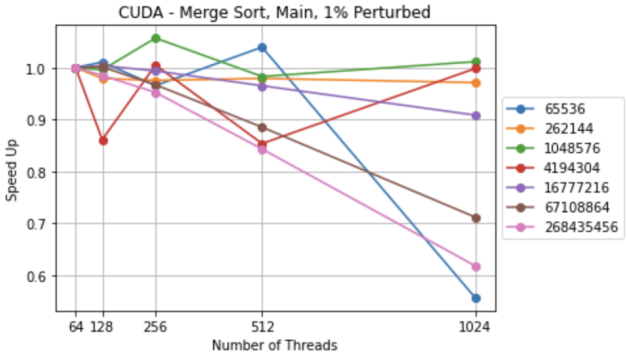
Reverse Sorted Array

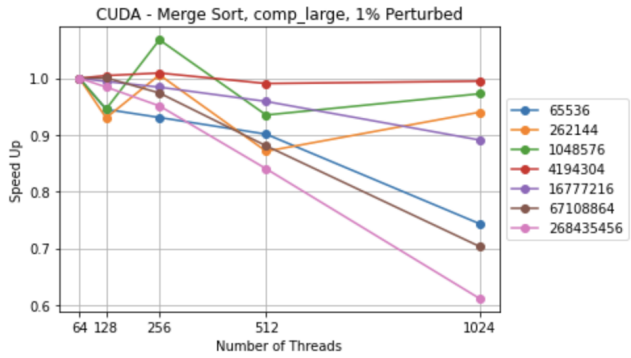


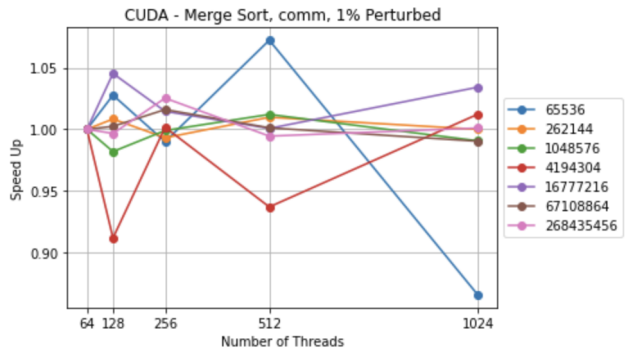




1% Perturbed Array







Bitonic Sort

Weak Scaling

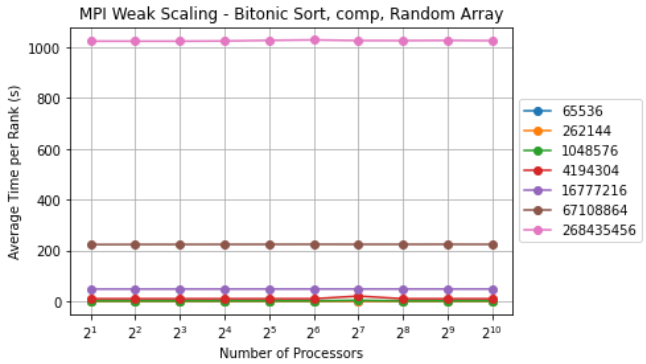
MPI

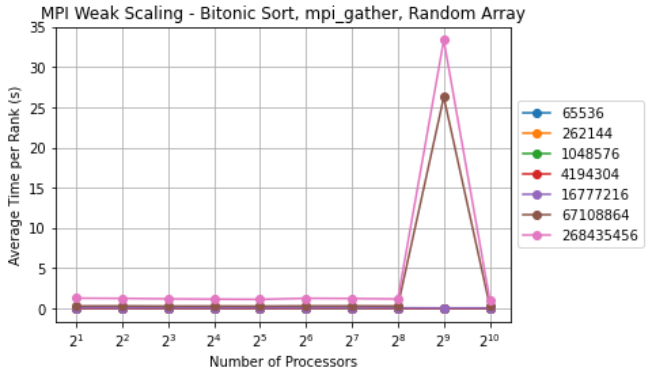
Like Shawn for merge sort, I also opted to assess the average time per rank in relation to the number of processors instead of total time since total time invariably increases with the addition of processors, as it

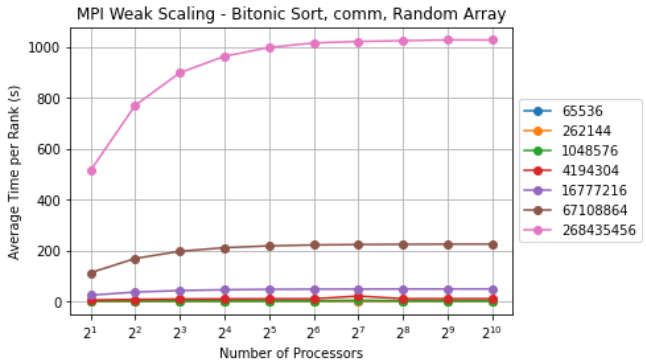
represents an aggregate of all times across all processors.

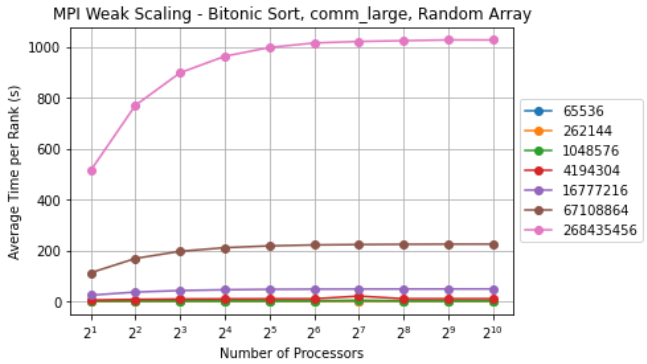
For this analysis, I begin with the weak scaling graphs for MPI across input types. First, we look at the randomly sorted arrays. The general trend is that most lines remain constant over the increase in the number of processors. Looking at the comp function, the largest input size (2^28) takes much longer by a much larger margin than the other differences which is likely because of other overhead in the large input size. The second largest input size (2^26) also takes much longer than the other input sizes. When examining the comm graph, you can see that the increase in time overall is mostly because of the communication involved with a much larger input size. Data initialization seems to parallelize well given that it decreases over the increased number of processors for all input sizes. For the sorted array, there is a similar trend except there seems to be an outlier for 2^26 input size at 2^9 processors. There is a similar trend for the reverse sorted arrays with a spike at 2^10 processors for 2^28 input size. The 1% perturbed graphs follow very closely to the reverse sorted graphs except that they are slightly more variant throughout.

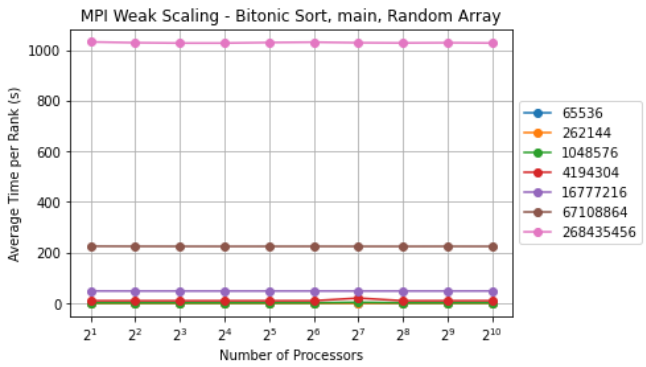
RANDOM INPUT ARRAY

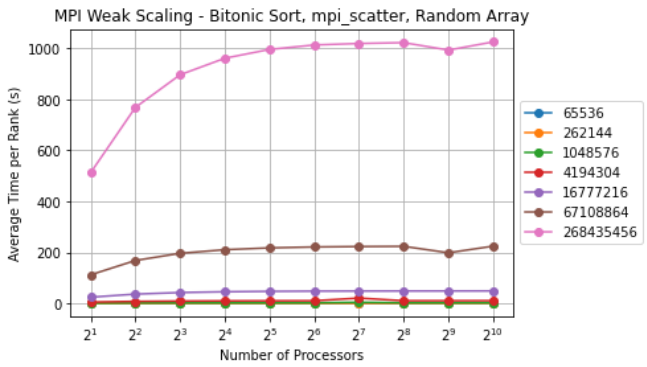


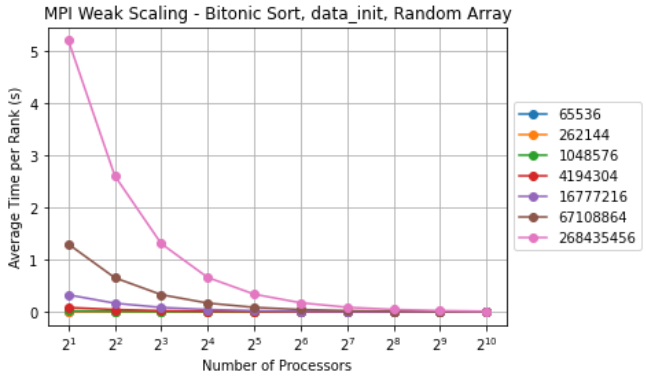


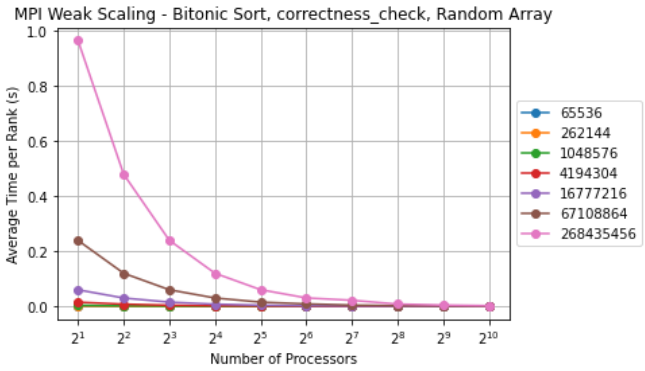


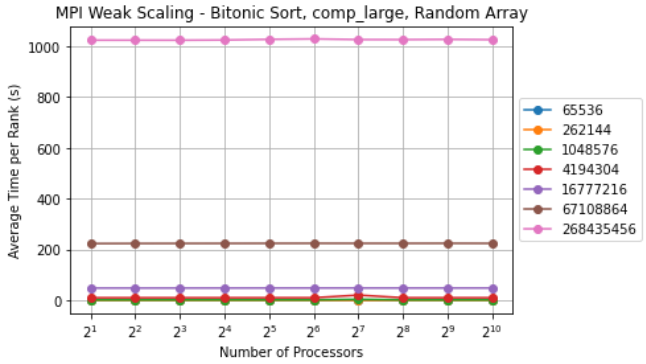




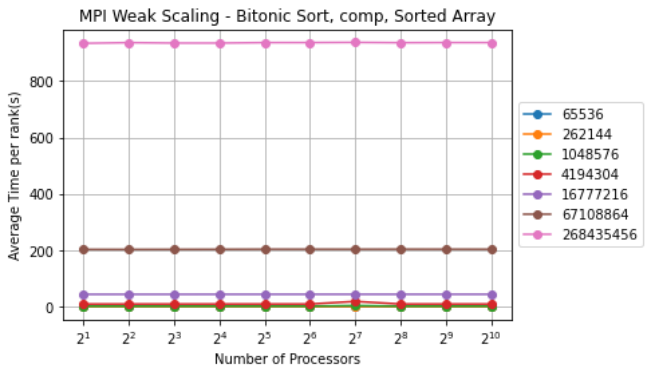


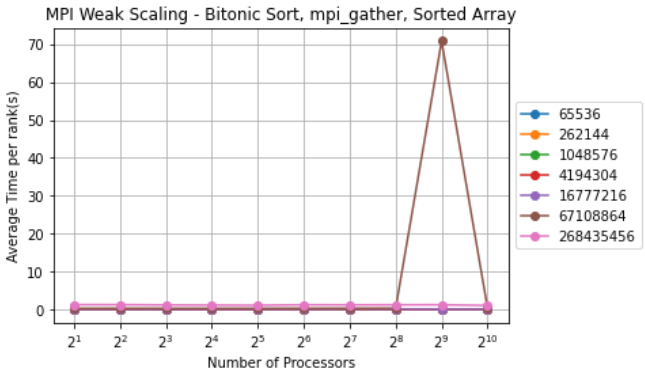


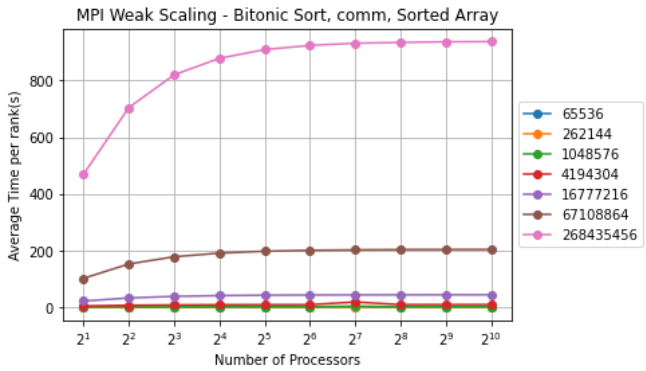


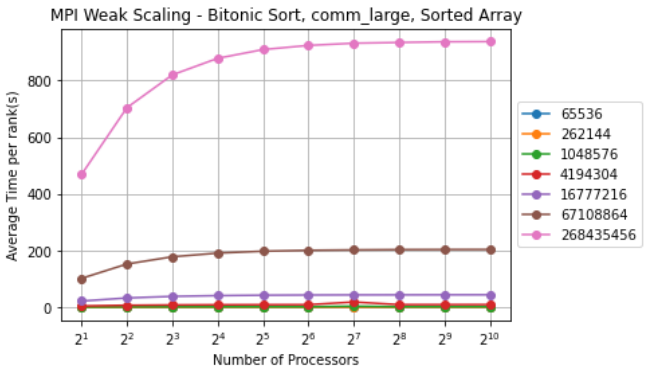


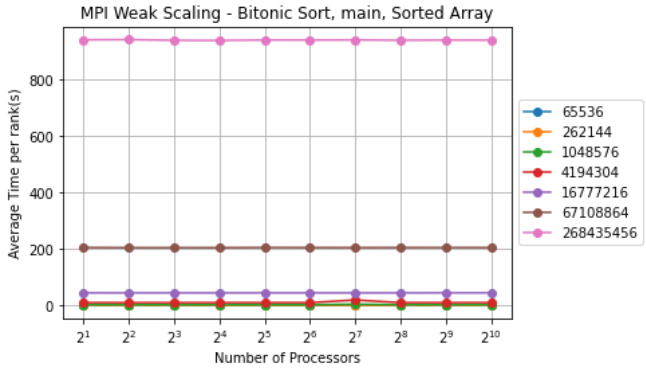
SORTED INPUT ARRAY

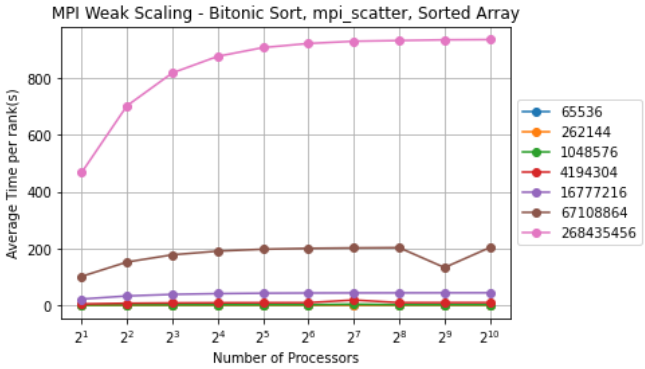


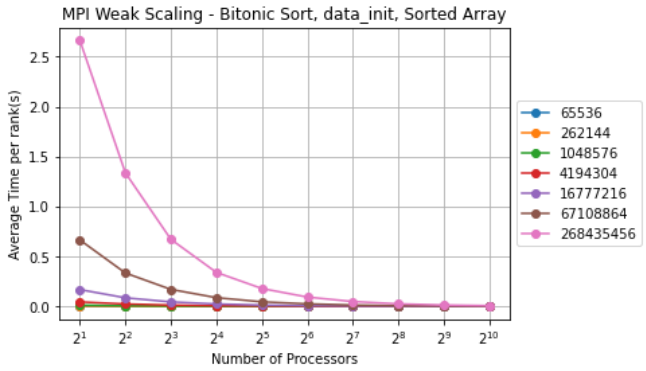


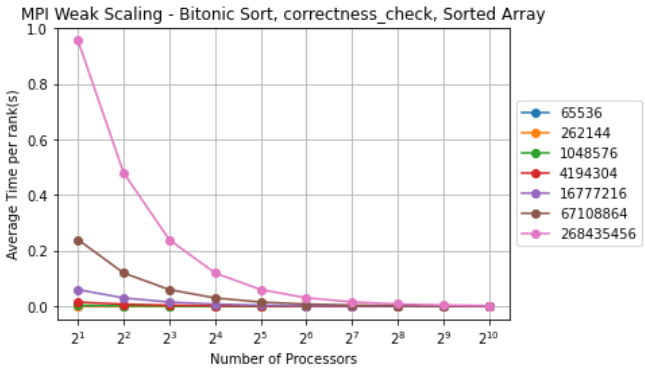


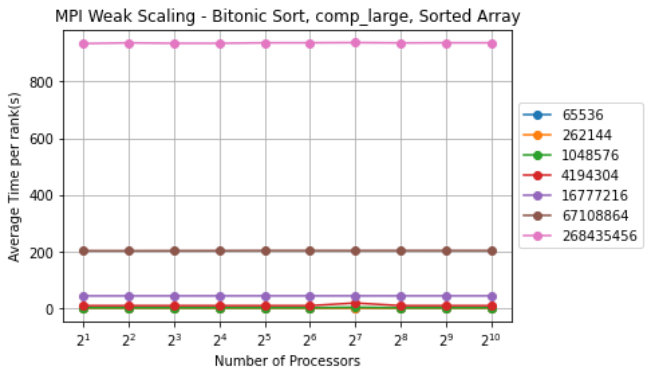




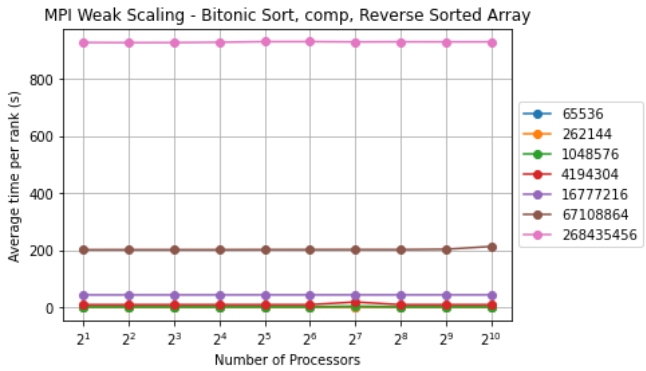


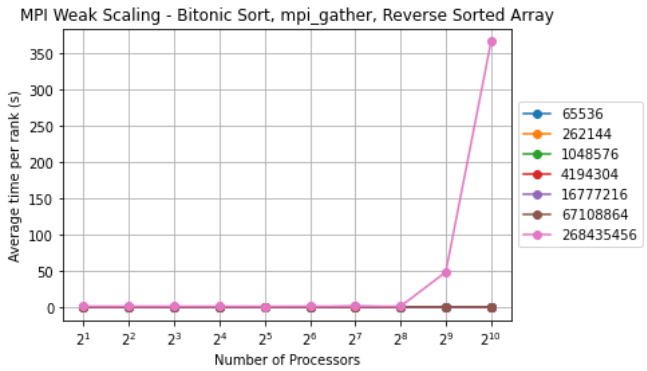


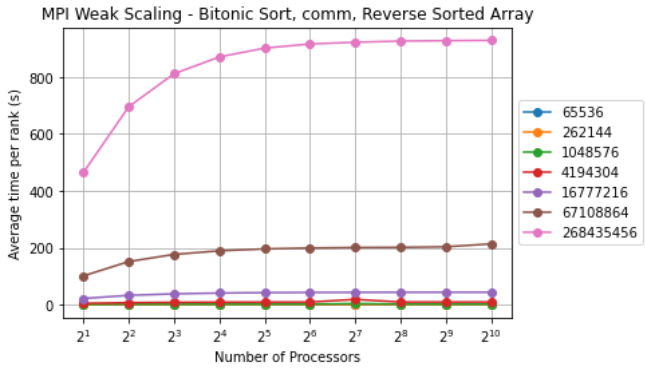


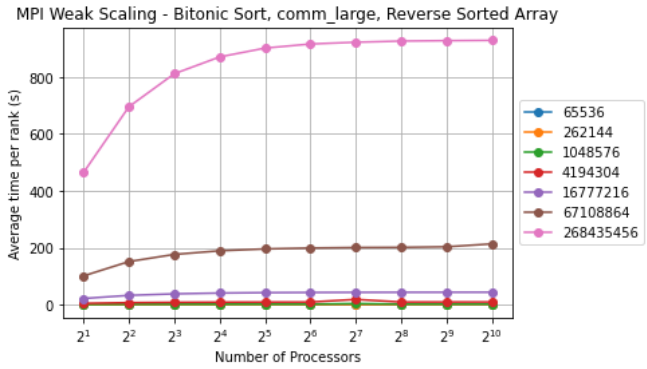


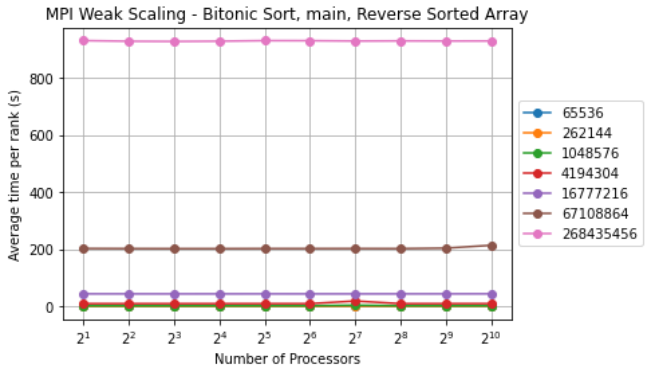
REVERSE SORTED INPUT ARRAY

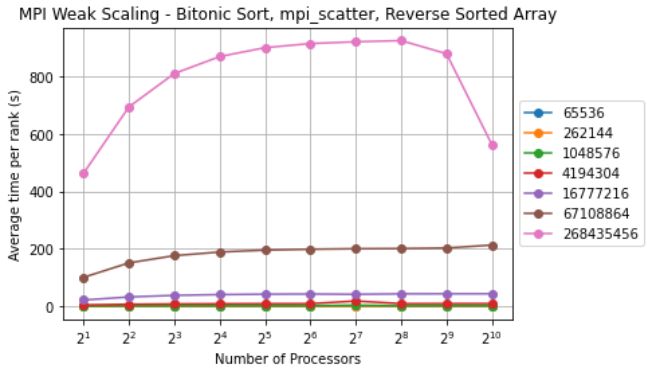


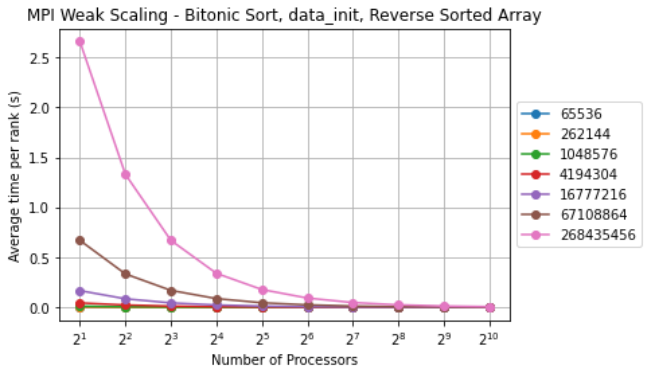


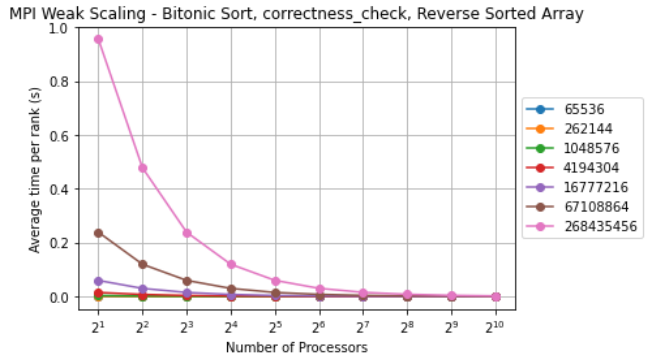


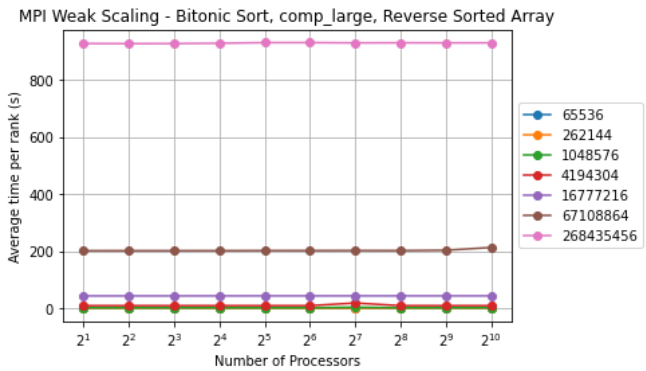




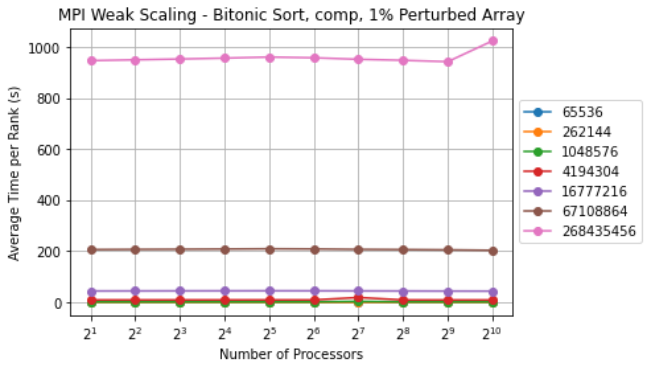


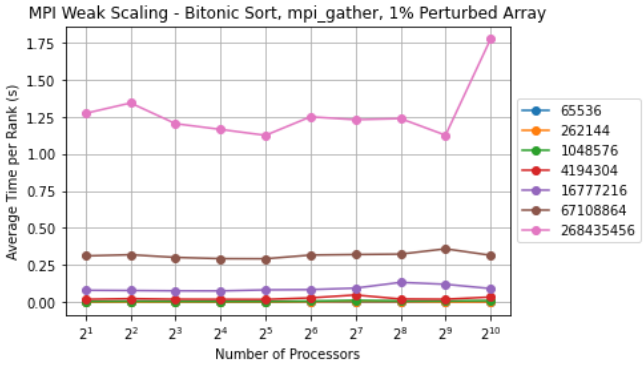


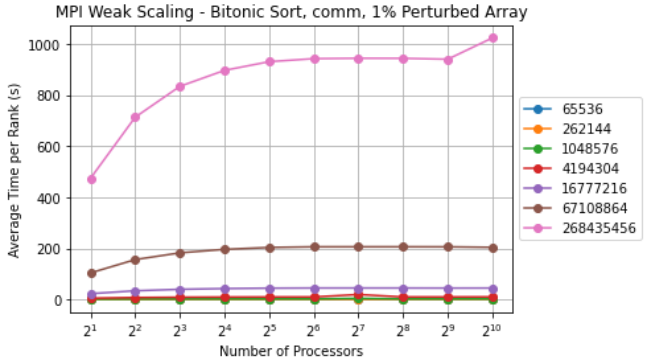


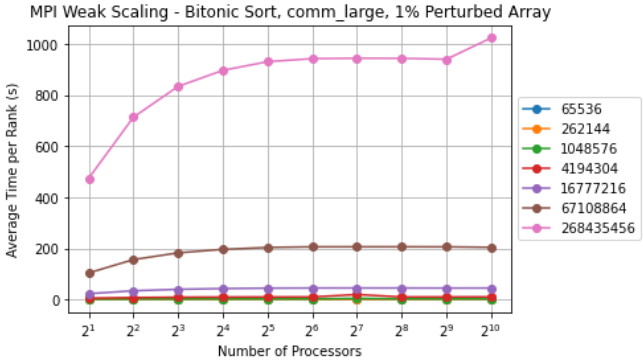


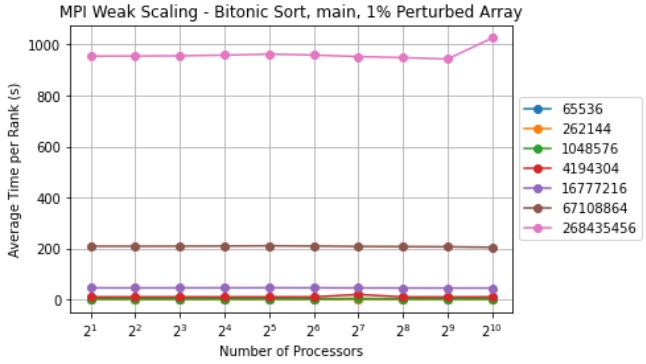
1% PERTURBED INPUT ARRAY

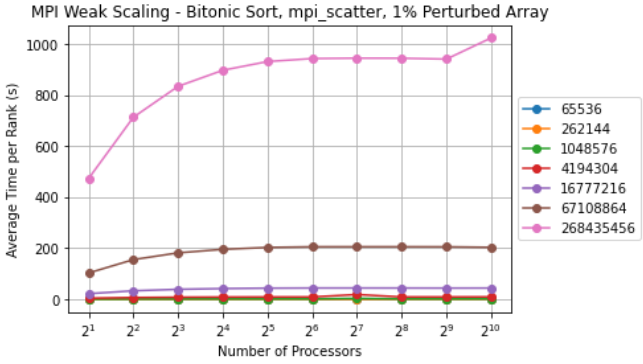

MPI Weak Scaling - Bitonic Sort, comp, 1% Perturbed Array
Average Time per Rank (s)
Number of Processors
65536
262144
1048576
4194304
16777216
67108864
268435456

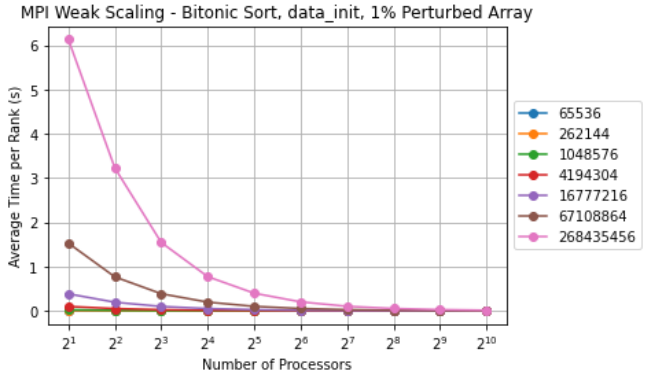

MPI Weak Scaling - Bitonic Sort, mpi_gather, 1% Perturbed Array
Average Time per Rank (s)
Number of Processors
65536
262144
1048576
4194304
16777216
67108864
268435456

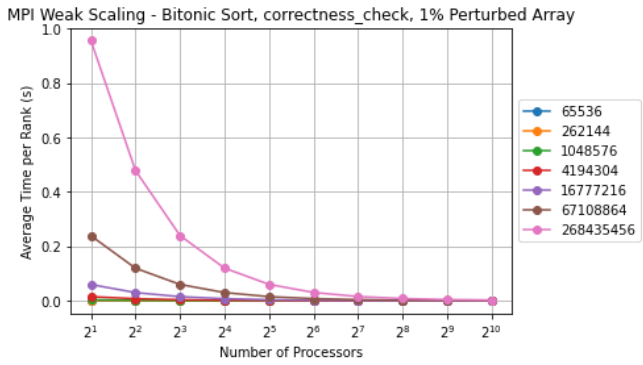

MPI Weak Scaling - Bitonic Sort, comm, 1% Perturbed Array
Average Time per Rank (s)
Number of Processors
65536
262144
1048576
4194304
16777216
67108864
268435456

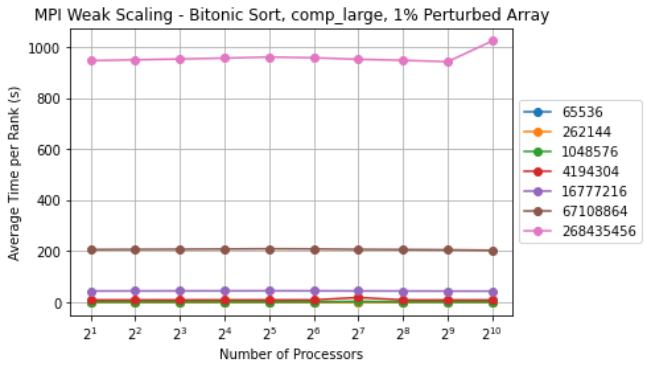

MPI Weak Scaling - Bitonic Sort, comm_large, 1% Perturbed Array
Average Time per Rank (s)
Number of Processors
65536
262144
1048576
4194304
16777216
67108864
268435456


MPI Weak Scaling - Bitonic Sort, main, 1% Perturbed Array
Average Time per Rank (s)
Number of Processors
65536
262144
1048576
4194304
16777216
67108864
268435456


MPI Weak Scaling - Bitonic Sort, mpi_scatter, 1% Perturbed Array
Average Time per Rank (s)
Number of Processors
65536
262144
1048576
4194304
16777216
67108864
268435456


MPI Weak Scaling - Bitonic Sort, data_init, 1% Perturbed Array
Average Time per Rank (s)
Number of Processors
65536
262144
1048576
4194304
16777216
67108864
268435456

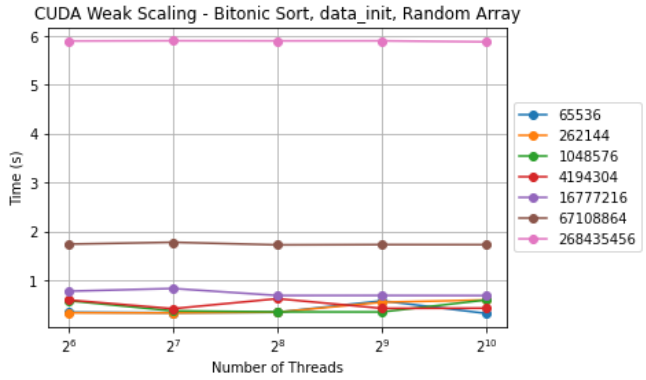

MPI Weak Scaling - Bitonic Sort, correctness_check, 1% Perturbed Array
Average Time per Rank (s)
Number of Processors
65536
262144
1048576
4194304
16777216
67108864
268435456

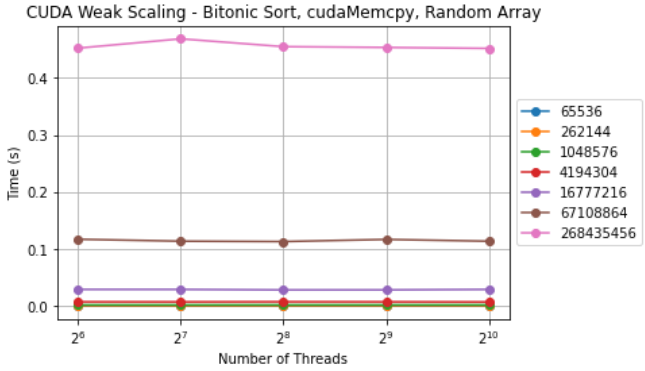

MPI Weak Scaling - Bitonic Sort, comp_large, 1% Perturbed Array
Average Time per Rank (s)
Number of Processors
65536
262144
1048576
4194304
16777216
67108864
268435456

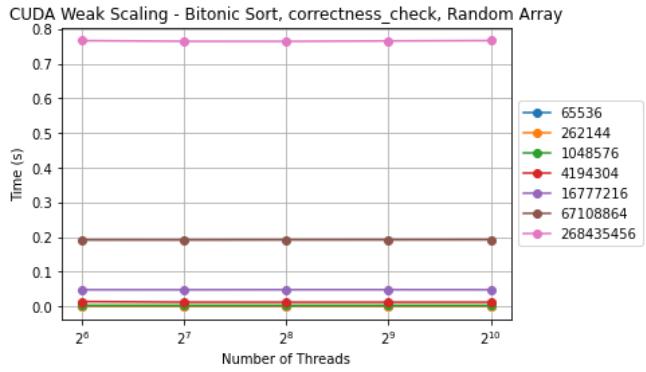
CUDA

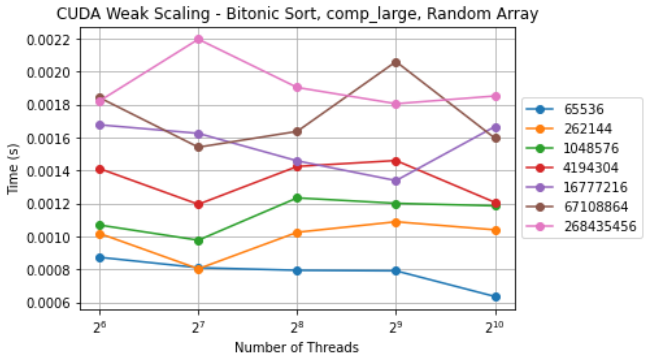
Next, we look at weak scaling for CUDA across input types. The order is the same in that we look at the randomly sorted arrays first. Similar to MPI, the general trend is pretty linear flat for the main and comm graphs. There is not a large change in time with the increased number of threads. The two larger input sizes are significantly slower than the other input sizes likely due to the overhead of many more numbers to sort. When looking at the comp graphs, it looks like there is much more variance but the y-axis scale is much smaller so there is very little variance in times in reality for all input sizes. The sorted array follows the same trend. The reverse sorted array sees a spike after 2^9 threads likely due to overhead. The 1& perturbed graphs see slightly more variance throughout since they are randomly perturbed.

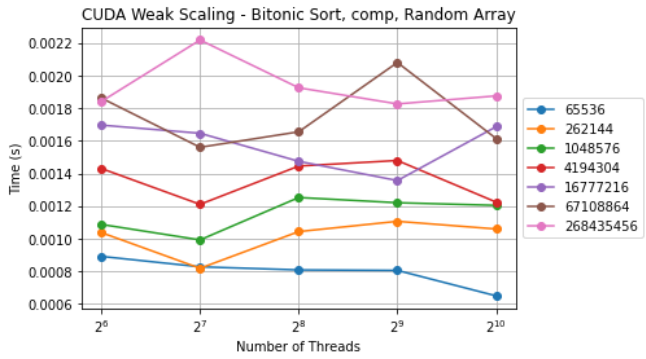
RANDOM INPUT ARRAY

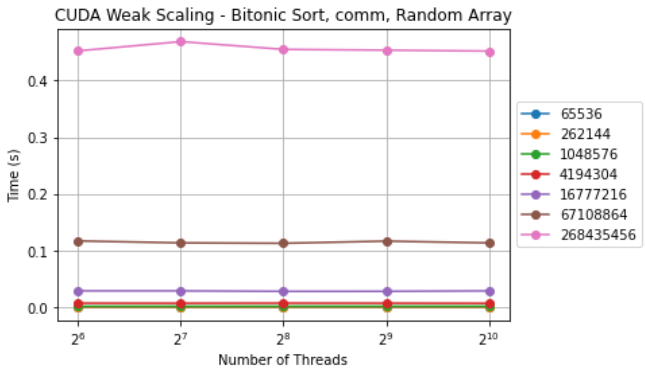


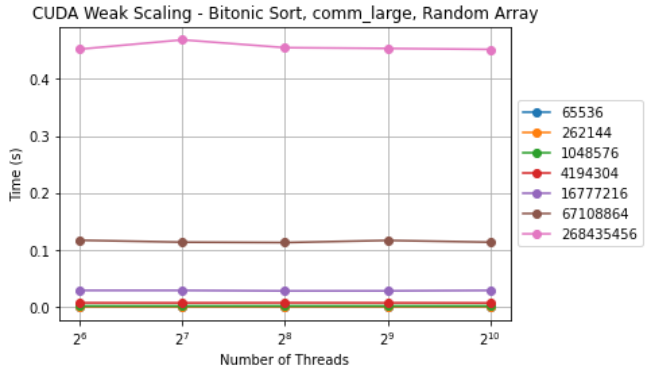


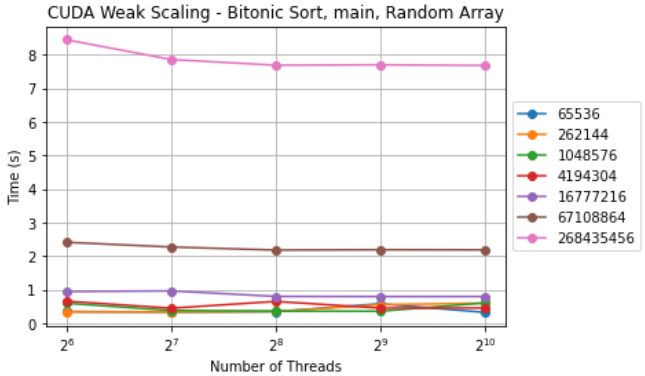




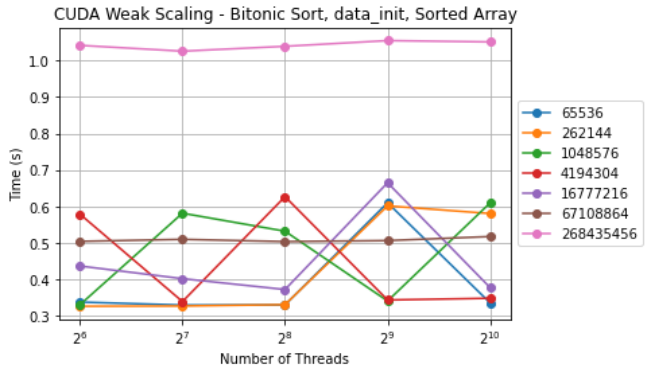


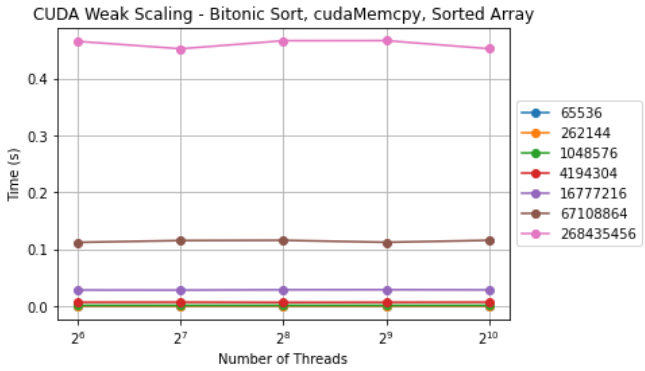


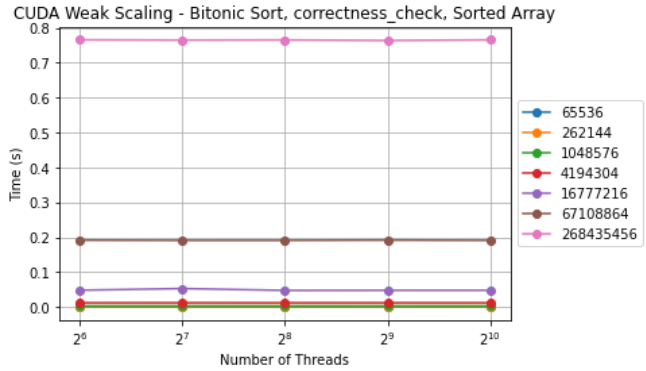


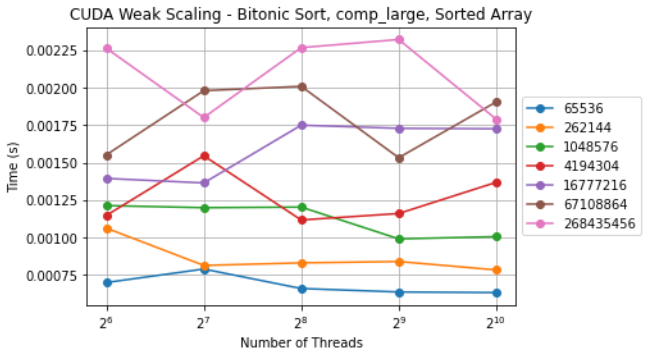


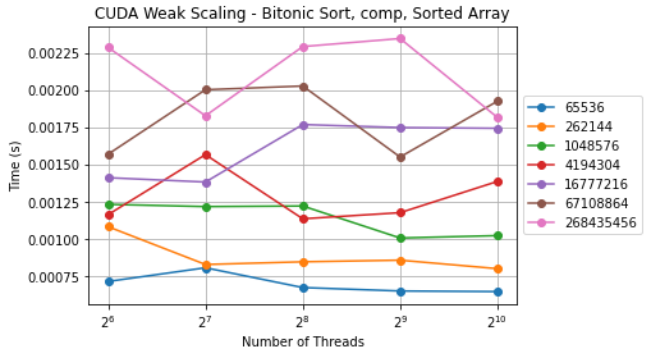
SORTED INPUT ARRAY

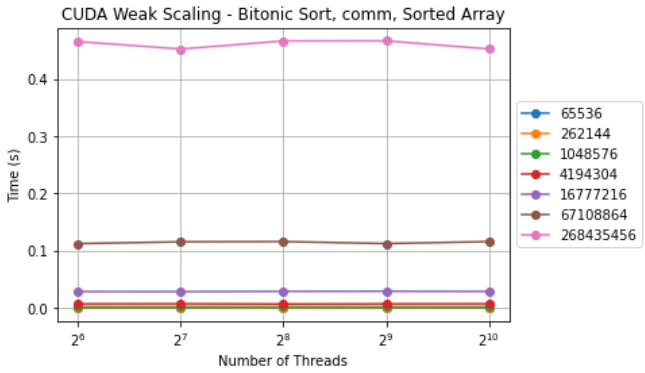


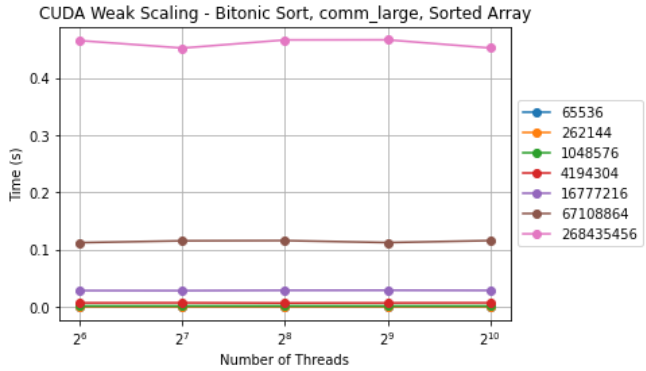


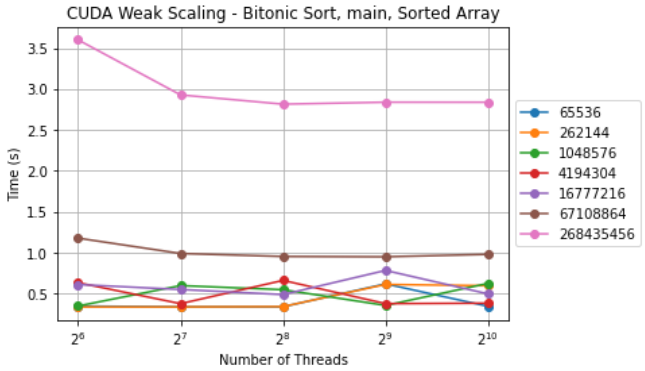




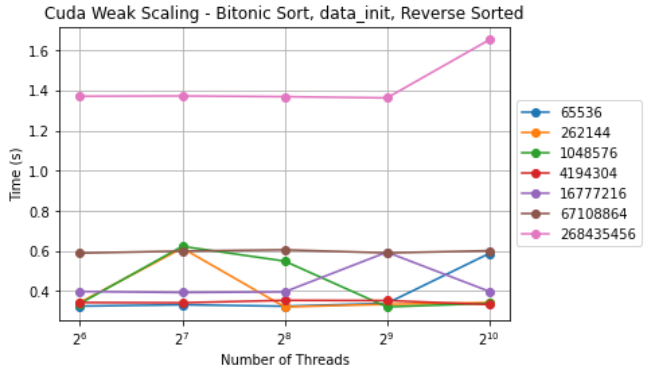


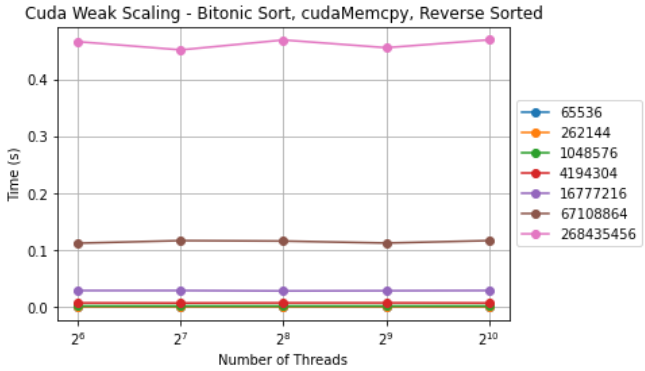


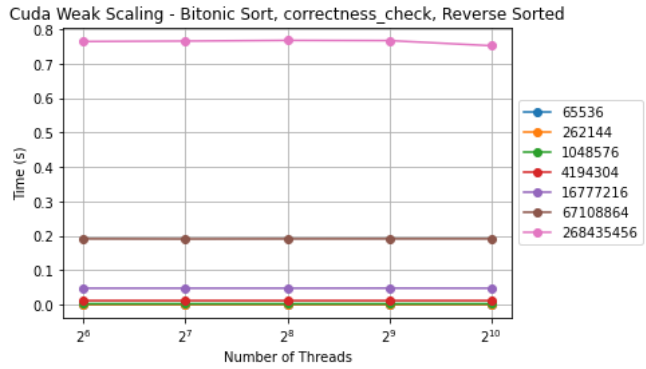


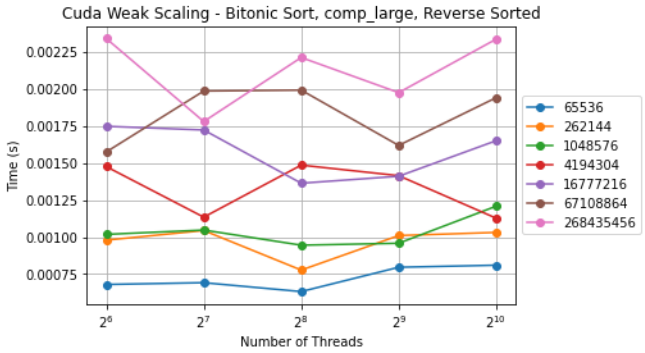


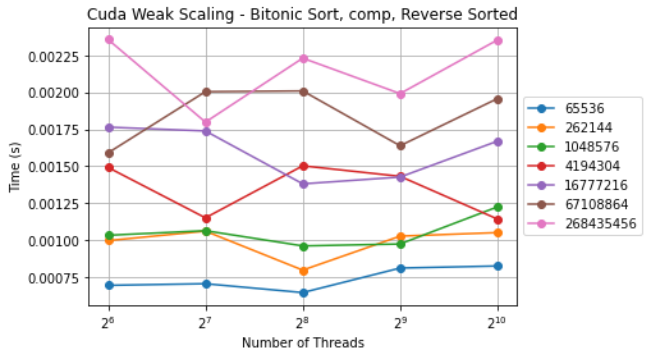
REVERSE SORTED INPUT ARRAY

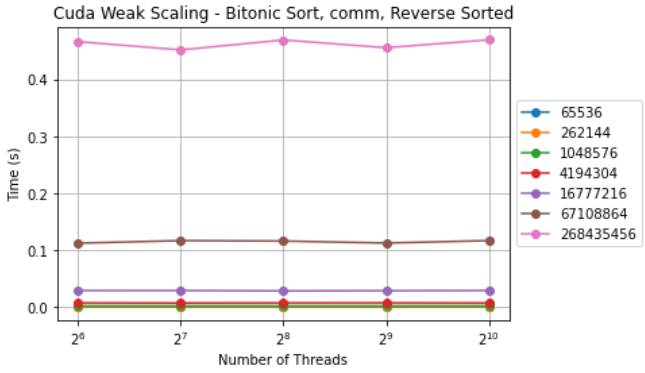


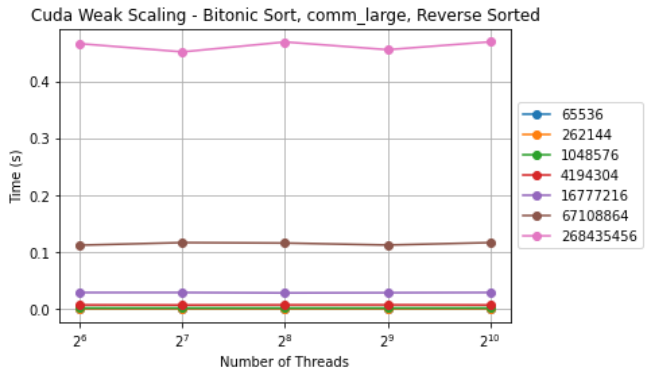


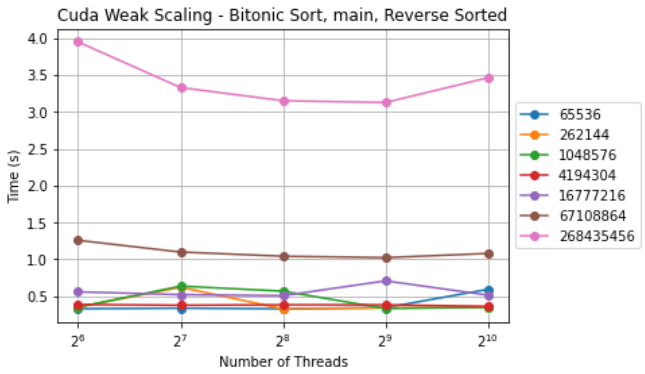




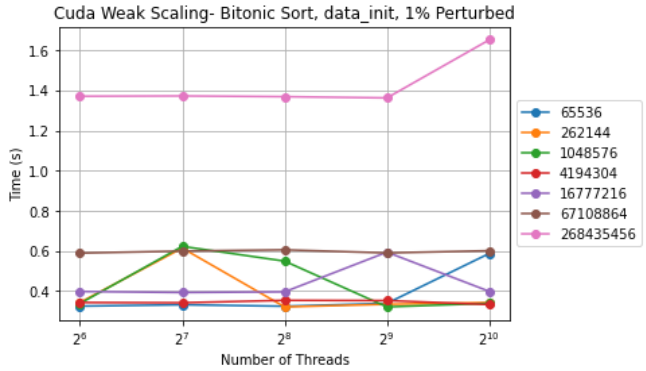


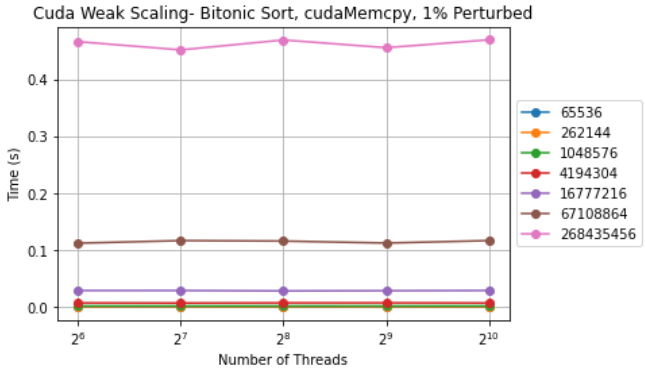


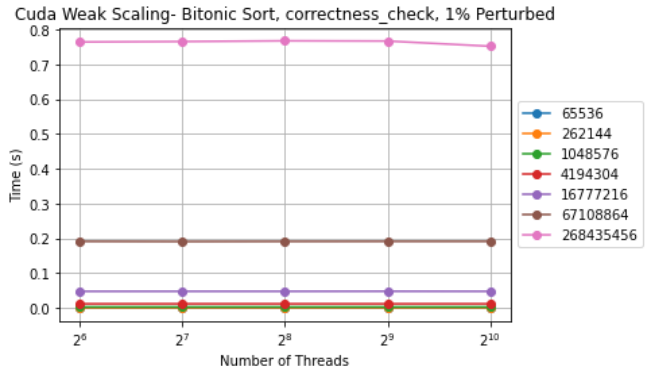


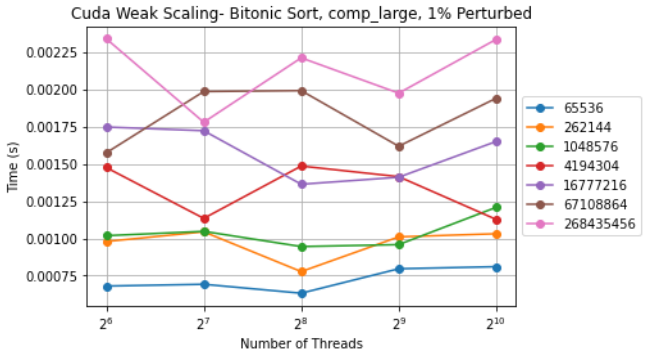


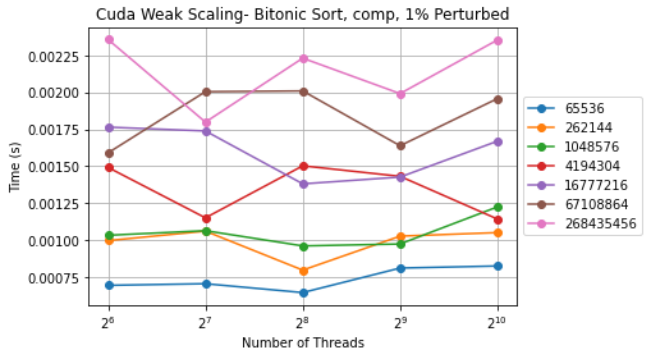
1% PERTURBED INPUT ARRAY

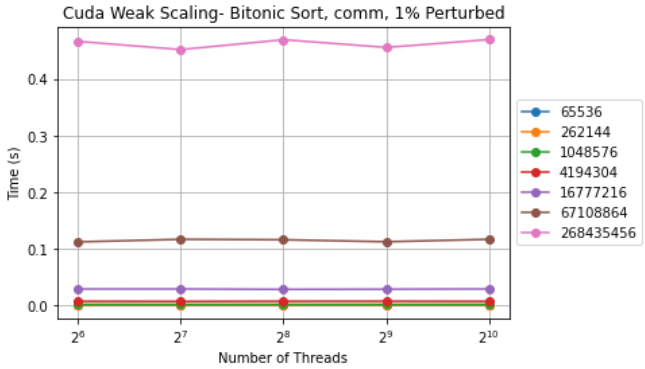


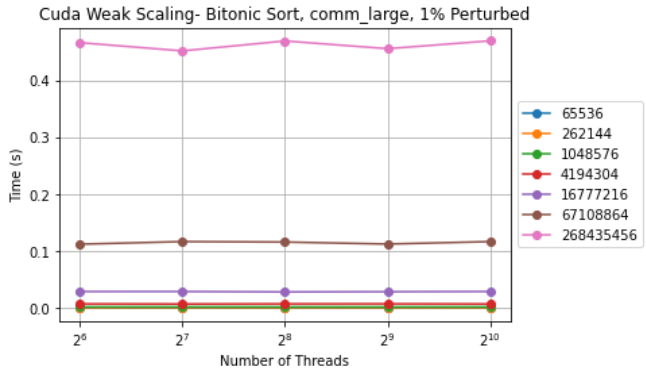


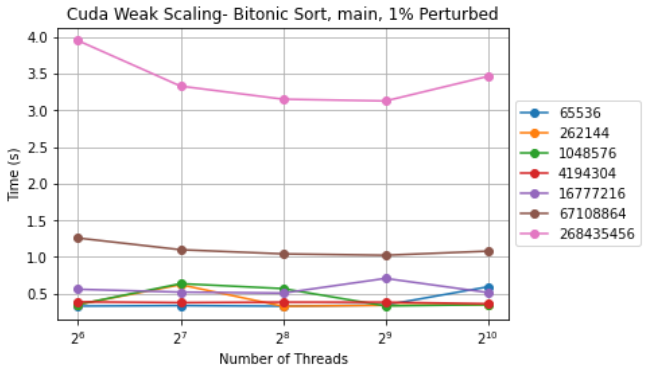












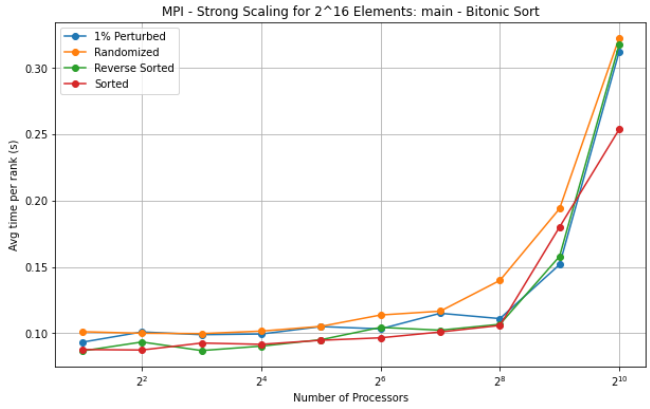
Strong Scaling

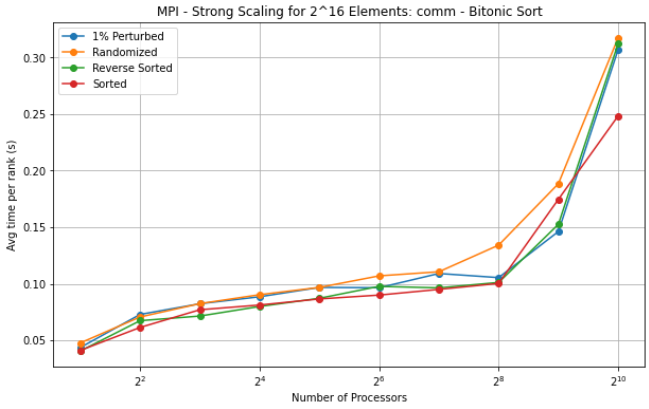
MPI

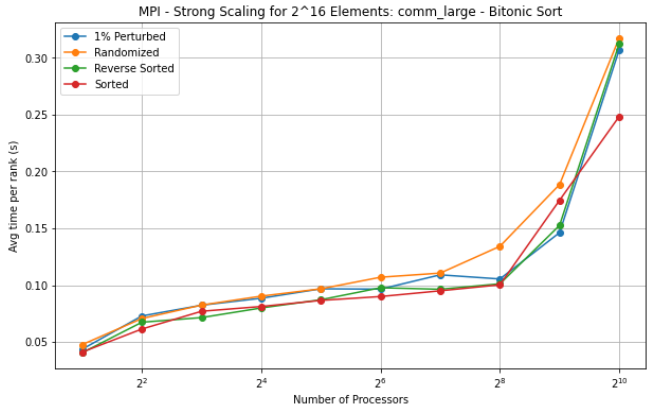
For the next analysis, we look at strong scaling MPI across different input sizes. For the smallest input size (2^16), time seems to increase with the number of processors but the margin is very small and negligible. As we increase input size, time generally decreases with the number of processors which is the best scenario showing that parallelization is effective. The comm graphs show an increase in time likely since increase the input size generally takes longer to communicate. For the larger input sizes, there is likely an outlier at 2^7 processors. But this is an interesting point to note since it seems like it is the same outlier for these graphs.

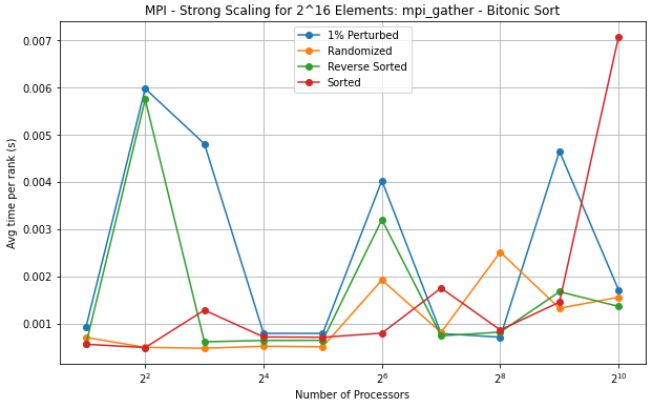
Additionally, for the larger input sizes, we can see a clear winner in input type for fastest time. The sorted array consistently exhibits the fastest runtime. This scenario represents the optimal condition for bitonic sort, as it involves zero switches for elements. The efficiency of this input type remains largely unaffected by the increase in processors, given its inherently efficient nature. The next fastest is the 1% perturbed input, which demonstrates a noticeable increase in runtime as the number of processors grows. The subsequent rankings alternate between reverse sorted and randomized inputs, as expected, considering they require a greater number of sorts.

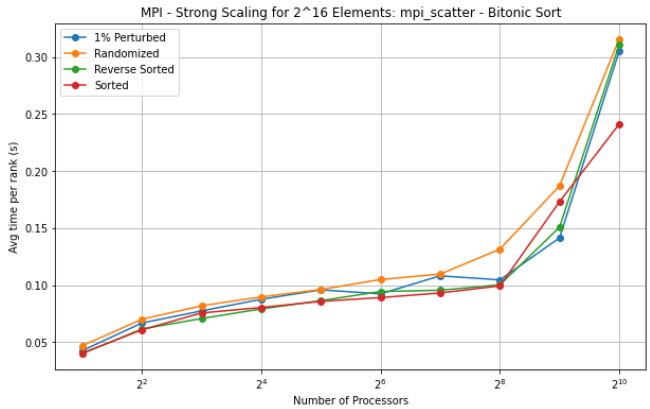
2^16 Elements

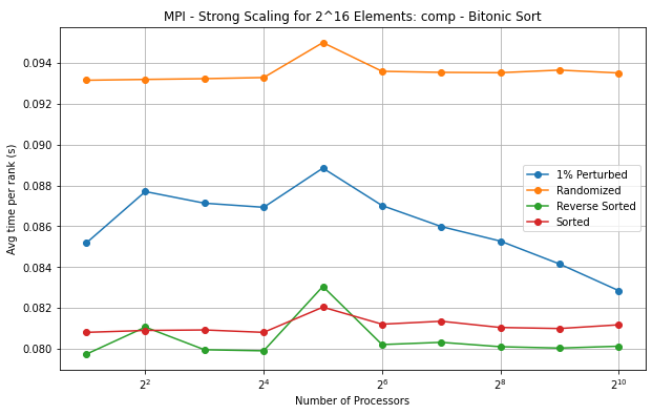


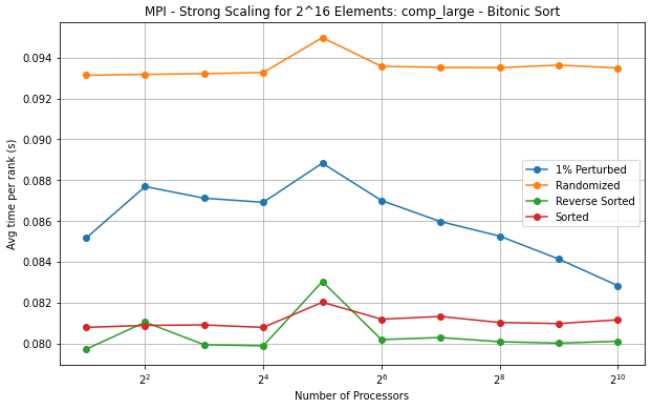


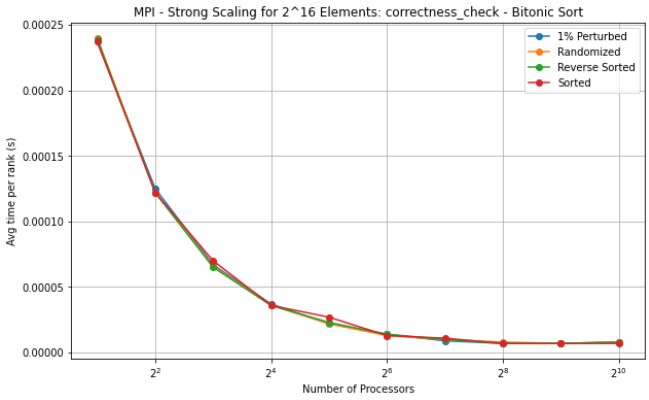


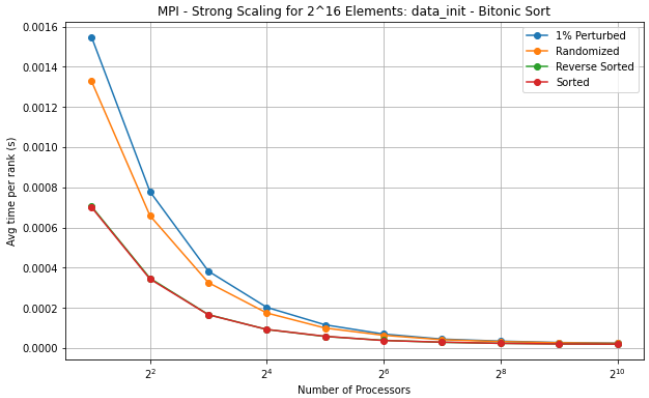




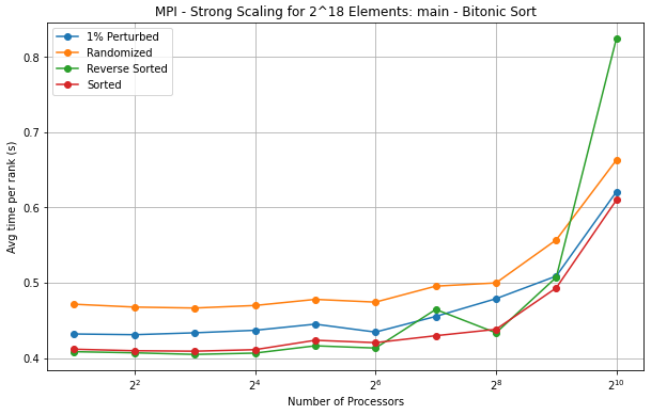


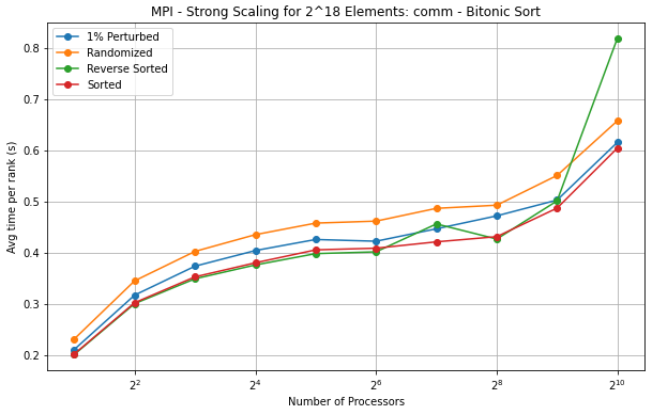


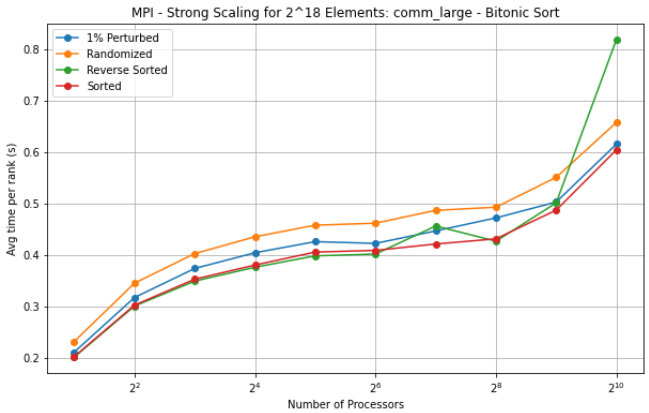


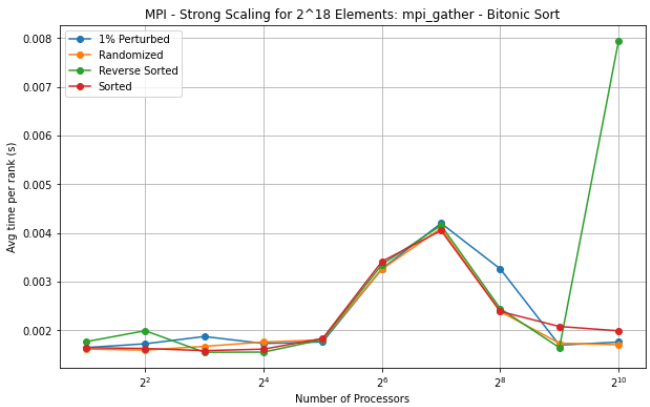


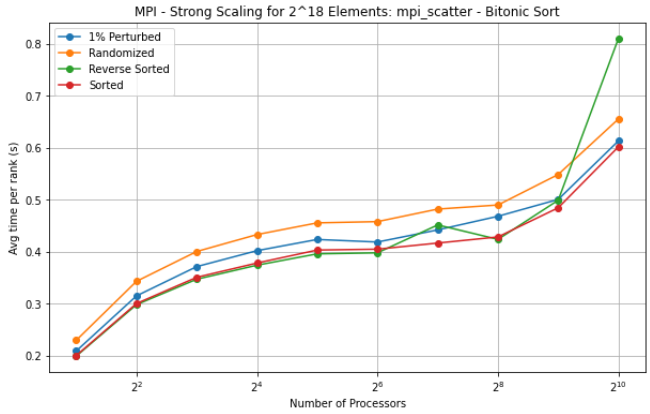
2 ^18 Elements

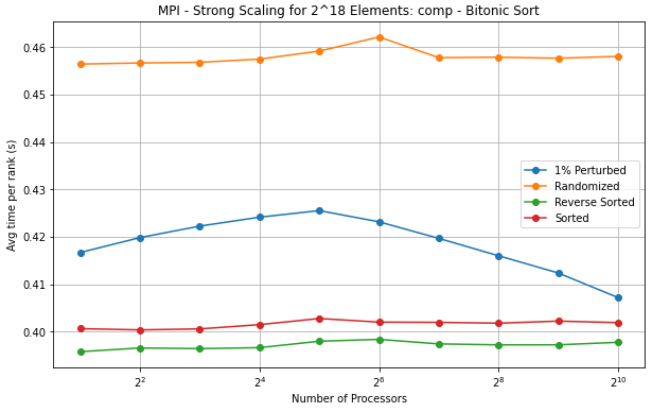


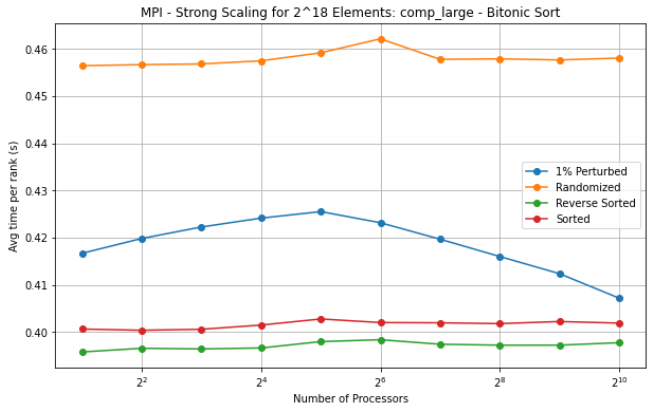


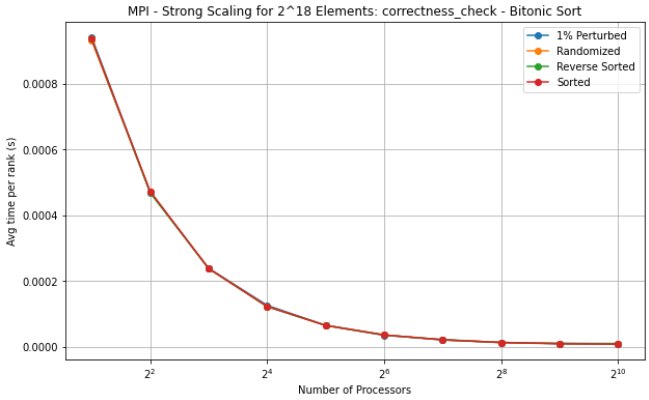


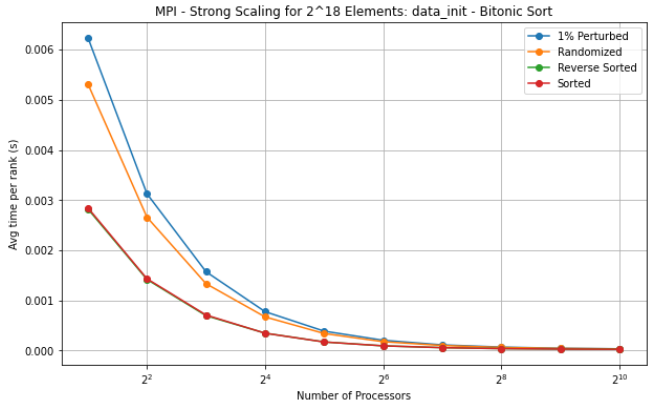




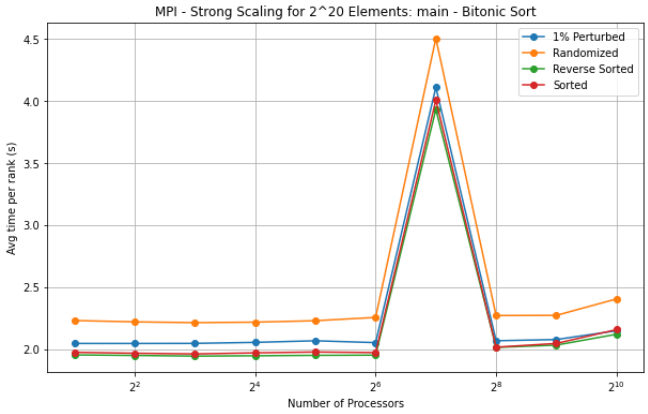


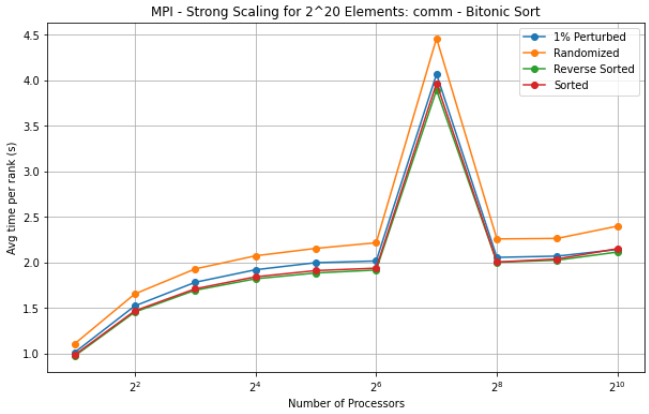


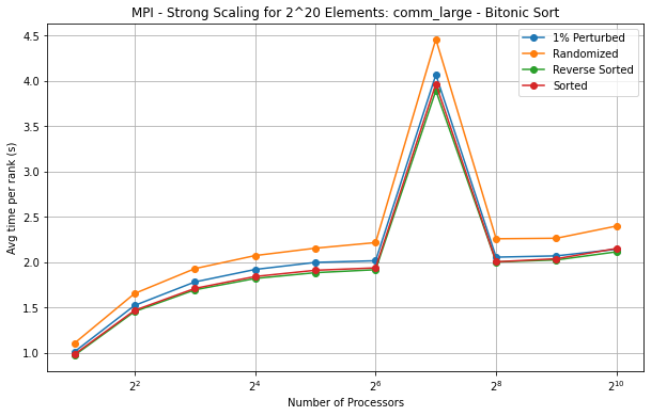


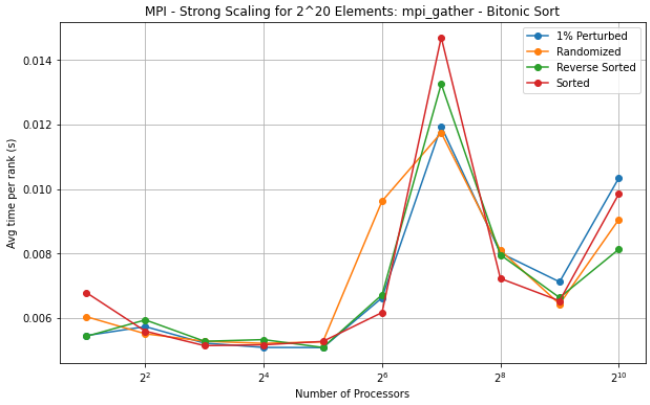


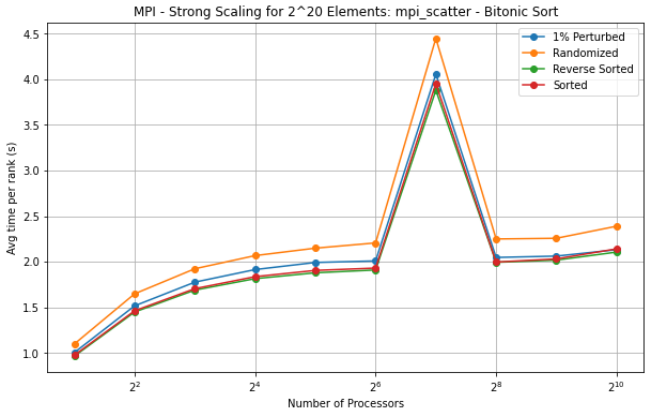
2^20 Elements

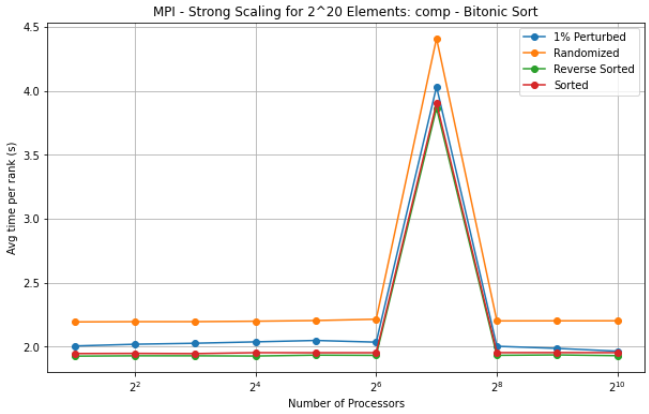

MPI - Strong Scaling for 2^20 Elements: main - Bitonic Sort
Avg time per rank (s)
Number of Processors
1% Perturbed
Randomized
Reverse Sorted
Sorted

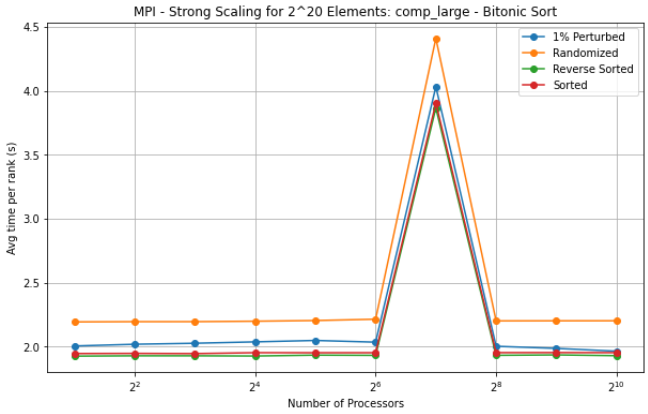

MPI - Strong Scaling for 2^20 Elements: comm - Bitonic Sort
Avg time per rank (s)
Number of Processors
1% Perturbed
Randomized
Reverse Sorted
Sorted

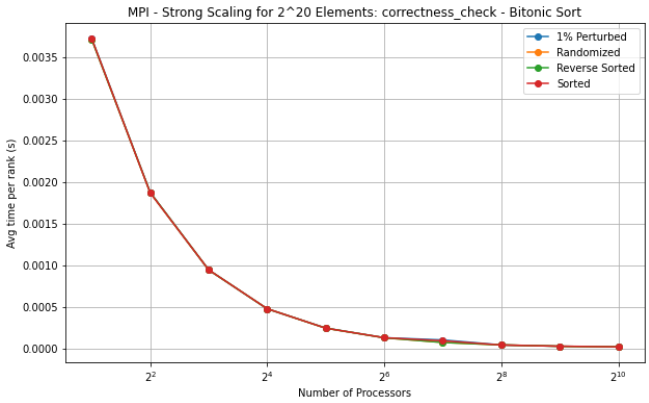

MPI - Strong Scaling for 2^20 Elements: comm_large - Bitonic Sort
Avg time per rank (s)
Number of Processors
1% Perturbed
Randomized
Reverse Sorted
Sorted

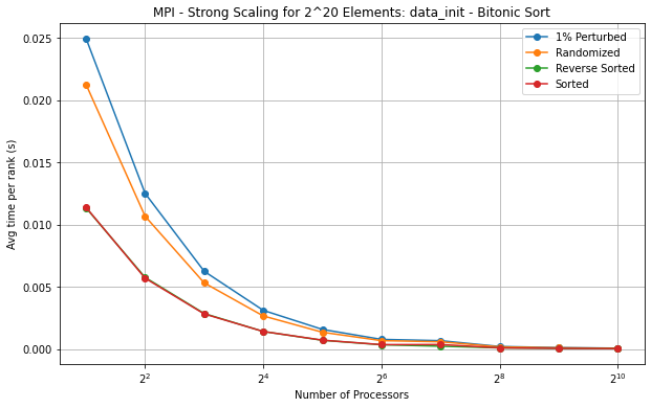

MPI - Strong Scaling for 2^20 Elements: mpi_gather - Bitonic Sort
Avg time per rank (s)
Number of Processors
1% Perturbed
Randomized
Reverse Sorted
Sorted


MPI - Strong Scaling for 2^20 Elements: mpi_scatter - Bitonic Sort
Avg time per rank (s)
Number of Processors
1% Perturbed
Randomized
Reverse Sorted
Sorted

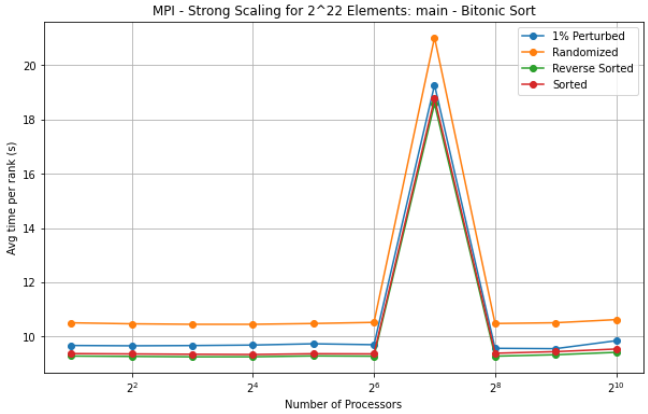

MPI - Strong Scaling for 2^20 Elements: comp - Bitonic Sort
Avg time per rank (s)
Number of Processors
1% Perturbed
Randomized
Reverse Sorted
Sorted

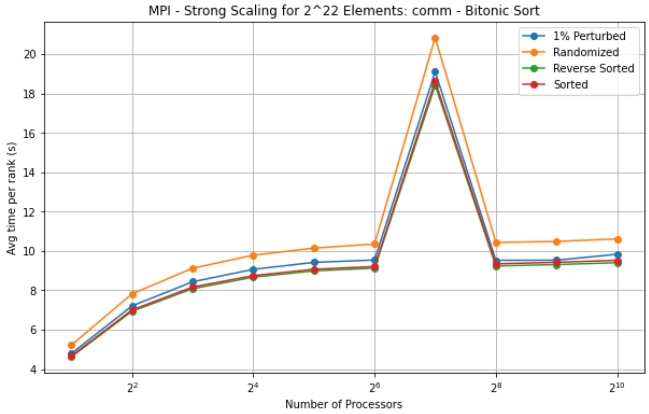


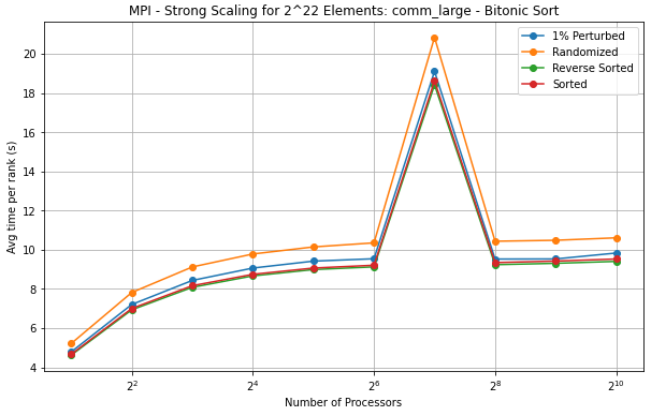


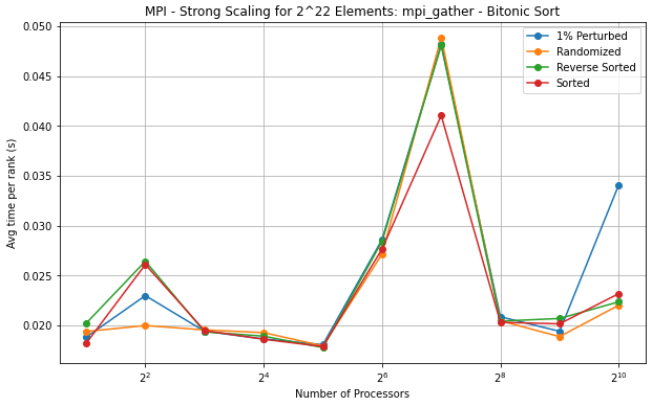


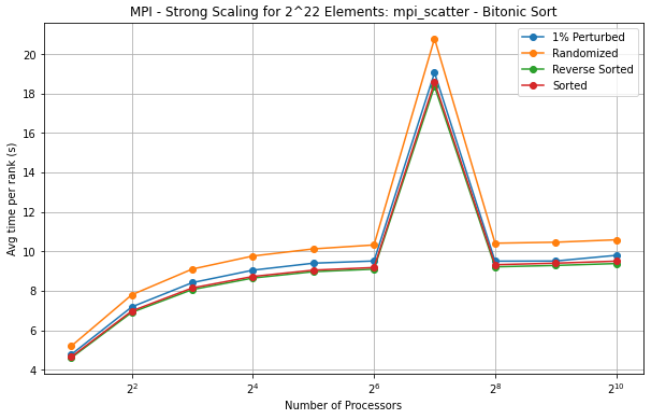
2^22 Elements

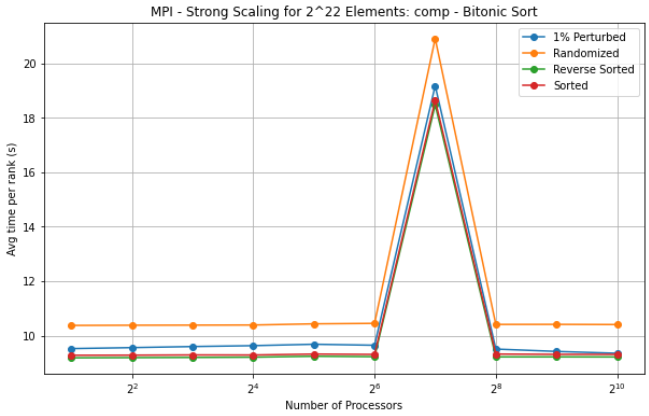


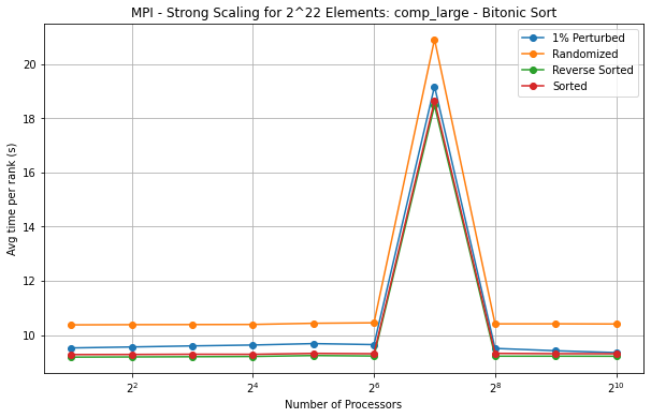


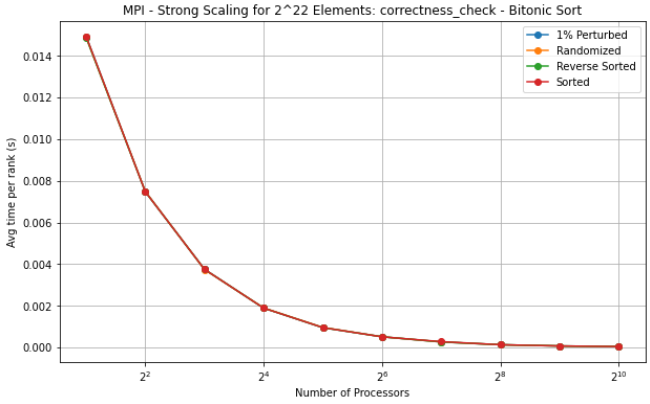


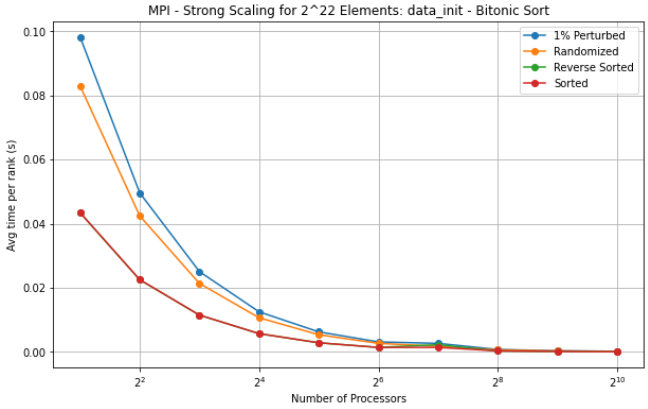




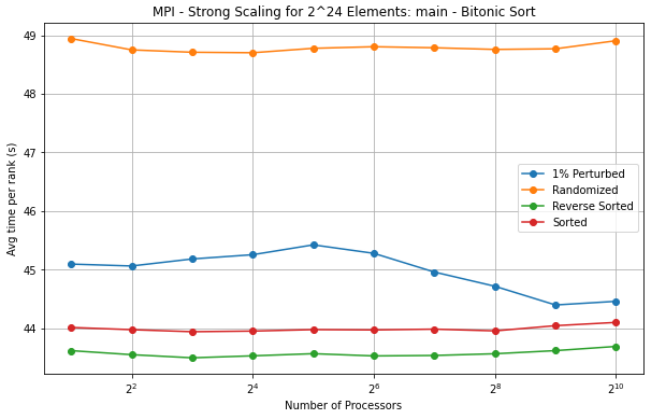


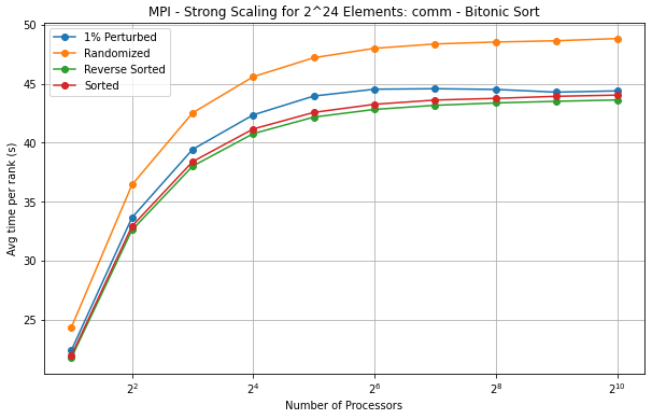


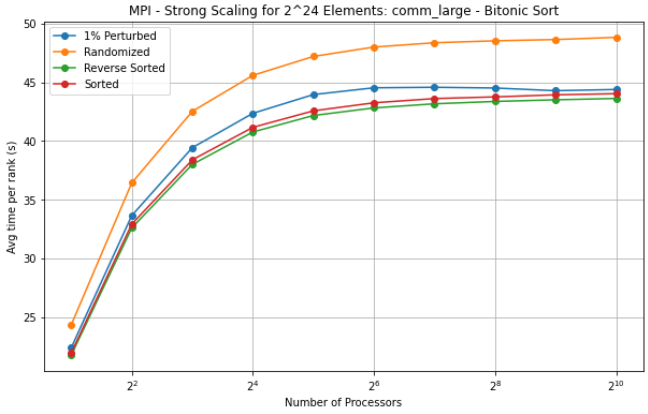


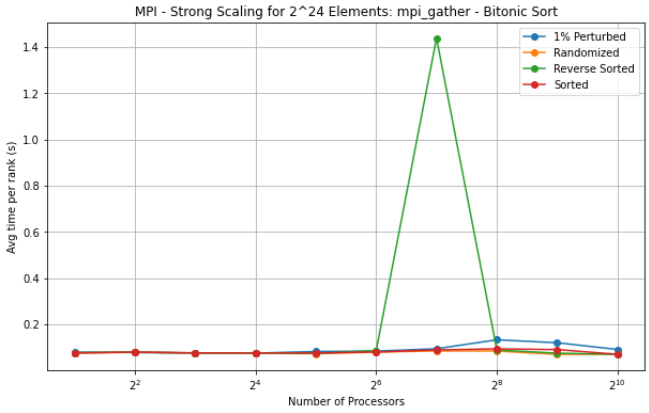


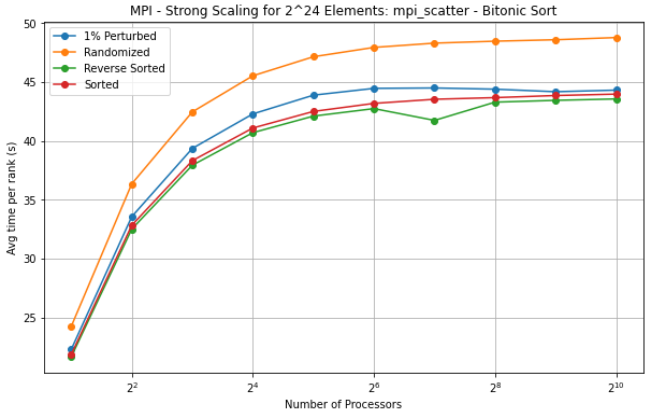
2^24 Elements

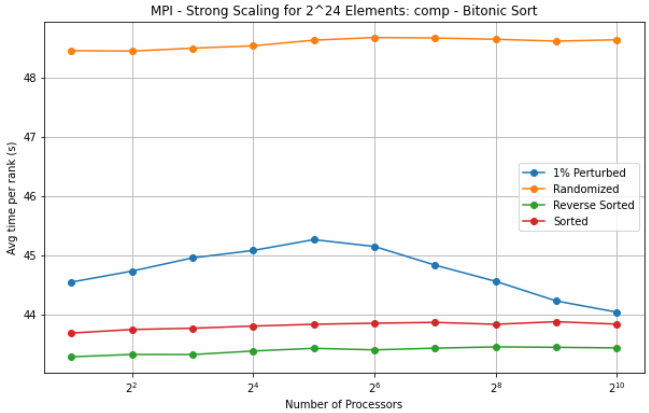

MPI - Strong Scaling for 2^24 Elements: main - Bitonic Sort
Avg time per rank (s)
Number of Processors
1% Perturbed
Randomized
Reverse Sorted
Sorted

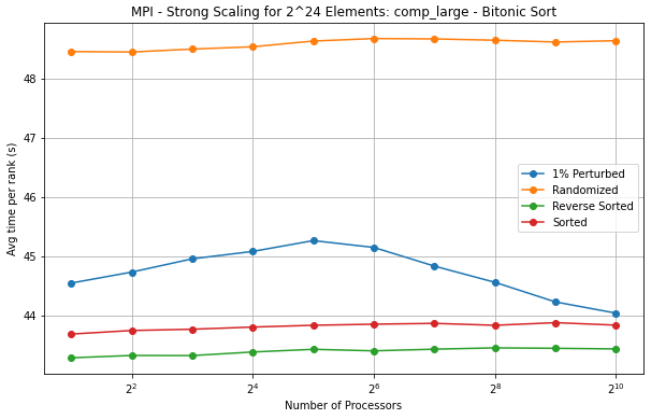

MPI - Strong Scaling for 2^24 Elements: comm - Bitonic Sort
Avg time per rank (s)
Number of Processors
1% Perturbed
Randomized
Reverse Sorted
Sorted

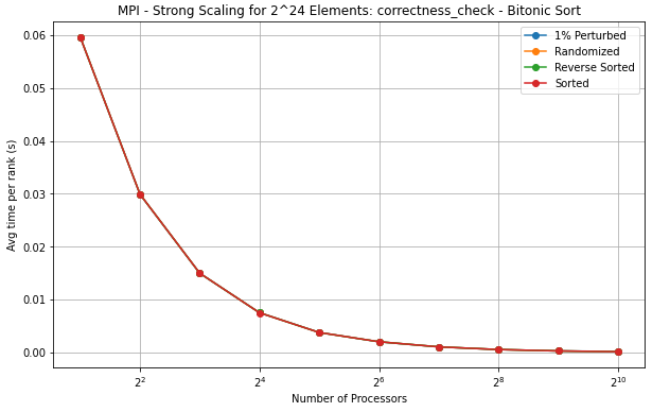

MPI - Strong Scaling for 2^24 Elements: comm_large - Bitonic Sort
Avg time per rank (s)
Number of Processors
1% Perturbed
Randomized
Reverse Sorted
Sorted

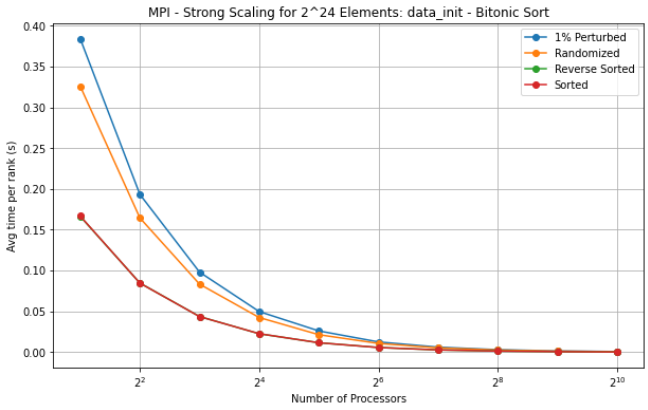

MPI - Strong Scaling for 2^24 Elements: mpi_gather - Bitonic Sort
Avg time per rank (s)
Number of Processors
1% Perturbed
Randomized
Reverse Sorted
Sorted


MPI - Strong Scaling for 2^24 Elements: mpi_scatter - Bitonic Sort
Avg time per rank (s)
Number of Processors
1% Perturbed
Randomized
Reverse Sorted
Sorted

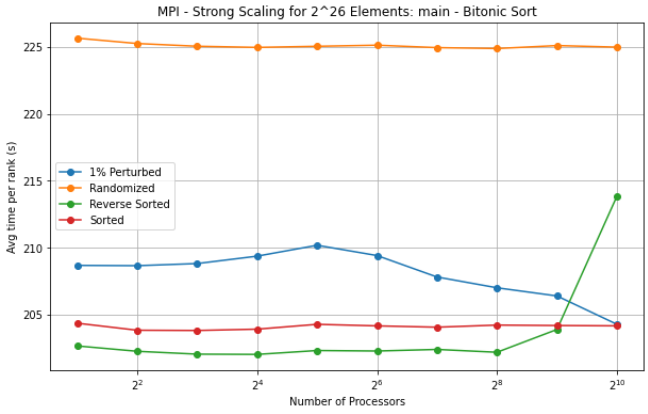

MPI - Strong Scaling for 2^24 Elements: comp - Bitonic Sort
Avg time per rank (s)
Number of Processors
1% Perturbed
Randomized
Reverse Sorted
Sorted

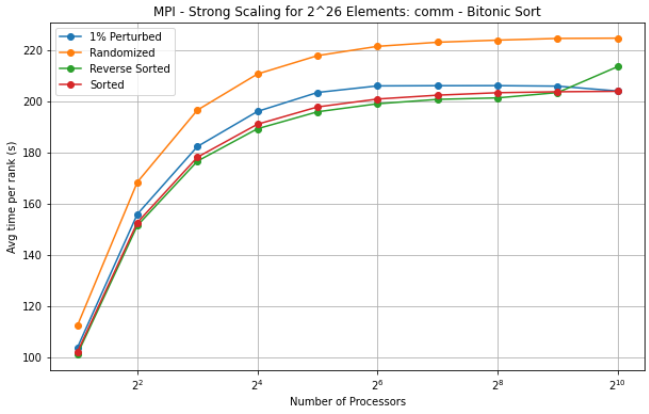


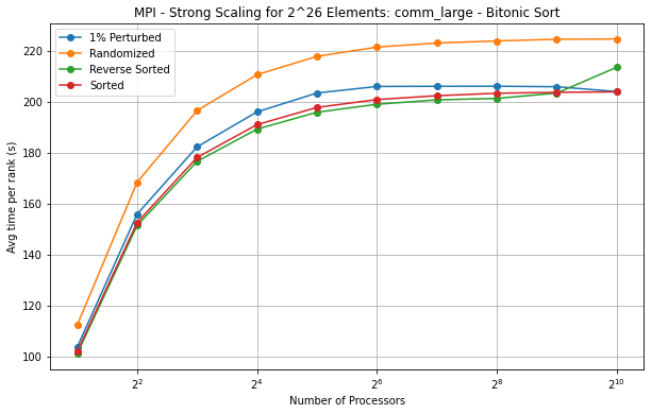


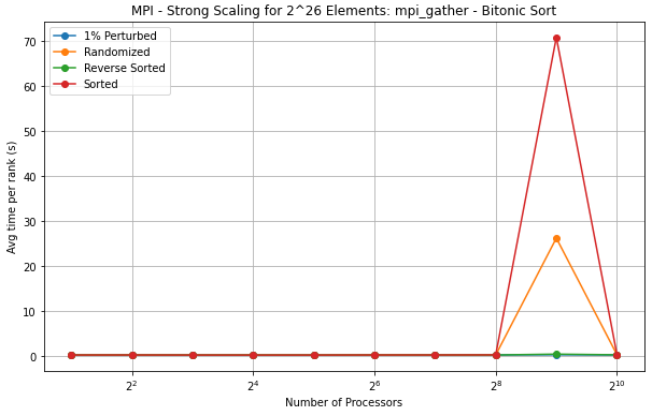


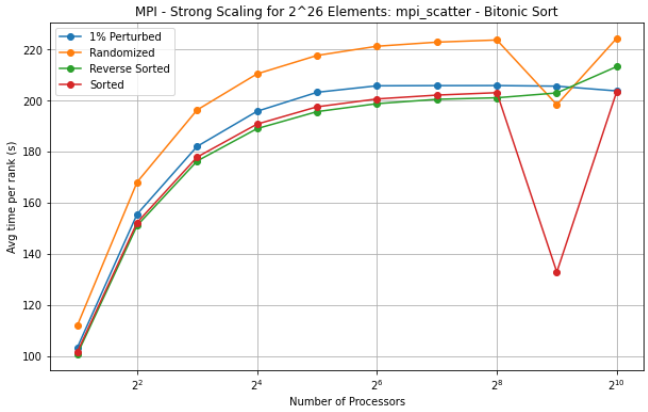
2^26 Elements

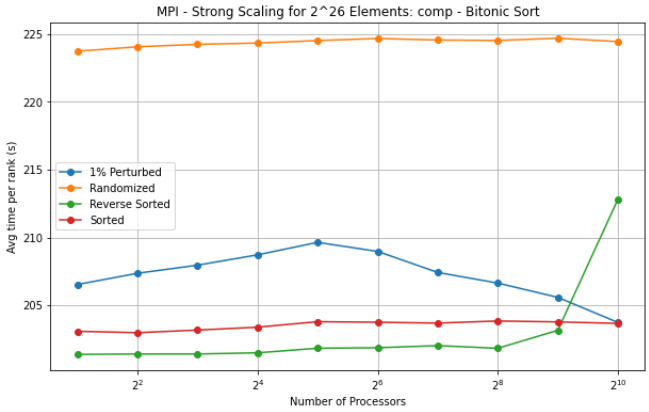


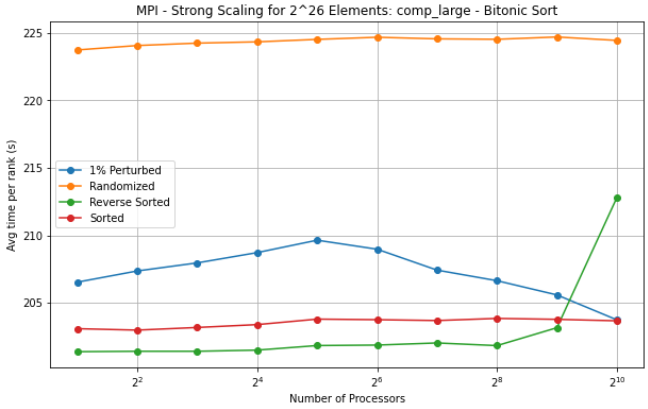


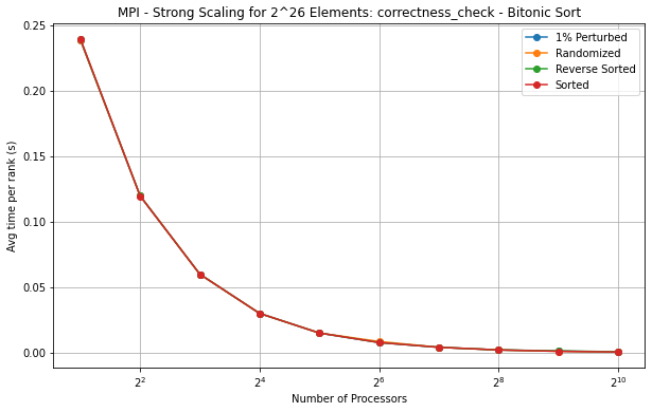


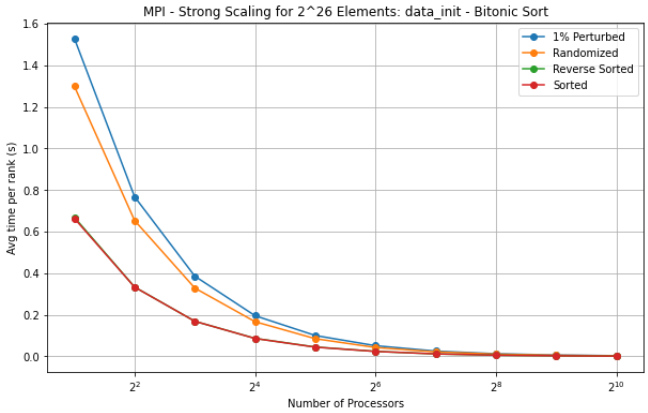




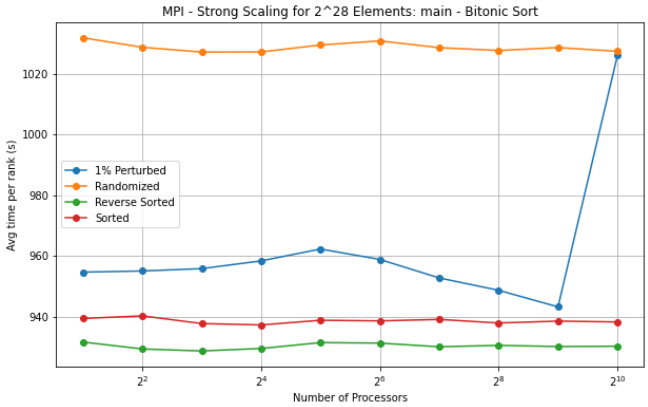


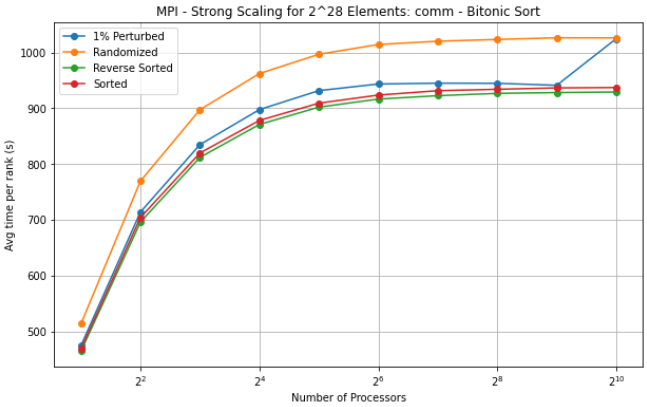


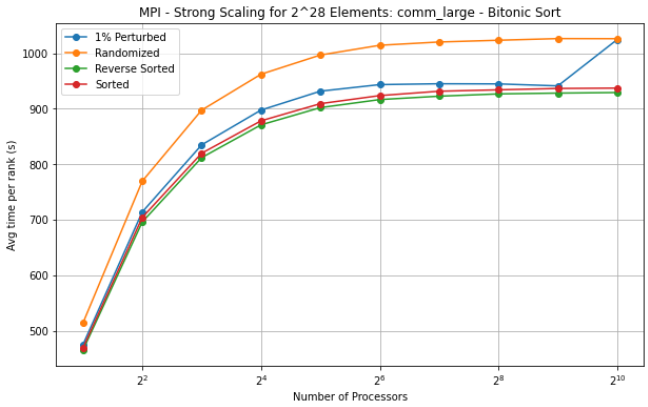


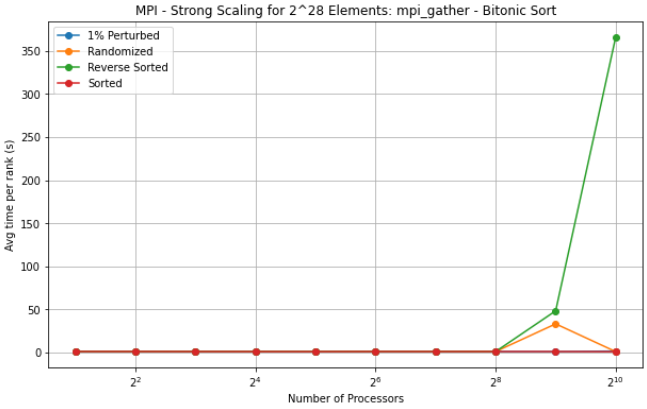


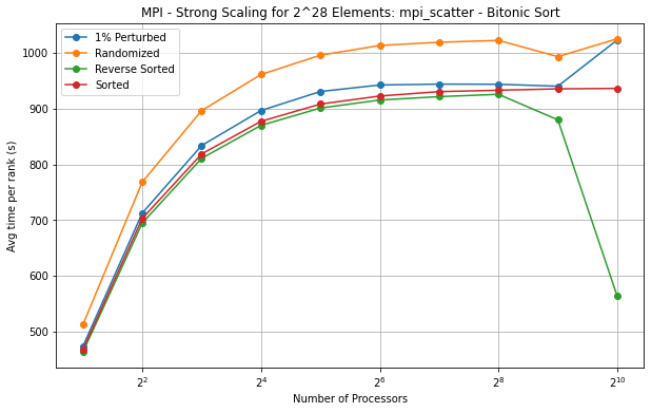
2^28 Elements

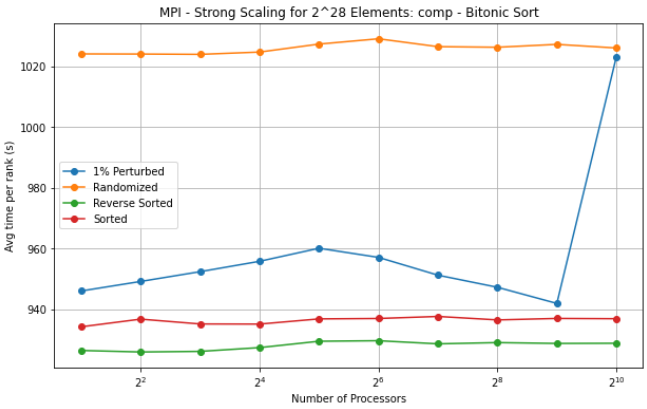

MPI - Strong Scaling for 2^28 Elements: main - Bitonic Sort
Avg time per rank (s)
Number of Processors
1% Perturbed
Randomized
Reverse Sorted
Sorted

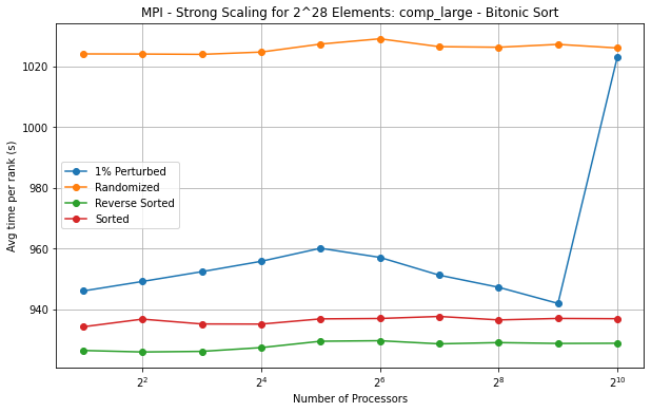

MPI - Strong Scaling for 2^28 Elements: comm - Bitonic Sort
Avg time per rank (s)
Number of Processors
1% Perturbed
Randomized
Reverse Sorted
Sorted

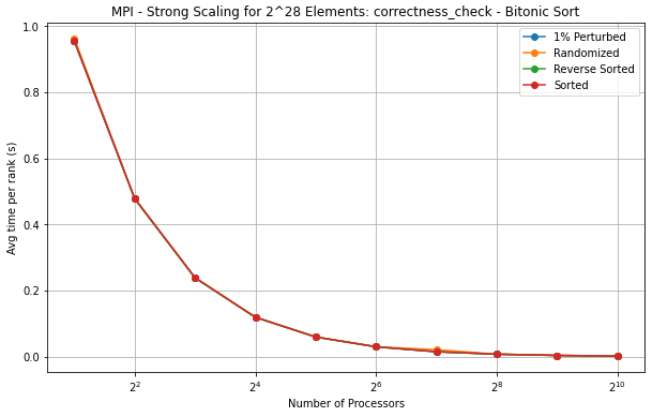

MPI - Strong Scaling for 2^28 Elements: comm_large - Bitonic Sort
Avg time per rank (s)
Number of Processors
1% Perturbed
Randomized
Reverse Sorted
Sorted

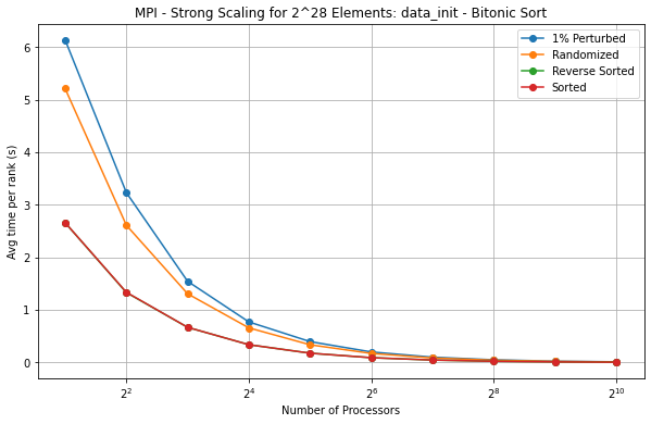

MPI - Strong Scaling for 2^28 Elements: mpi_gather - Bitonic Sort
Avg time per rank (s)
Number of Processors
1% Perturbed
Randomized
Reverse Sorted
Sorted


MPI - Strong Scaling for 2^28 Elements: mpi_scatter - Bitonic Sort
Avg time per rank (s)
Number of Processors
1% Perturbed
Randomized
Reverse Sorted
Sorted


MPI - Strong Scaling for 2^28 Elements: comp - Bitonic Sort
Avg time per rank (s)
Number of Processors
1% Perturbed
Randomized
Reverse Sorted
Sorted



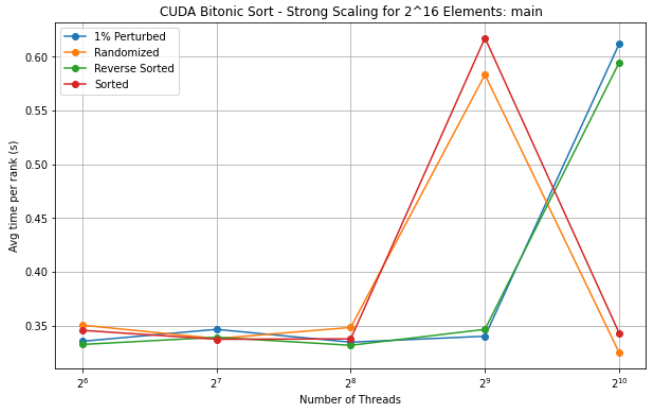


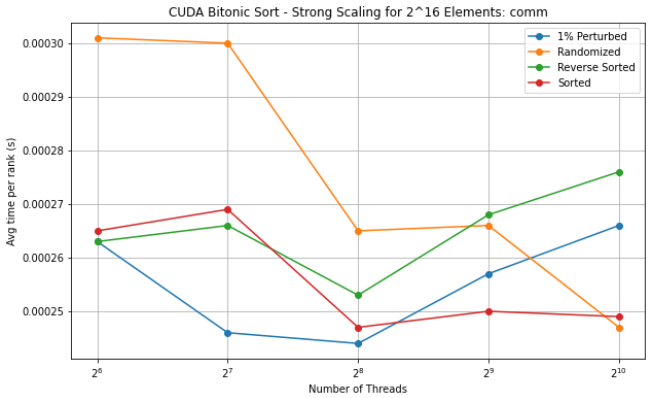


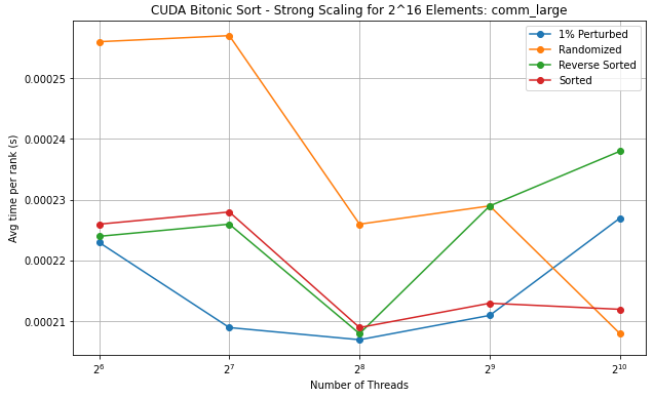
CUDA

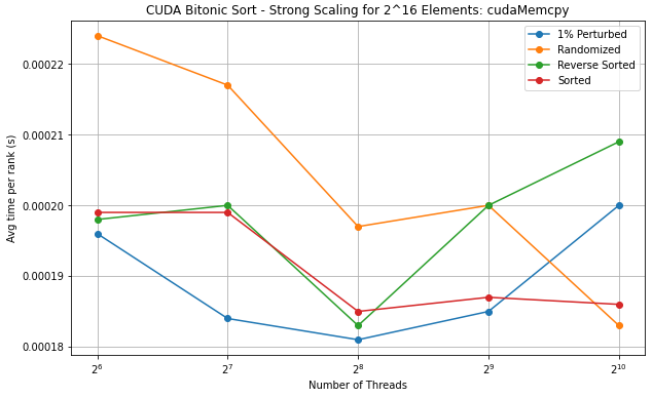
For this analysis, we are looking at CUDA strong scaling across different input sizes. The trends are not as general for these graphs as there is a lot of variance throughout. This may mean that CUDA and the parallelization is not as effective for this sort. It is also very interesting to note that even for the graphs for the main functions in all input sizes, there is not a clear winner in time for any input type until 2^26 elements, while the expected winner would be a presorted input array. However, it is important to also note that for all graphs, the y-axis scale is smaller than that of MPI's, meaning that the variance is large but within a smaller scale. At 2^26 elements and 2^26 elements we see that randomized does perform the worst as expected and sorted performs the best as expected.

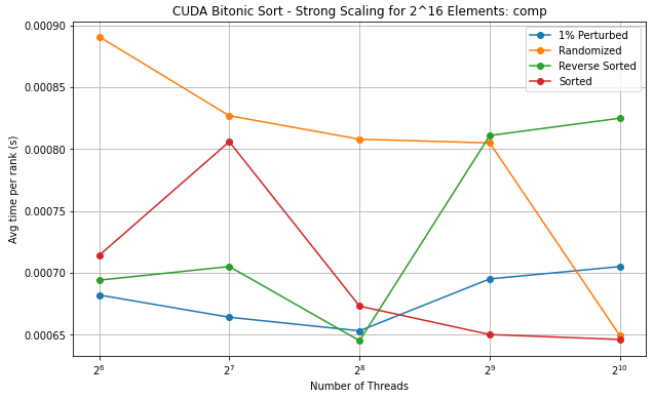
2^16 Elements

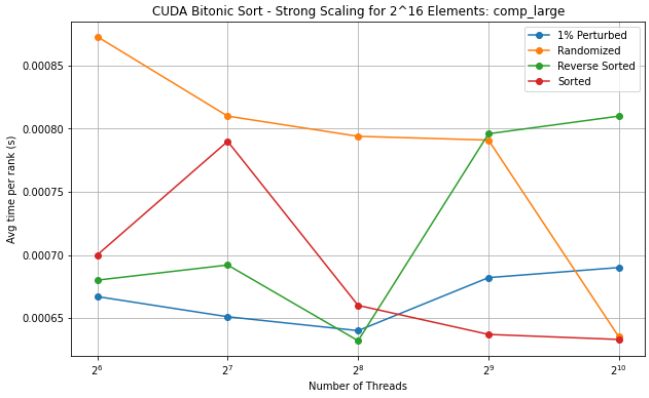


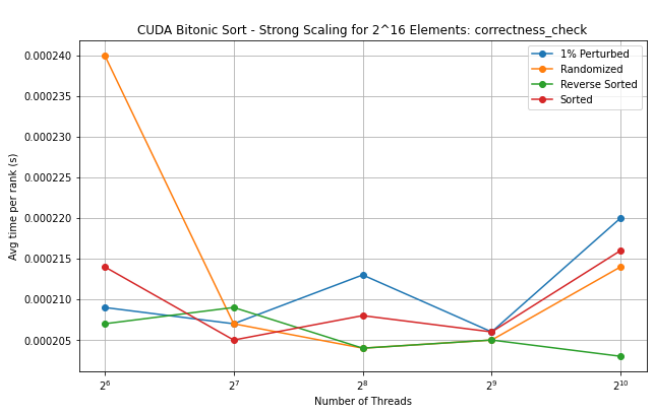


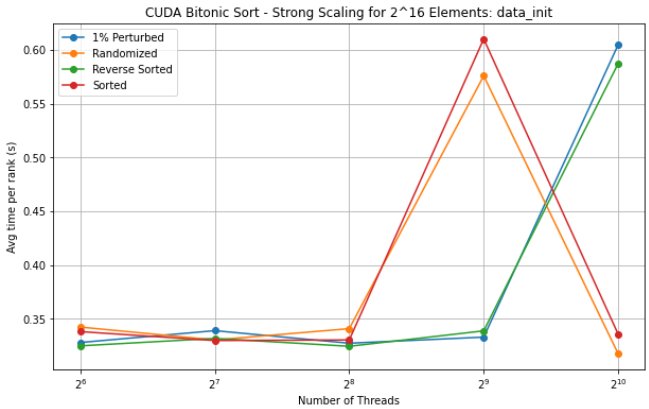




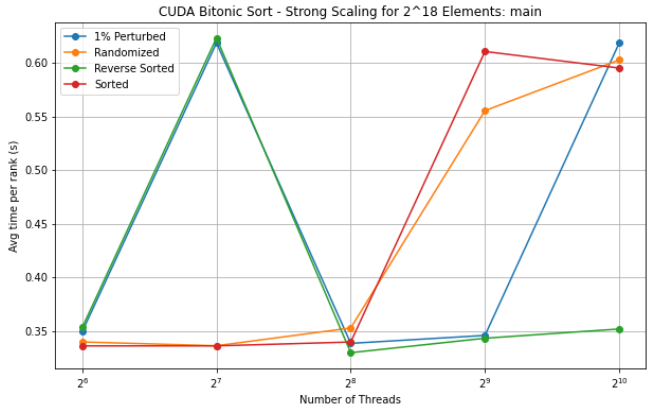


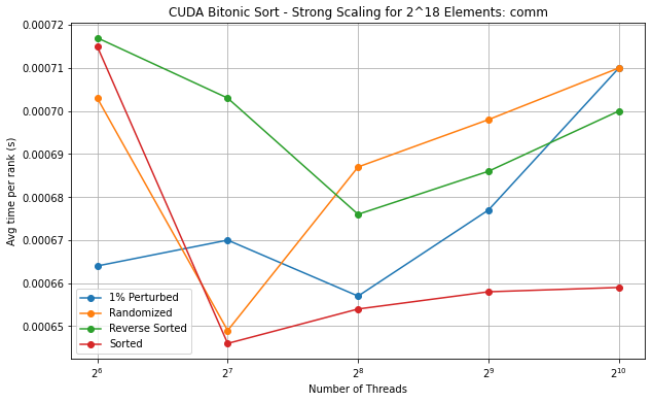


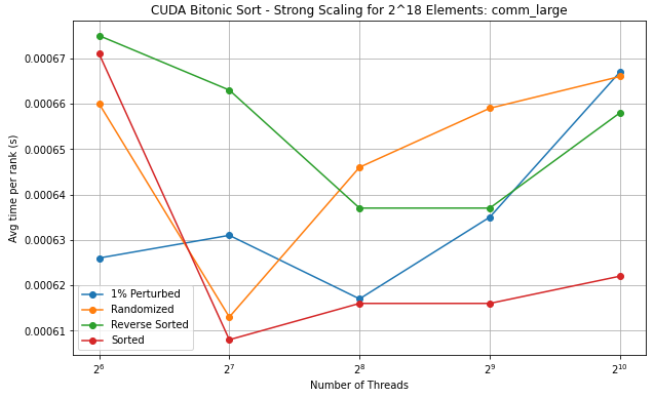


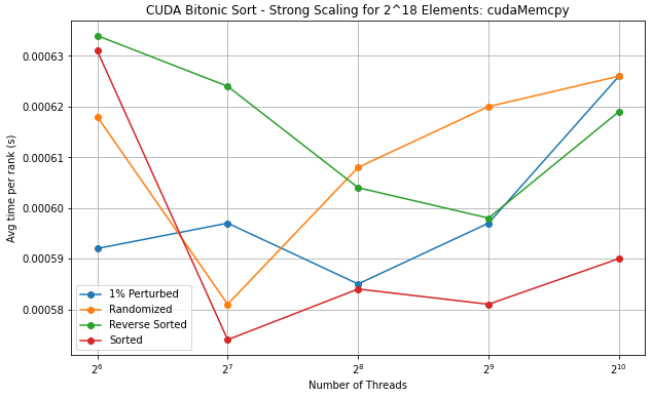


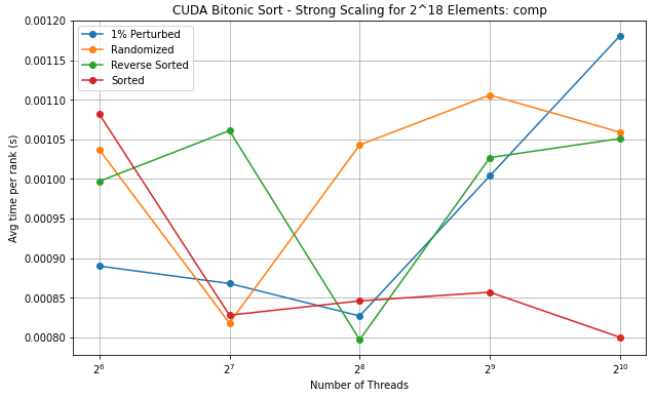
2 ^18 Elements

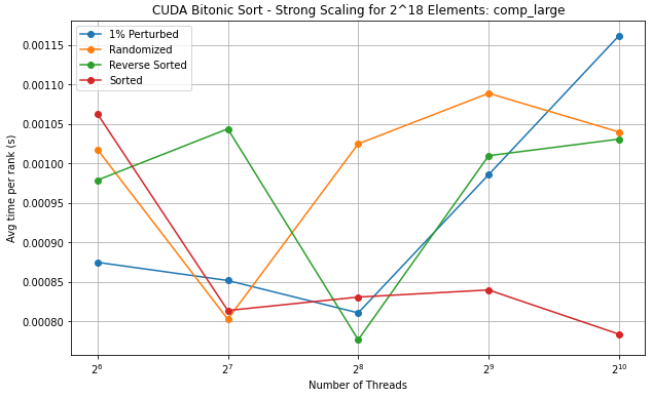


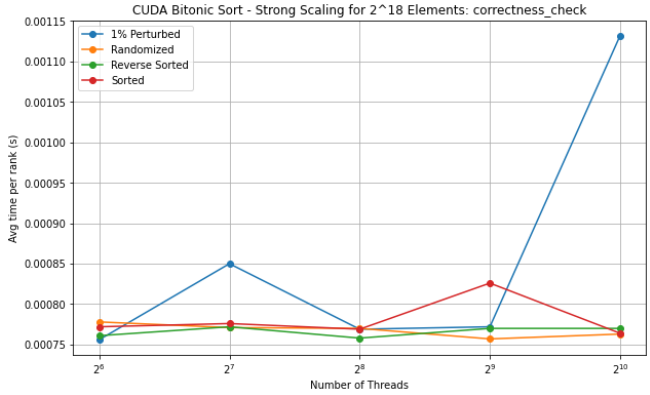


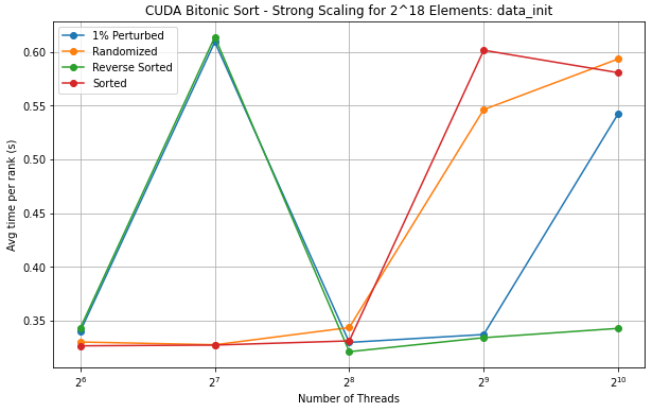




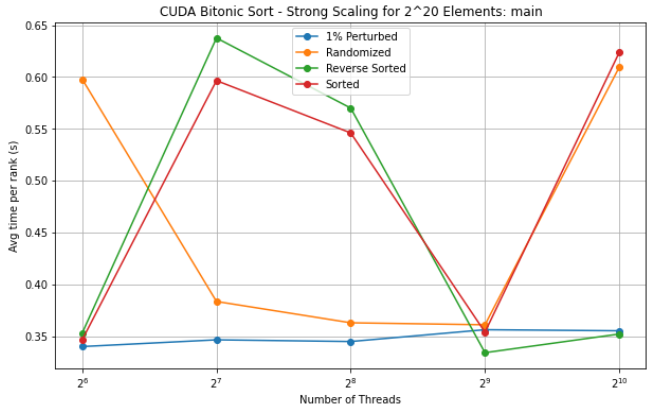


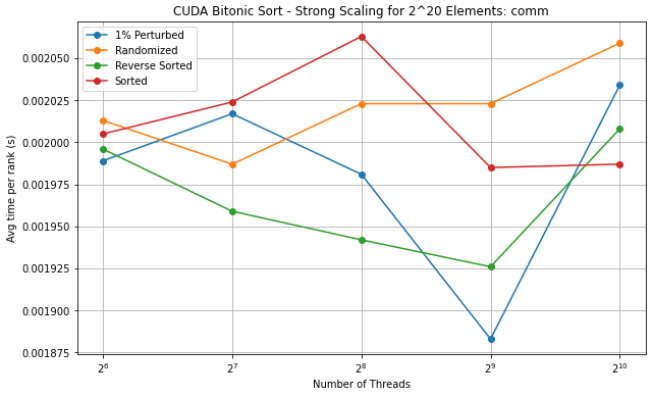


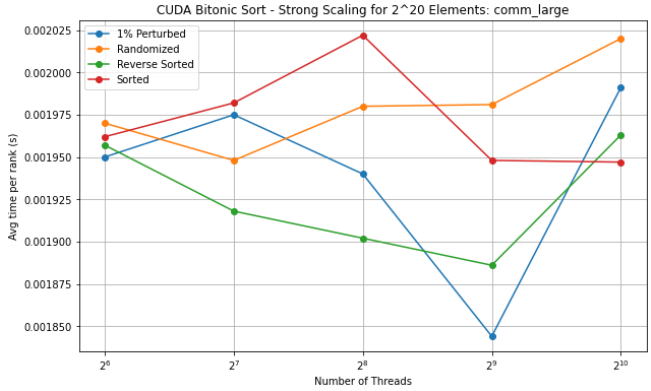


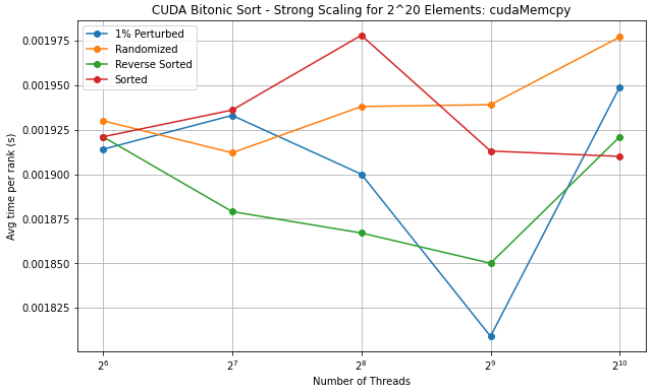


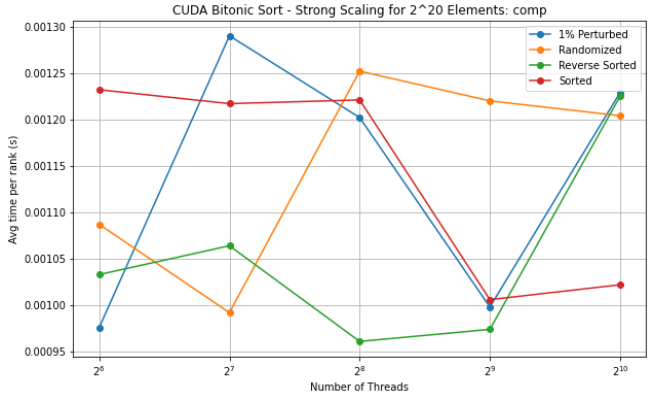
2^20 Elements

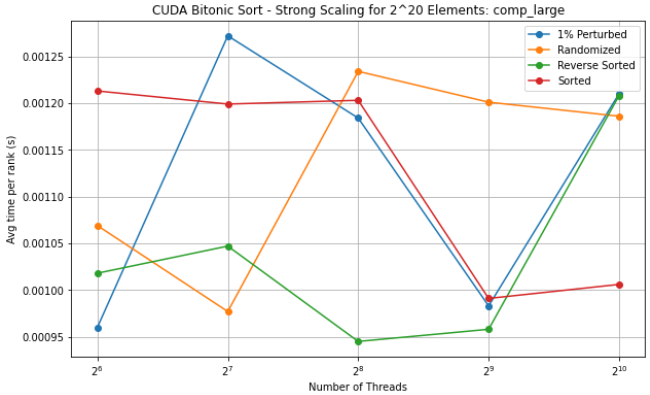


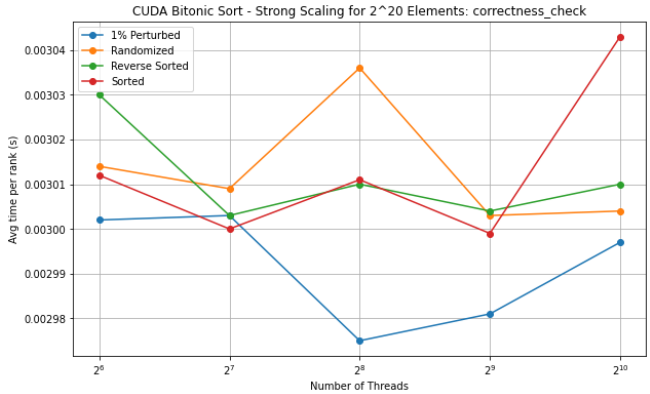


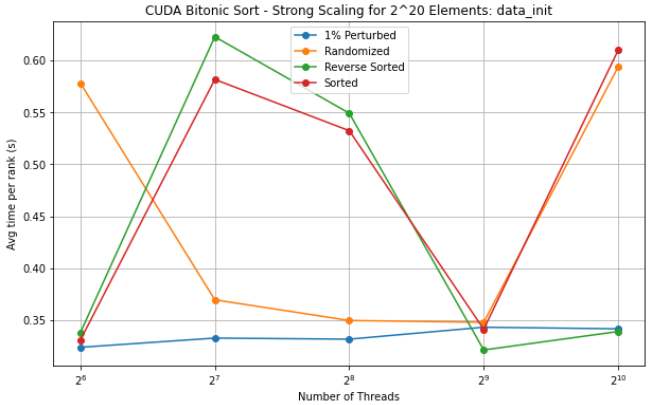




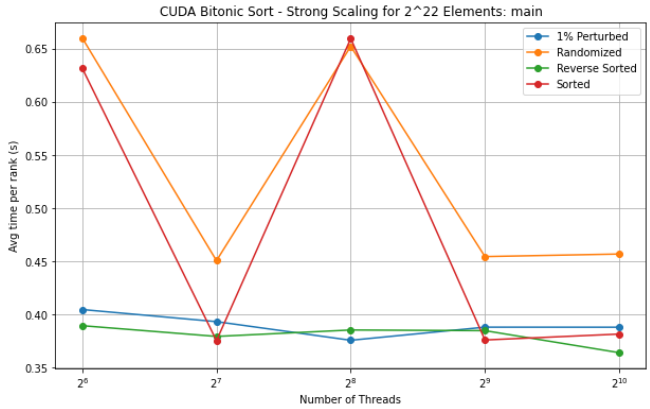


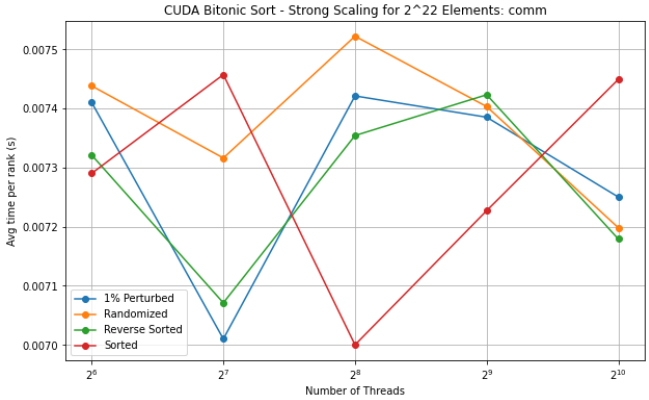


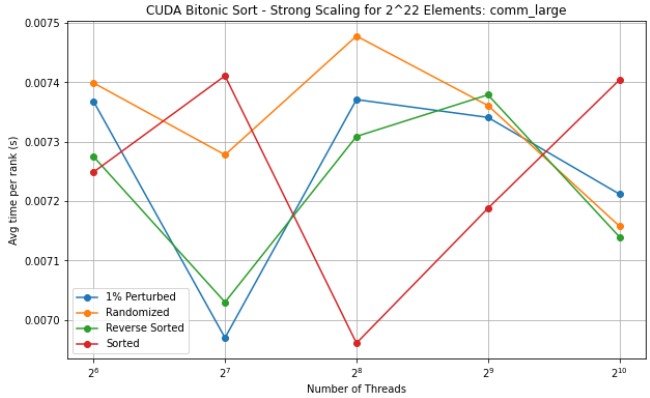


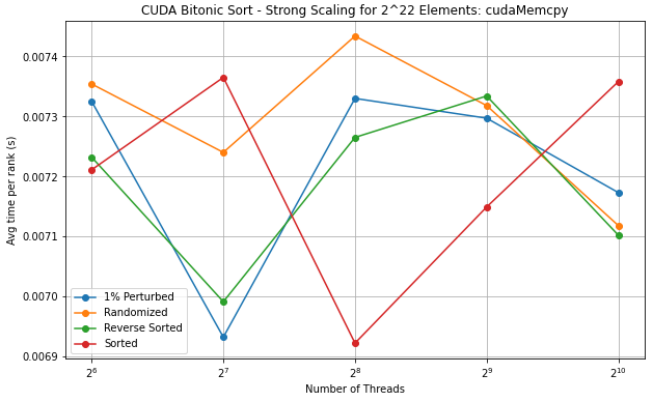


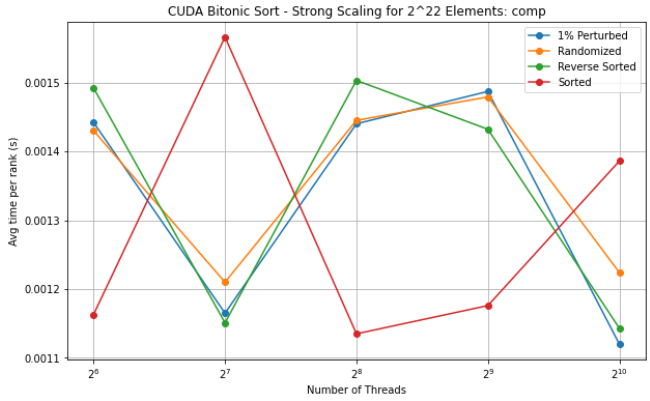
2^22 Elements

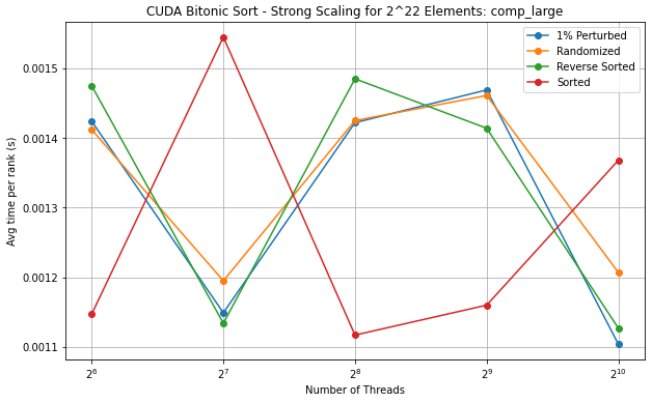


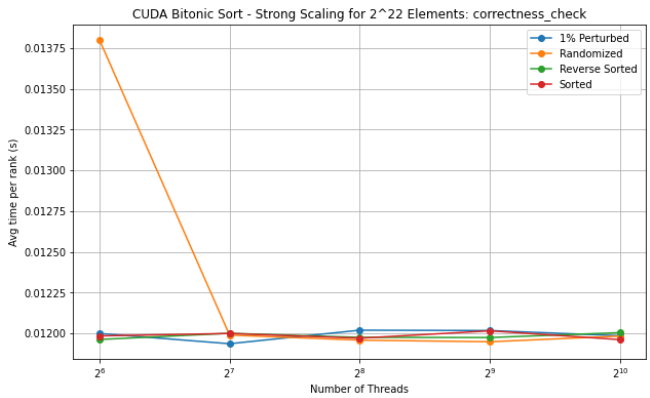


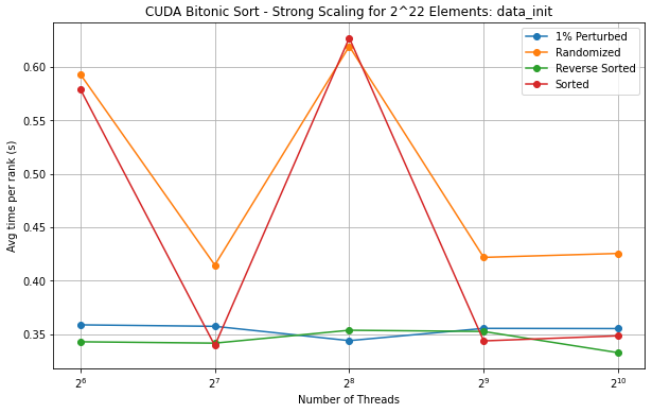




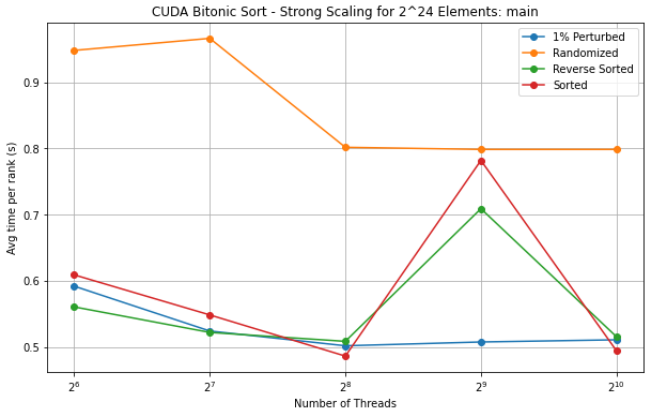


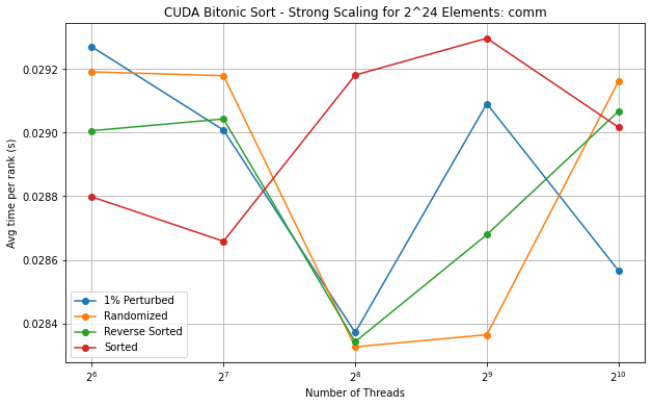


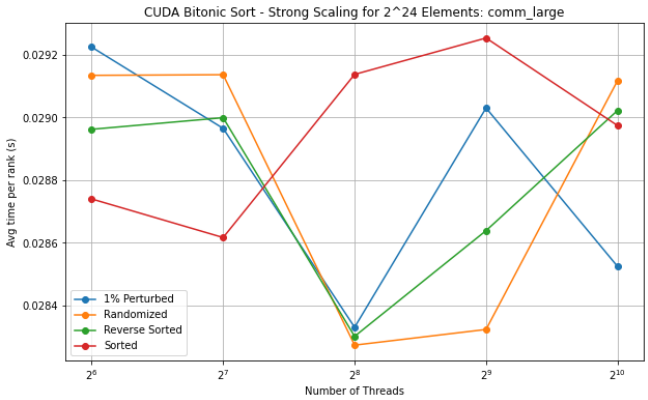


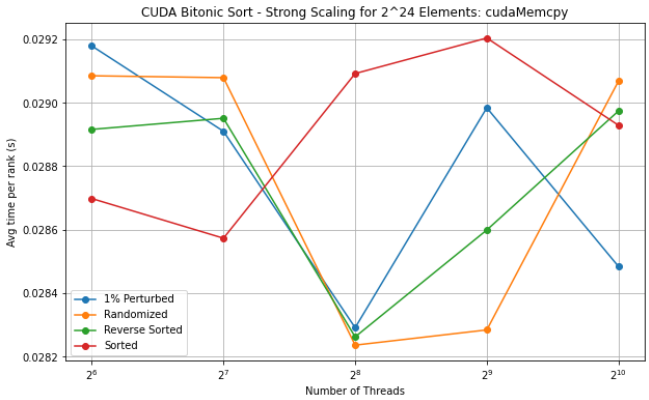


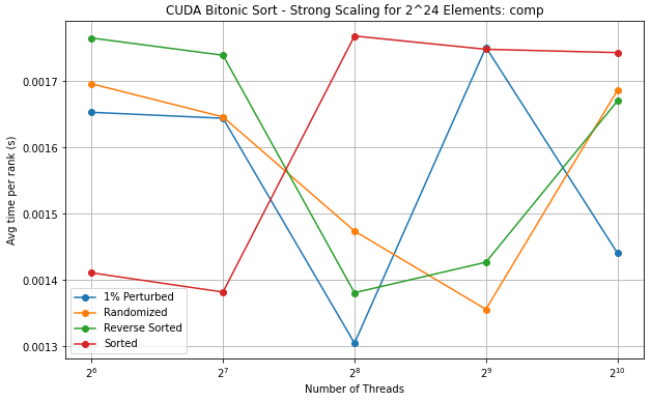
2^24 Elements

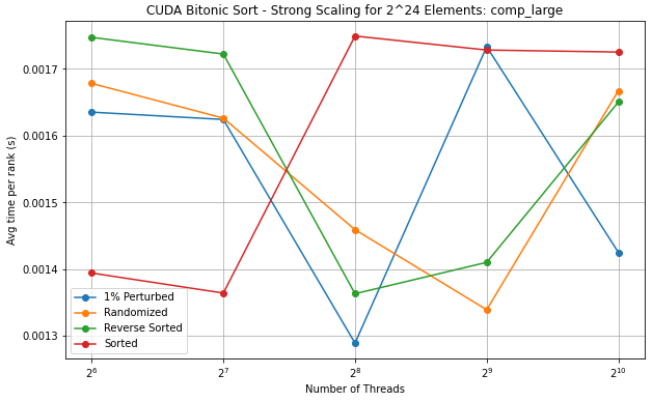


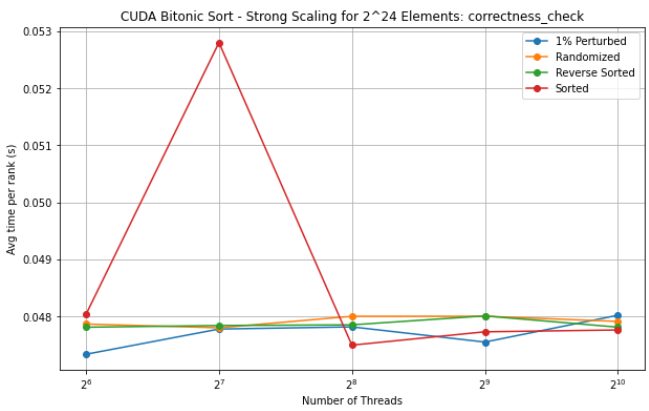


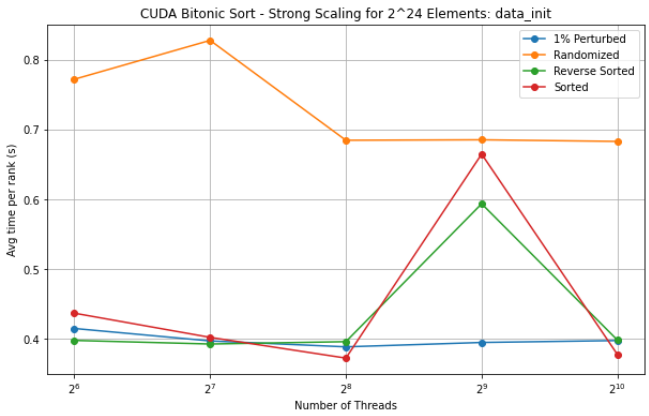




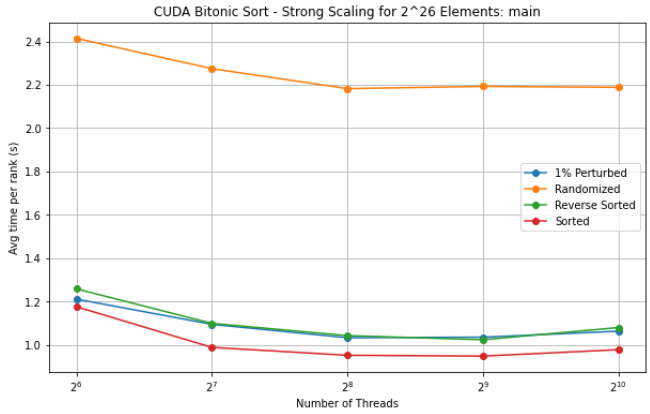


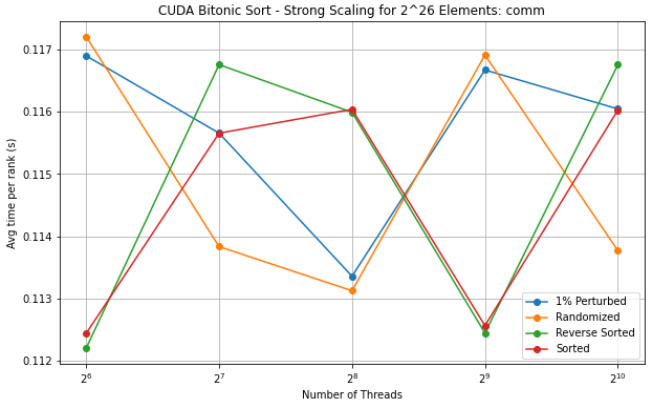


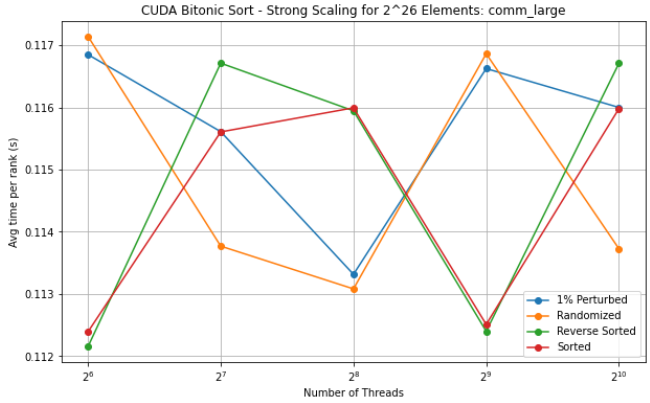


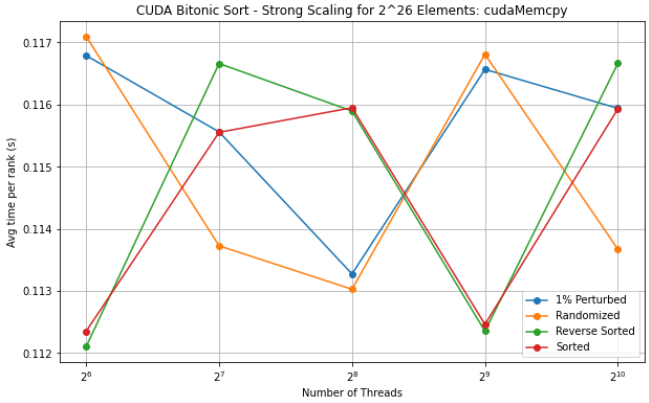


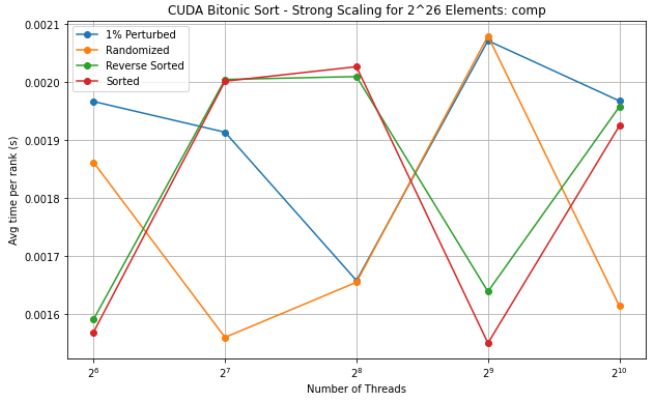
2^26 Elements

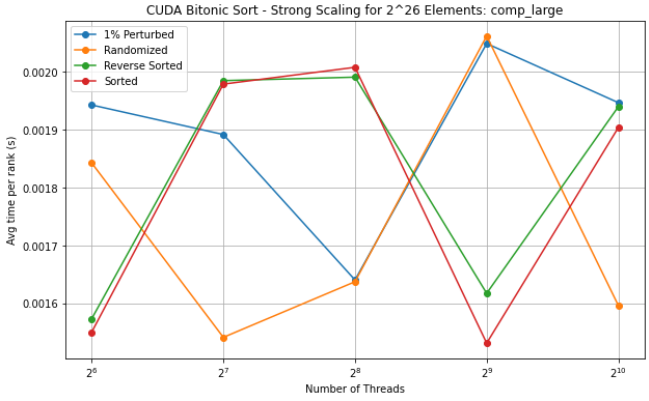


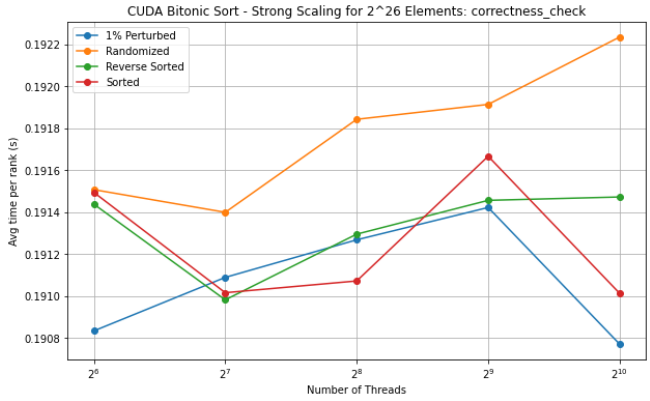


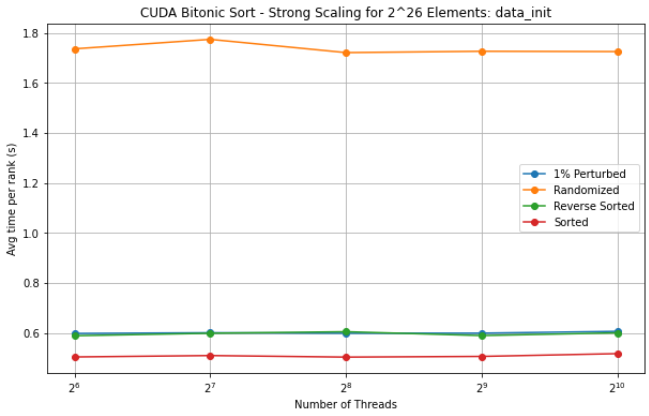




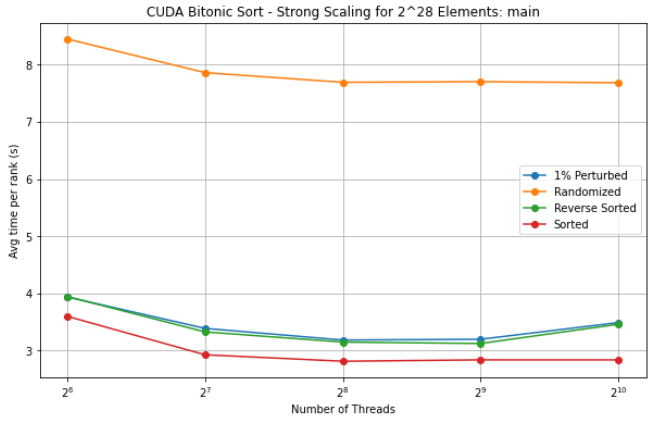


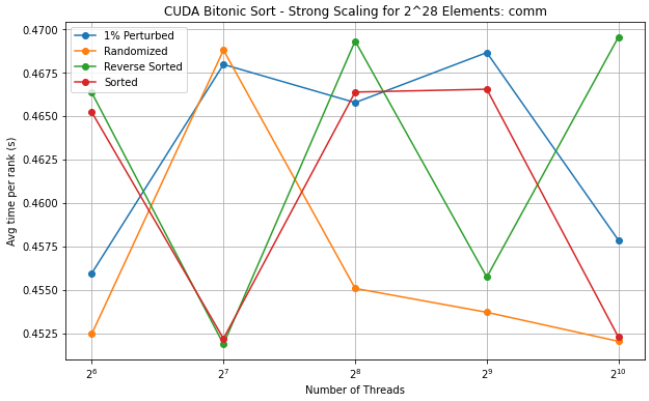


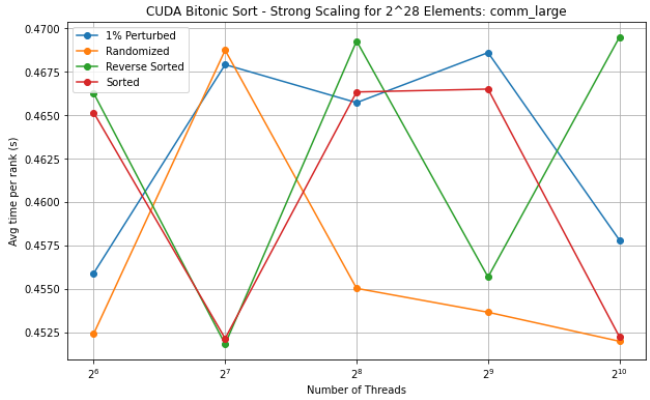


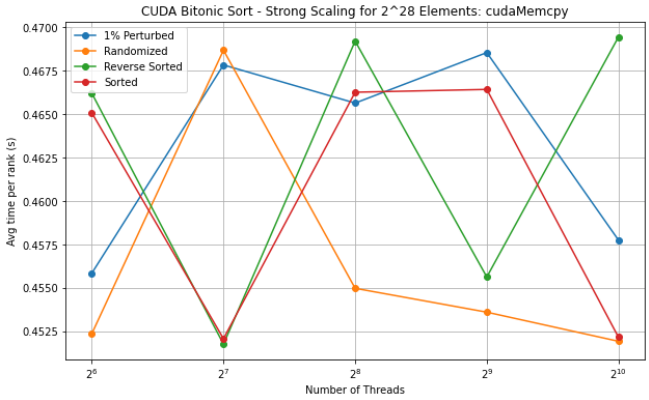


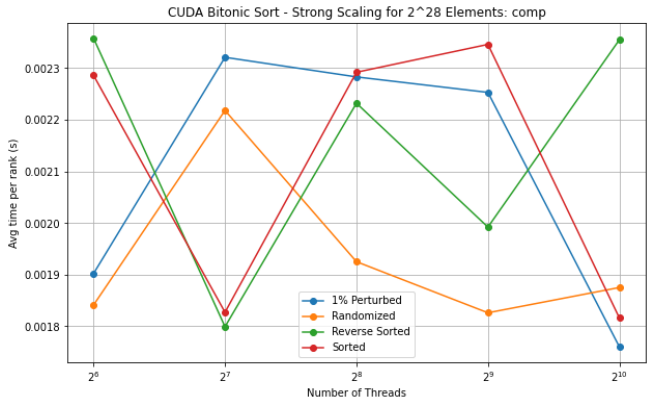
2^28 Elements

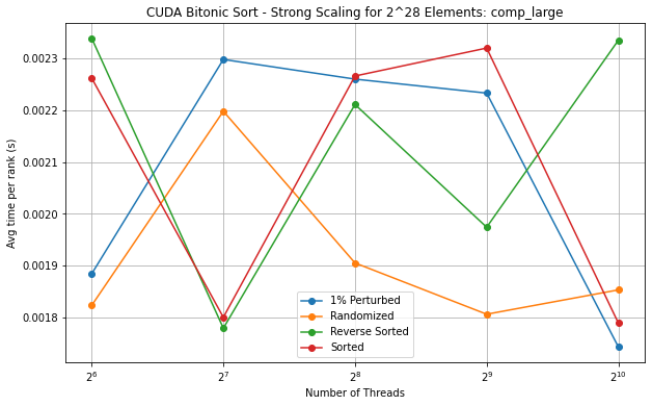


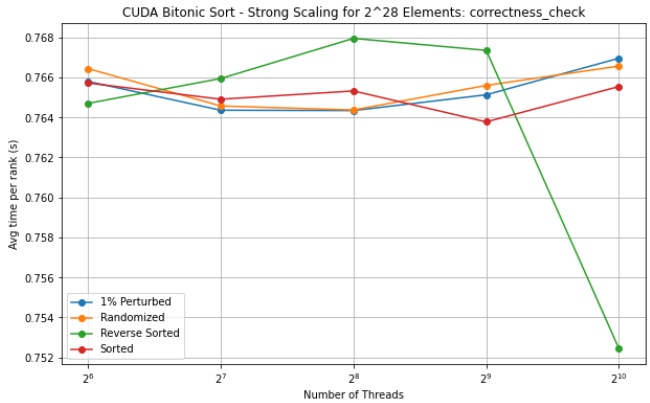


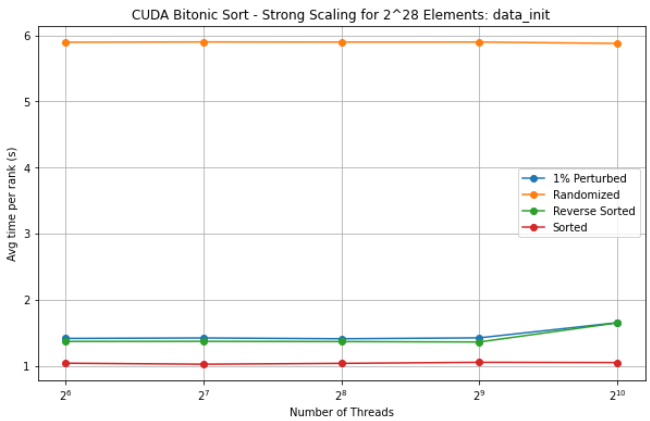












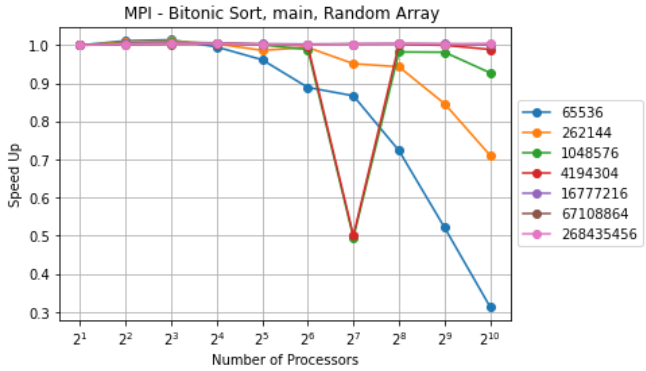
Speedup

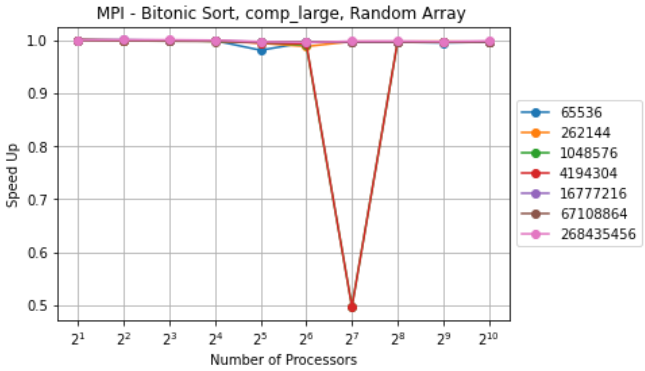
MPI

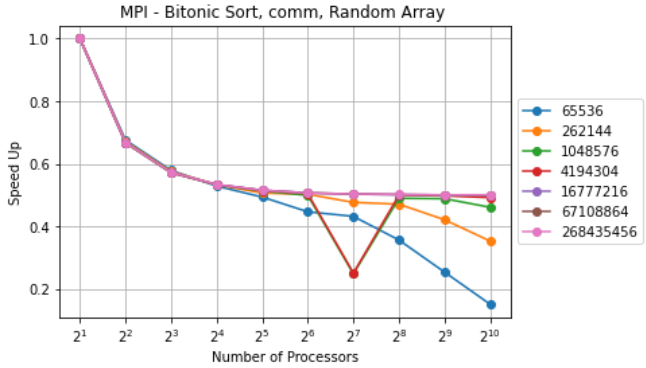
First we analyze speedup of MPI graphs across different input sizes. When looking at the computation speedups, we see a straight line for most graphs which is expected from out analysis of the MPI graphs earlier. Since the MPI graph times were generally stable throghout, there is an average speed up of 1 throughout. The

outlier that we saw earlier shows up again in the speedup at 2& processors having a speed up of 0.5, meaning that it gets slower there. The comm graphs also generally show a decreasing trend since communication generally takes longer with increasing input size. None of the graphs have a general increasing trend which would indicate the benefits and importance of parallelism.

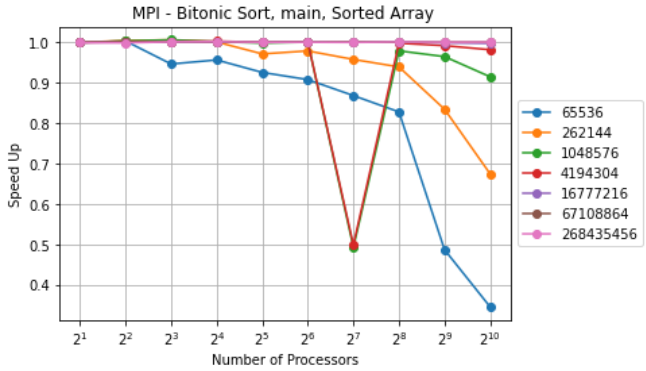
RANDOM INPUT ARRAY

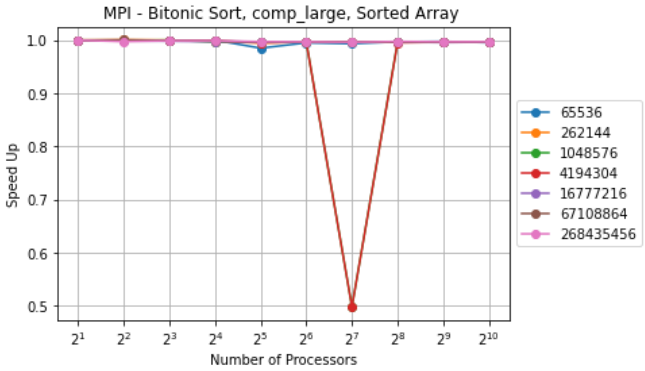


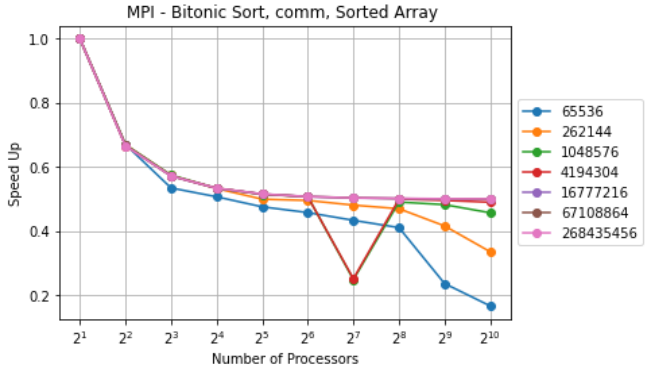




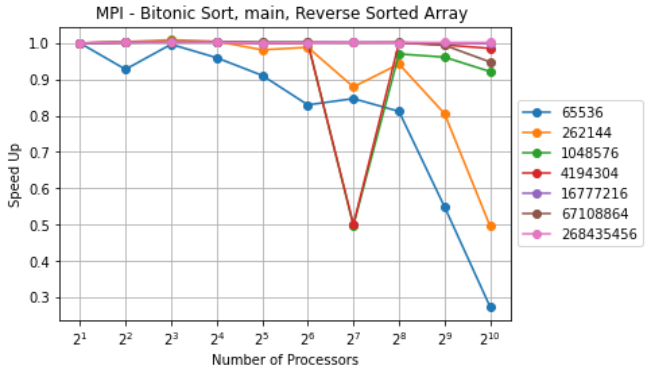
SORTED INPUT ARRAY

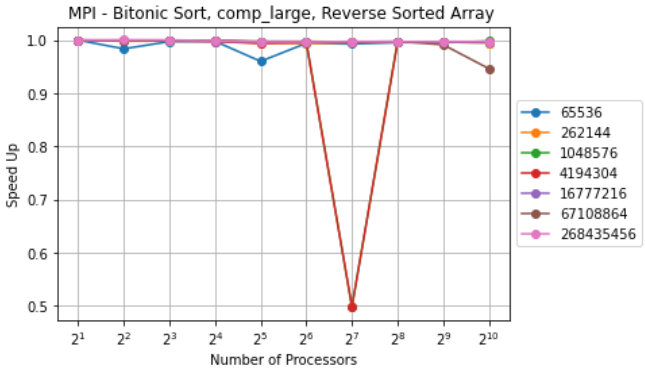


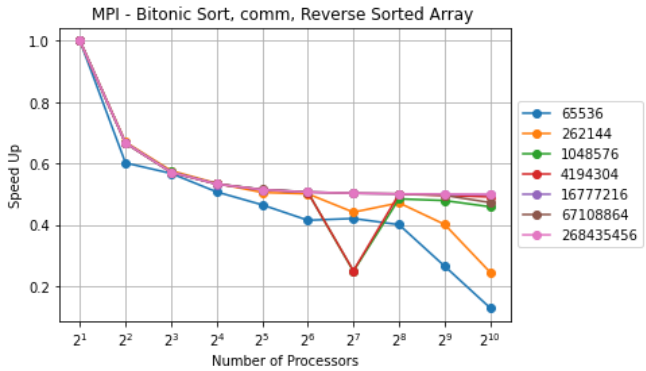




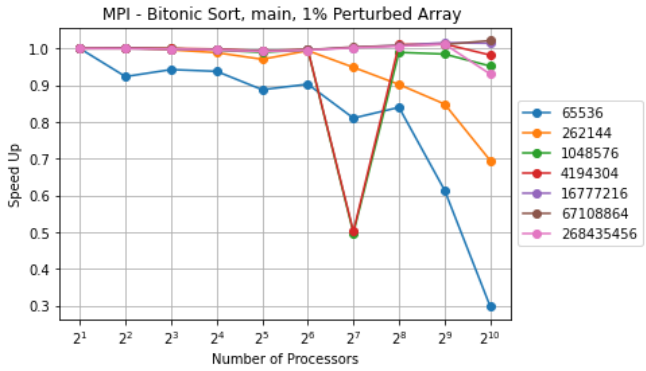
REVERSE SORTED INPUT ARRAY

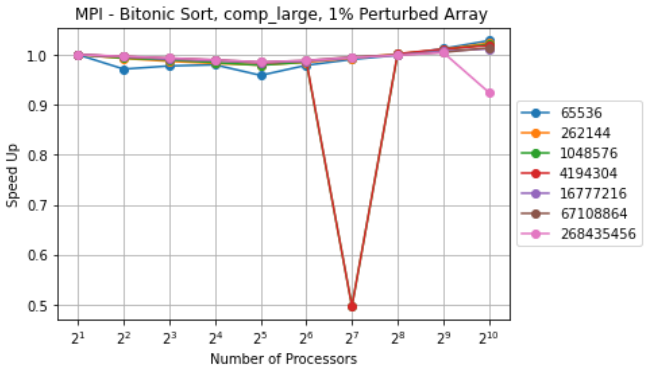


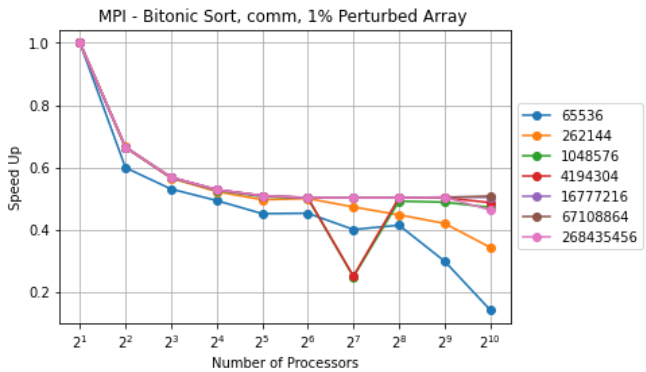




1% PERTURBED INPUT ARRAY



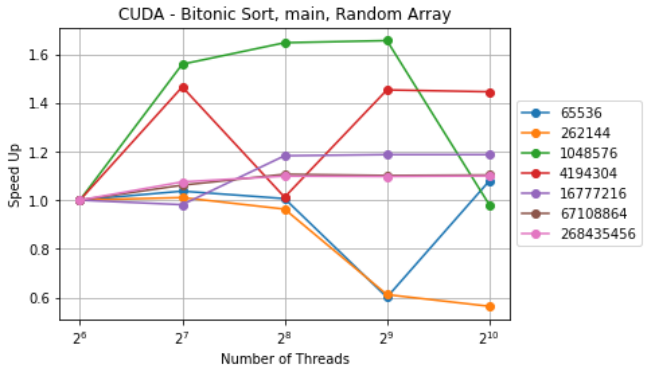


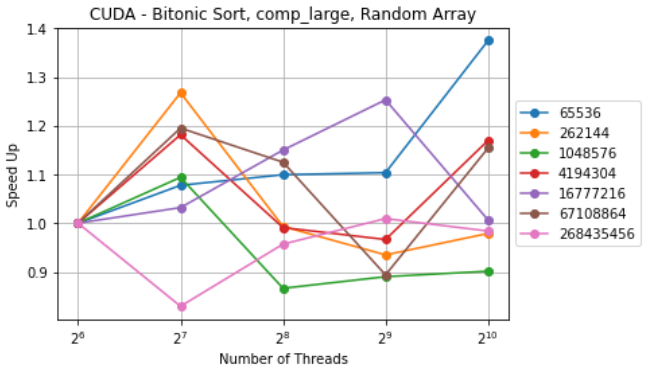


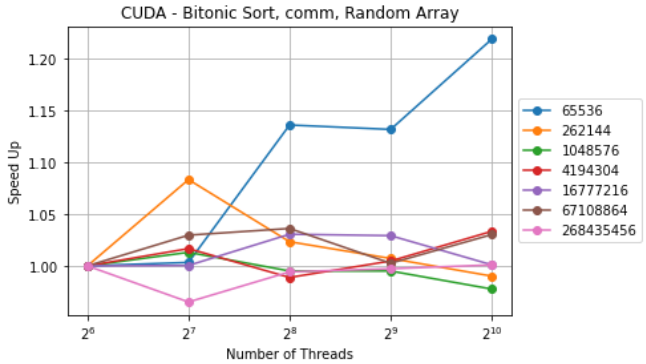
CUDA

Lastly, we analyze the speedup in CUDA graphs across input types. None of the CUDA speedup graphs show much benefit to parallelism. When we look at the main_function as a whole, the graphs vary up and down around 1.0 meaning that at times it speeds up whereas other times it slows down. This fits with our earlier analysis of CUDA graphs as we said they vary a lot more. However, for the largest input size (2^28) for the main_function, the line is entirely above 1.0 meaning that it does have parallelism benefit. The comm graphs tend to have the least variance across all input types likely due less communication for CUDA than MPI.

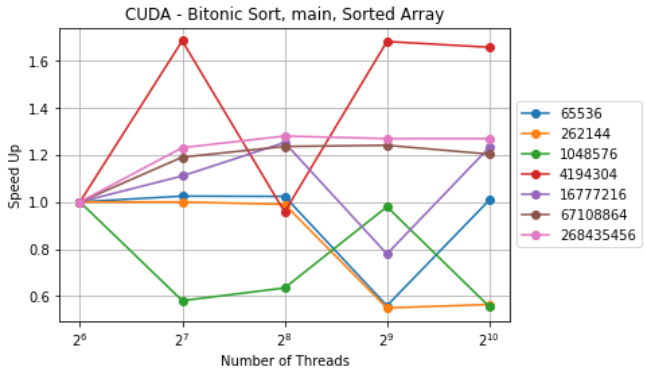
RANDOM INPUT ARRAY

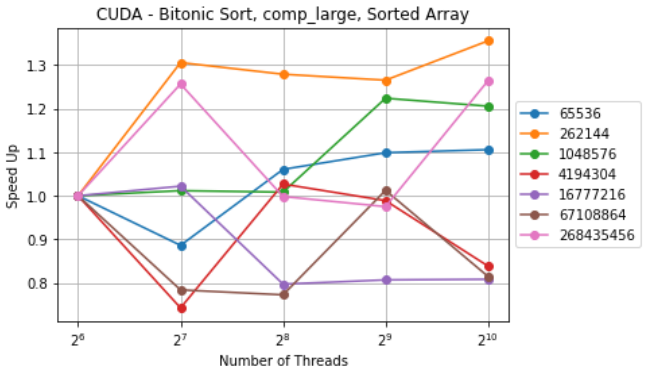


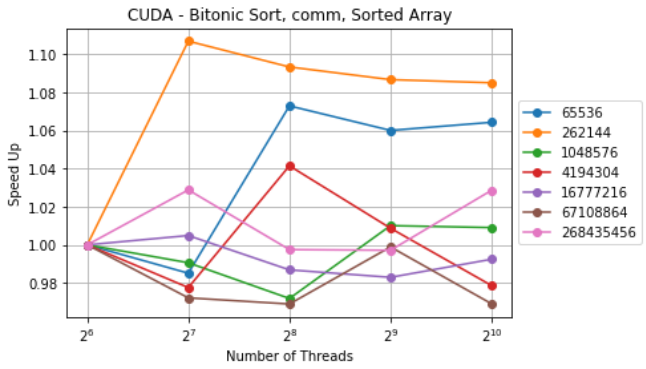




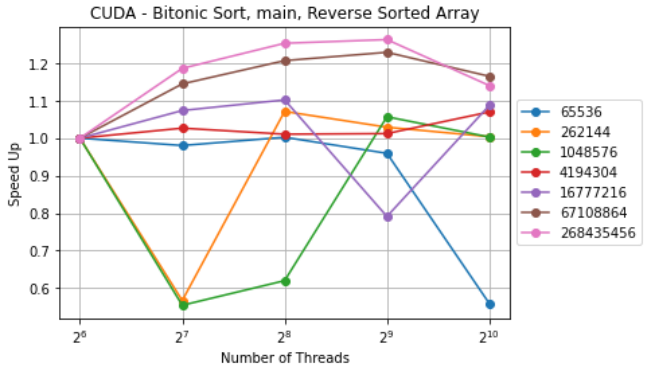
SORTED INPUT ARRAY

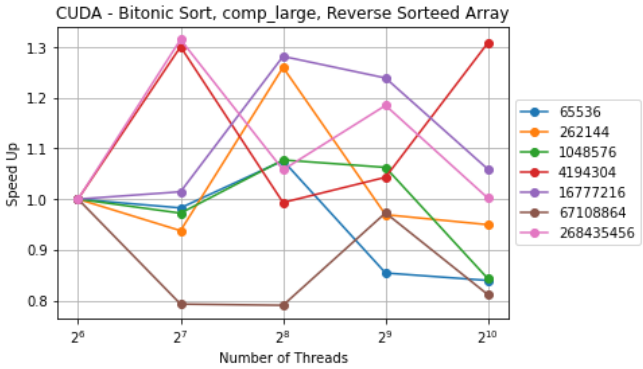


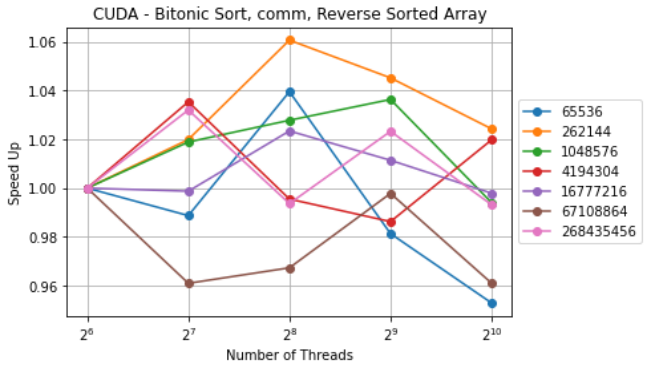




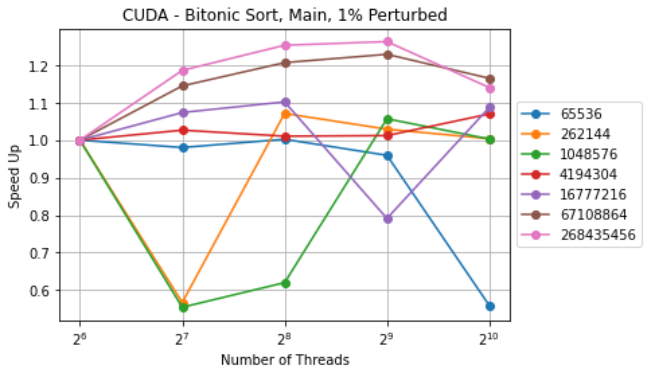
REVERSE SORTED INPUT ARRAY

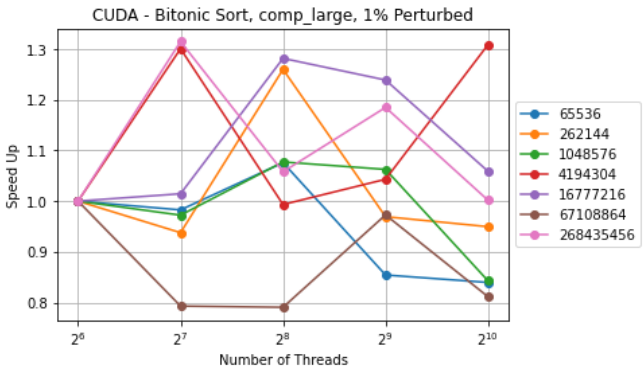


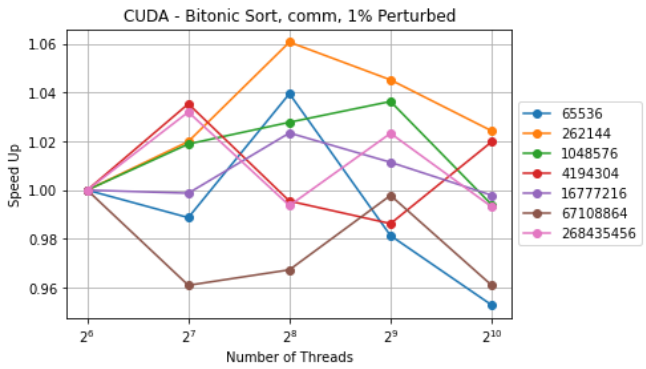




1% PERTURBED INPUT ARRAY







Selection Sort

Weak Scaling

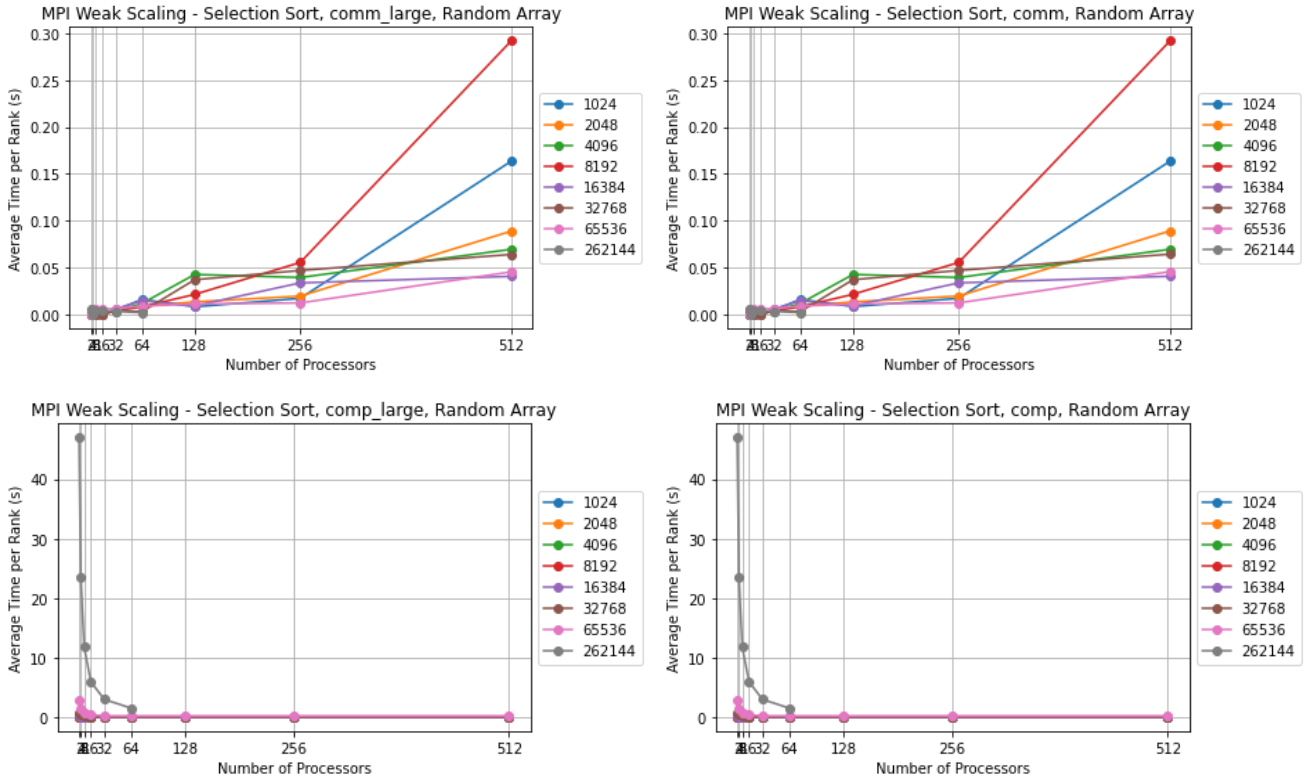
MPI

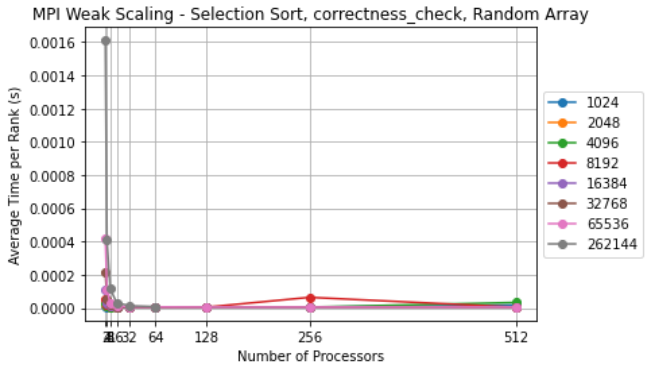
The graphs for MPI Weak Scaling are assesing the average time per rank for each set of number of processors. The input types for MPI Weak Scaling started from 2^10 and included 2^11, 2^12, 2^13, 2^14, 2^15, 2^16.

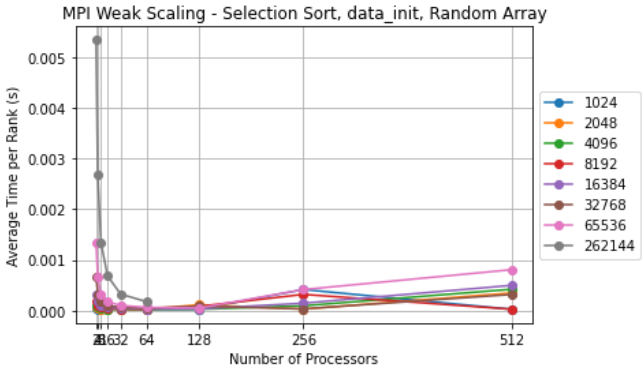
The graph analysis did not include input sizes higher than 2^16. The files to run 2^18 and above timed out due to memory thrashing. Because selection sort is an inefficient sort and takes a lot of type to sort the array, oftentimes the files for higher input sizes timed out. Therefore, the graphs were analyzed using a smaller set of numbers starting from 2^10. In addition, for the number of processors, the highest number of processors that the program ran for was 512. The output files for 1024 processors gave memory errors and even after increasing the memory in the job file to 128 and 256GB, the files were still outputting memory errors. Therefore the graphs run up to 512 processors.

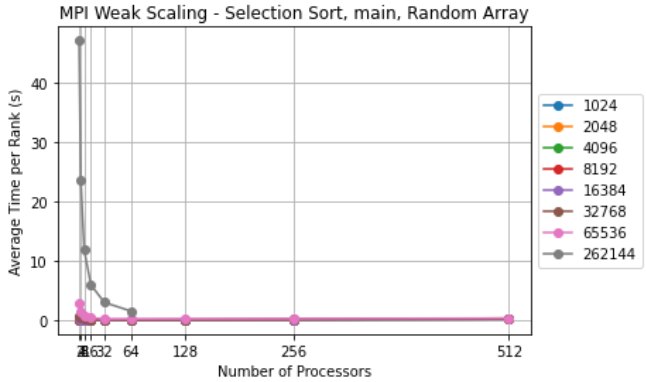
For the general trends of the graphs, the communication graphs increase as the nunmber of processors increases. This is because there are more processors to communicate, therefore, the relationship is directly proportional. However, for the computation graphs as the number of processors increases, there is a general decrease in the average time. This is due to parallelizing and spreading the inputs across the various processors. Therefore, as there are more processors, the time should often decrease. Due to the input sizes being small, it is quite hard to see the differentiation of the various input sizes, but for the computation graphs there is a general decrease in time.

RANDOM INPUT ARRAY

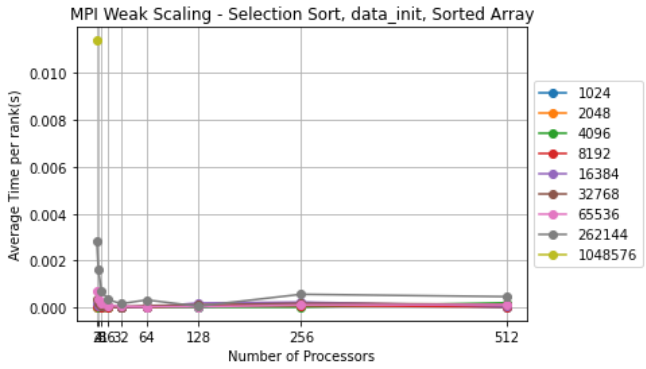


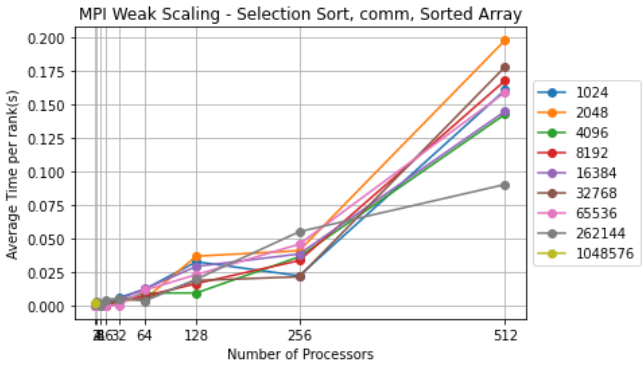


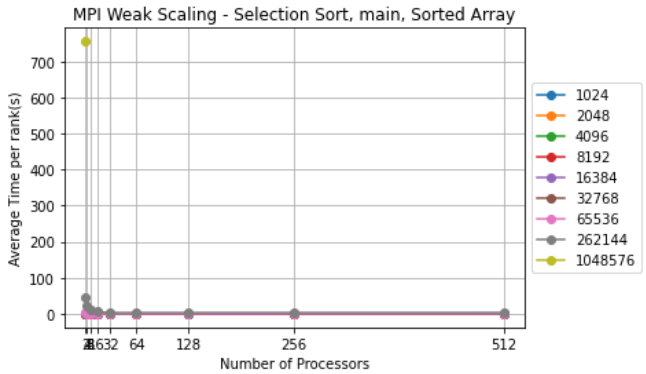


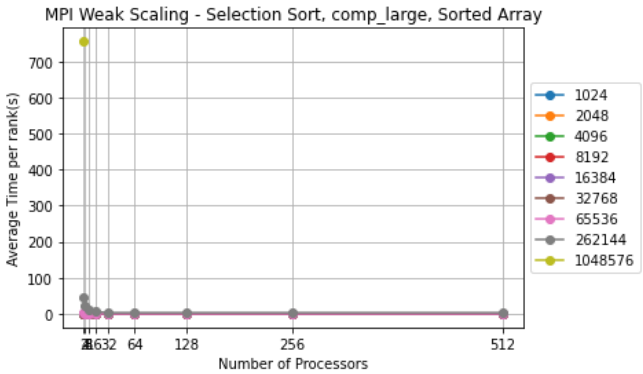


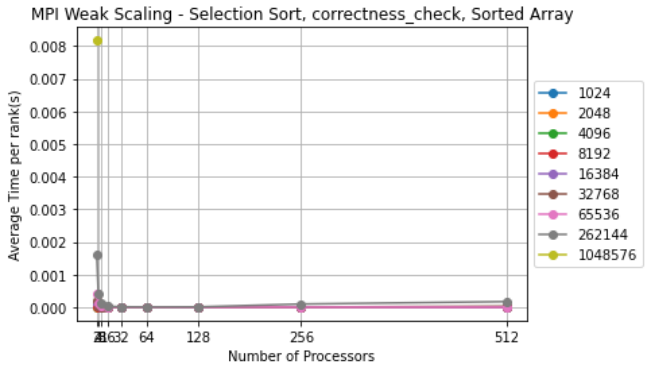
SORTED INPUT ARRAY

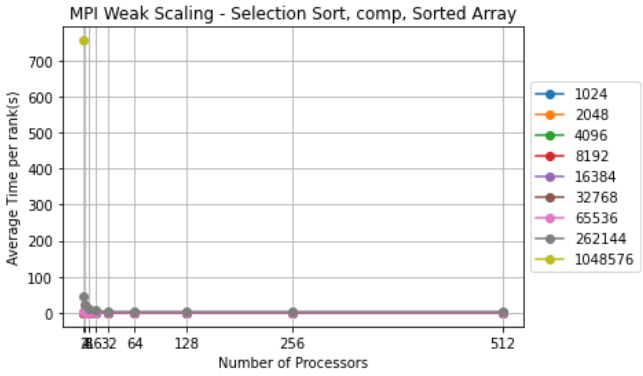


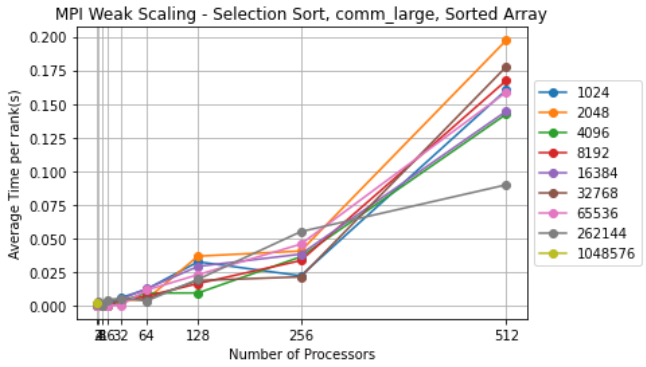




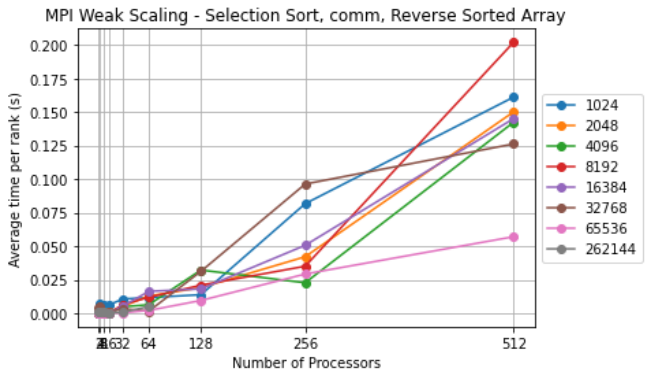


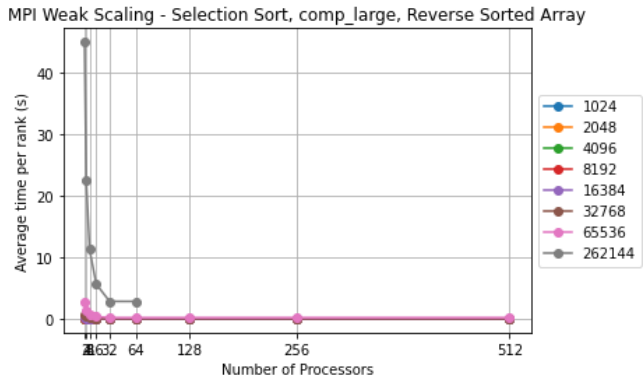


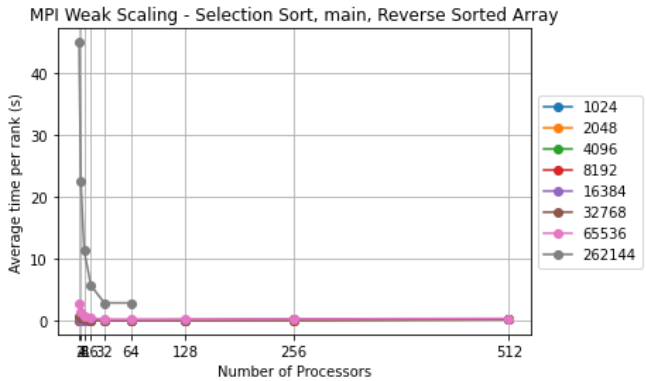


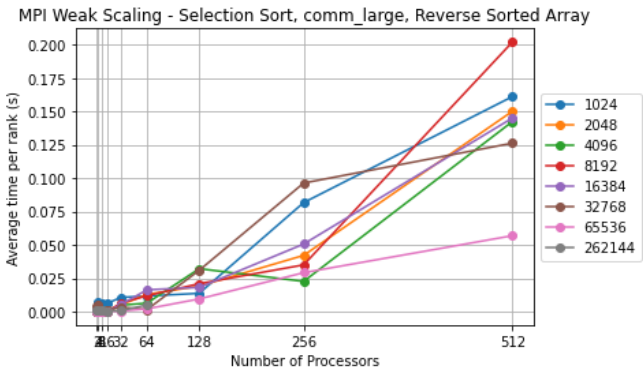


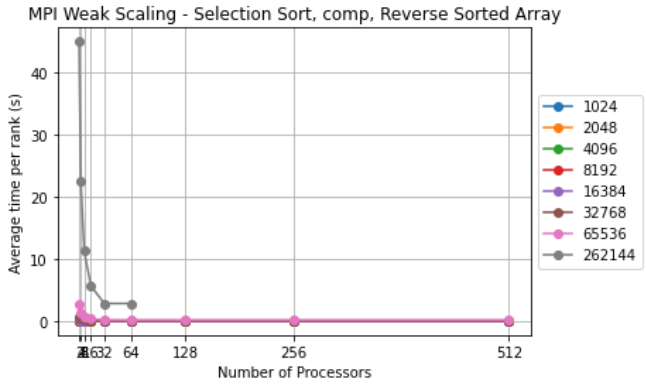
REVERSE SORTED INPUT ARRAY

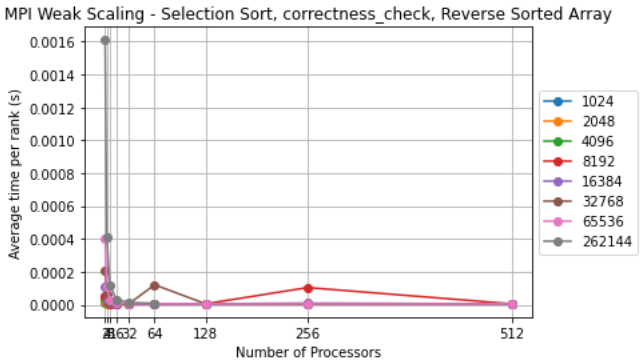


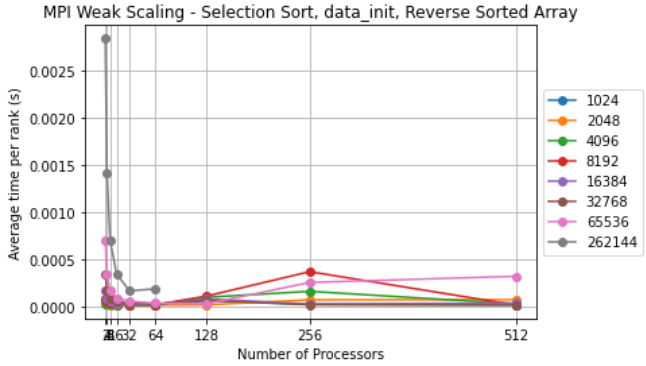




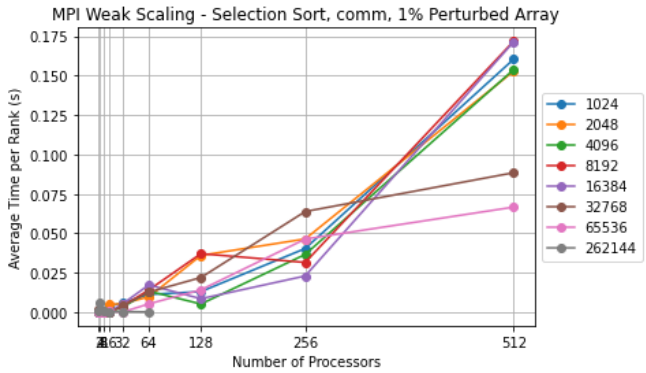


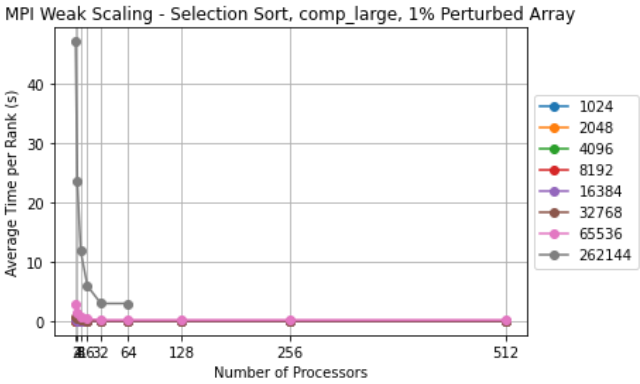


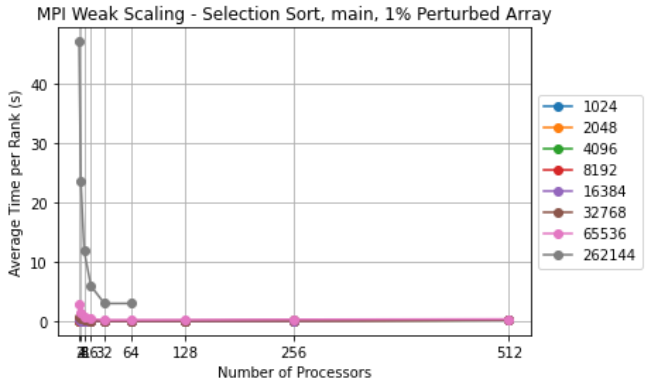


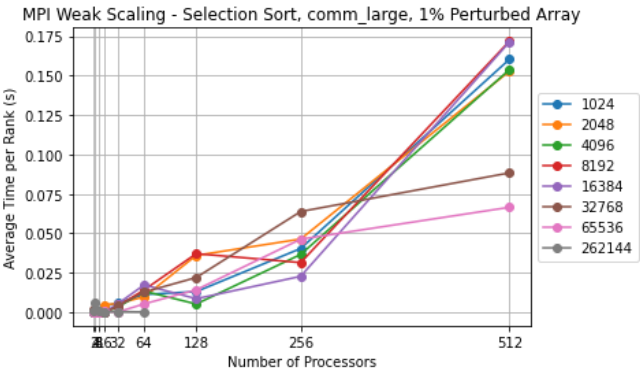


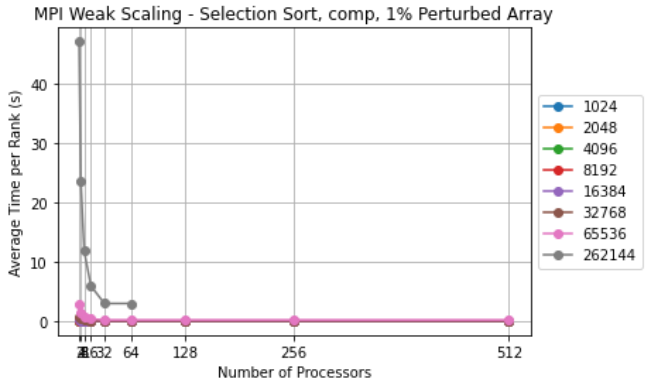
1% PERTURBED INPUT ARRAY

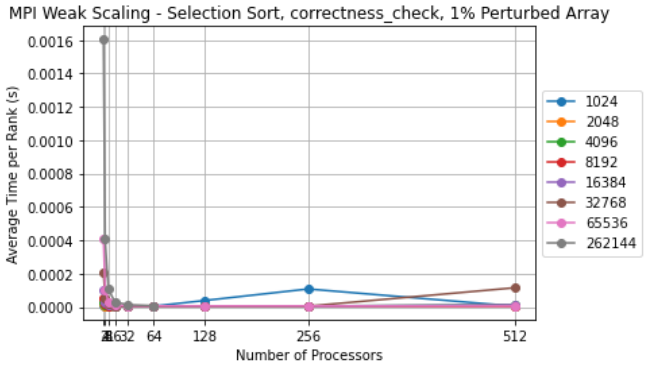


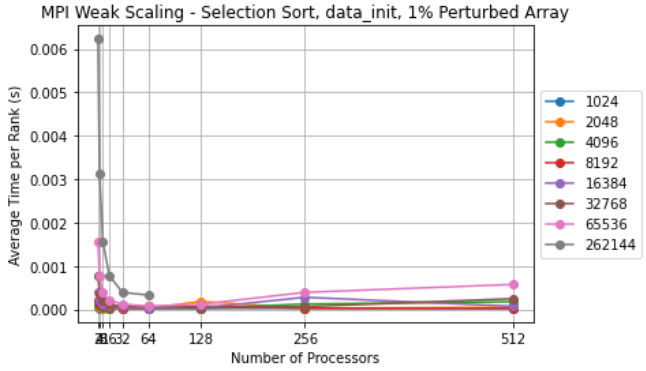












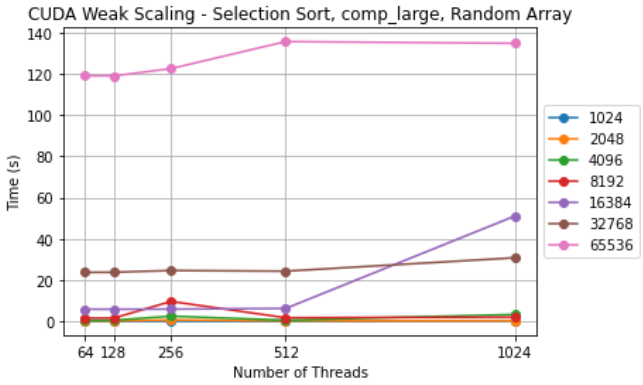
CUDA

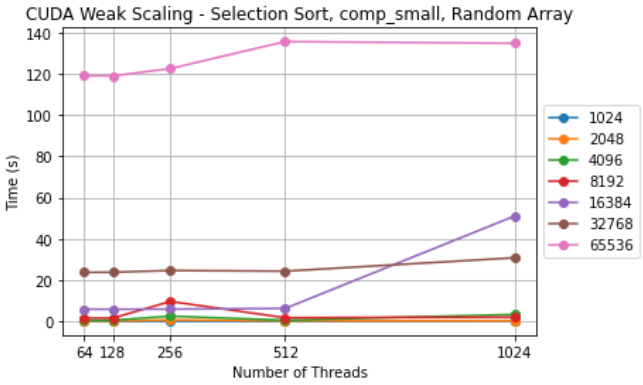
The following graphs are the weak scaling for CUDA.

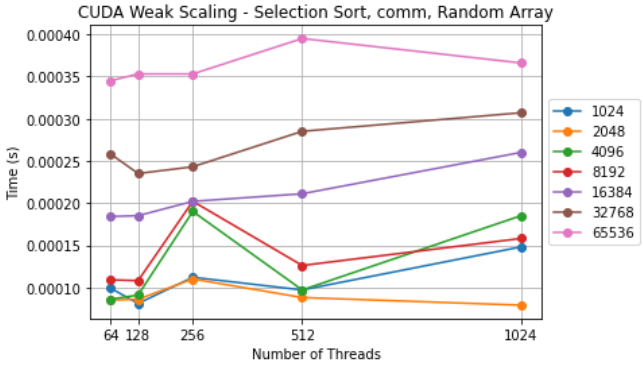
The comp_large graphs were generally quite flat across the graphs even as the number of threads increased. However, for 2^16 the line was significantly higher than the rest as it was the highest input size. The same trend is seen on the comp_small graphs. For 2^14 there is an increase from 512 to 1024 processors. For the comm graphs, the general trend showed an increase for the smaller input sizes. However, for the larger input sizes it increased until 512 threads and then started decreasing. For the main graphs, the general trend was a flat line as the number of processors increased. There was an outlier of increasing at 16384 for 512 threads, but if there were multiple trials, the general trend should follow the rest of the inputs. For correctess check, the general trend was an increase in time after 512 threads. There was a strange spike of increase at 256 threads and then a sharp decline at 512 threads for 2^12 and 2^13, however, the rest of the input sizes had a general trend of increase after 512 threads. For data init, the general trend was an increase from the initial 64 threads to 1024 threads, however, 2^15 and 2^16 had an increase spike at 256 threads. Then they decreased in time for 512 threads. Finally for comp, there is a steady flat line with slight increases. However, 2^16 took significantly more time compared to the rest of the input sizes.

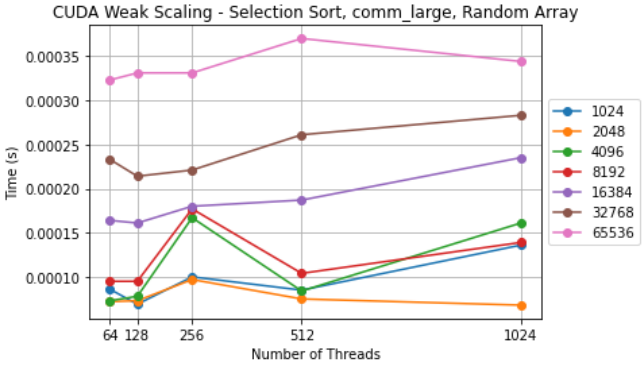
All of the array types had similar trends, however reverse sorted took less time than the rest of the array types. 1% perturbed took quite a long time compared to the others.

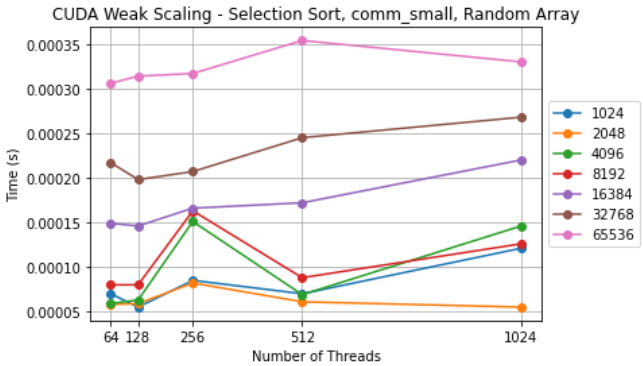
RANDOM INPUT ARRAY

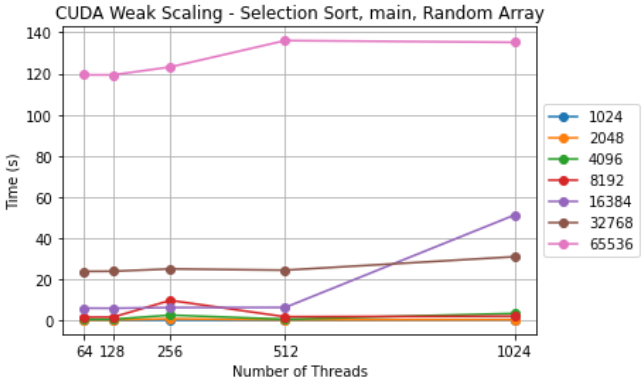

CUDA Weak Scaling - Selection Sort, comp_large, Random Array
Time (s)
Number of Threads
1024
2048
4096
8192
16384
32768
65536

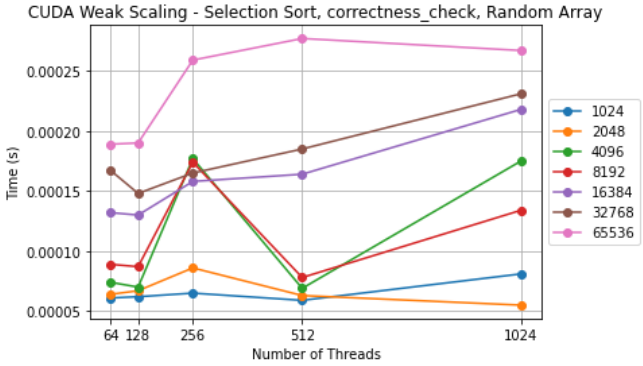

CUDA Weak Scaling - Selection Sort, comp_small, Random Array
Time (s)
Number of Threads
1024
2048
4096
8192
16384
32768
65536

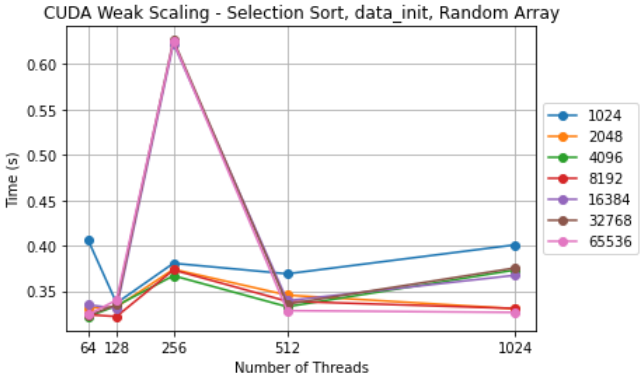

CUDA Weak Scaling - Selection Sort, comm, Random Array
Time (s)
Number of Threads
1024
2048
4096
8192
16384
32768
65536

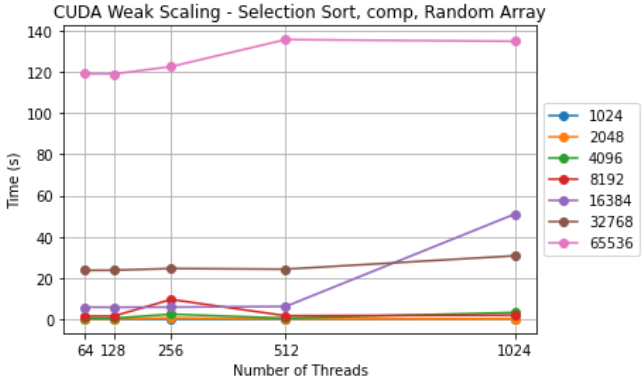

CUDA Weak Scaling - Selection Sort, comm_large, Random Array
Time (s)
Number of Threads
1024
2048
4096
8192
16384
32768
65536


CUDA Weak Scaling - Selection Sort, comm_small, Random Array
Time (s)
Number of Threads
1024
2048
4096
8192
16384
32768
65536

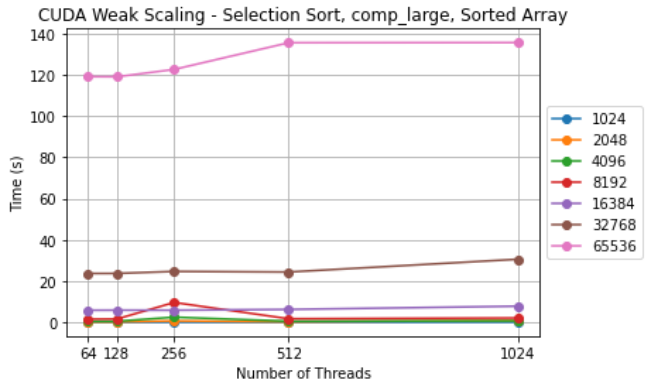

CUDA Weak Scaling - Selection Sort, main, Random Array
Time (s)
Number of Threads
1024
2048
4096
8192
16384
32768
65536

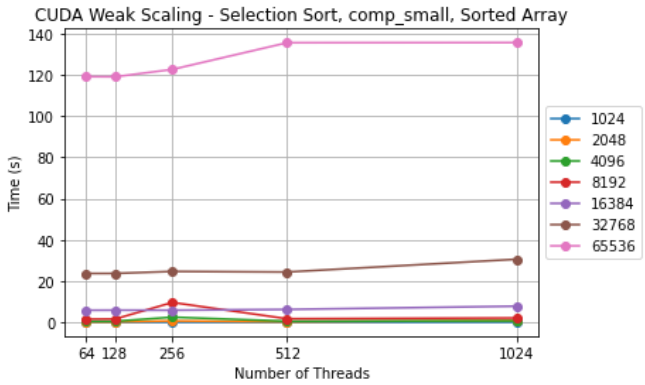

CUDA Weak Scaling - Selection Sort, correctness_check, Random Array
Time (s)
Number of Threads
1024
2048
4096
8192
16384
32768
65536

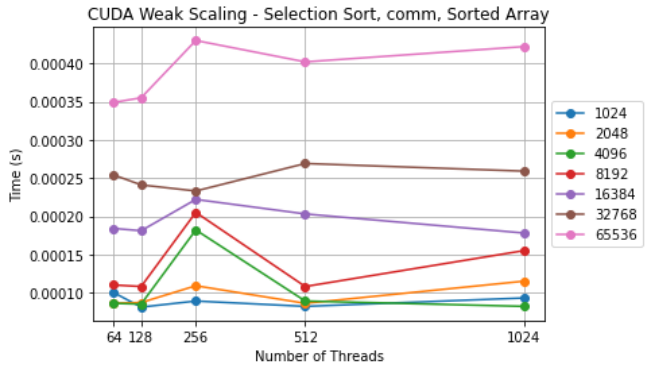

CUDA Weak Scaling - Selection Sort, data_init, Random Array
Time (s)
Number of Threads
1024
2048
4096
8192
16384
32768
65536

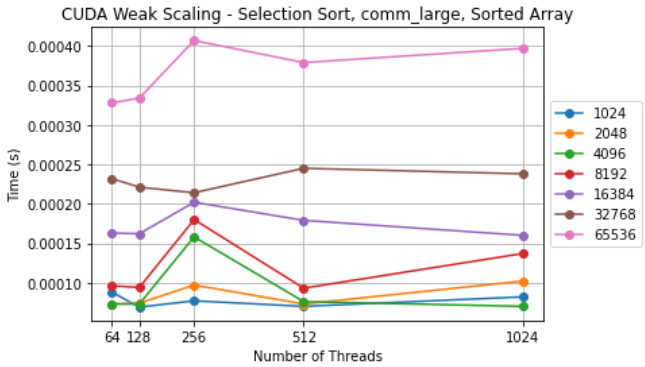

CUDA Weak Scaling - Selection Sort, comp, Random Array
Time (s)
Number of Threads
1024
2048
4096
8192
16384
32768
65536

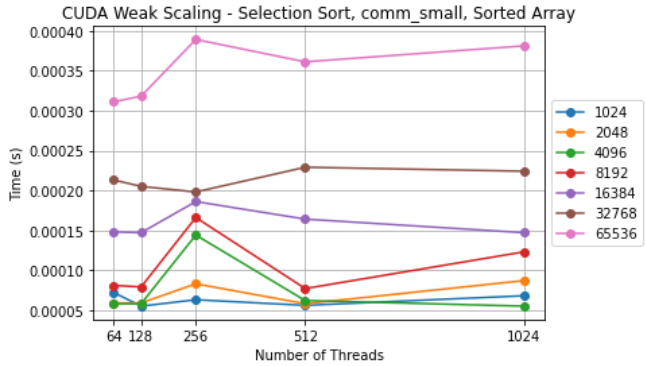
SORTED INPUT ARRAY

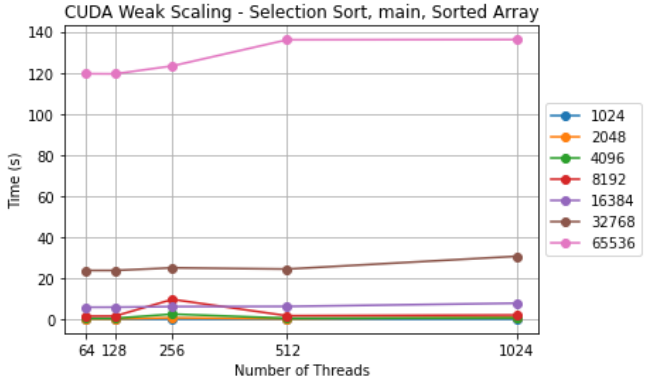


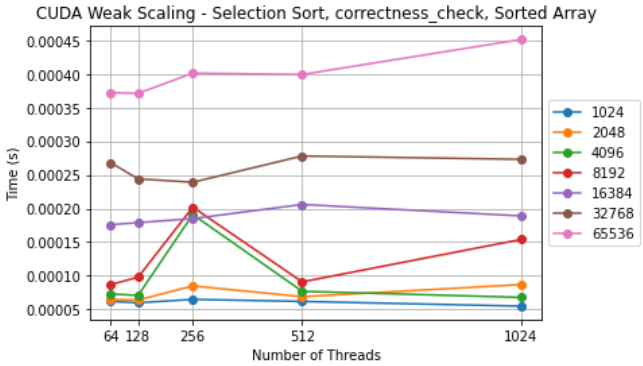


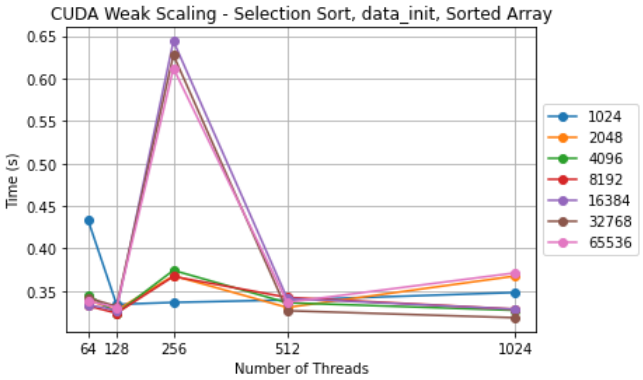


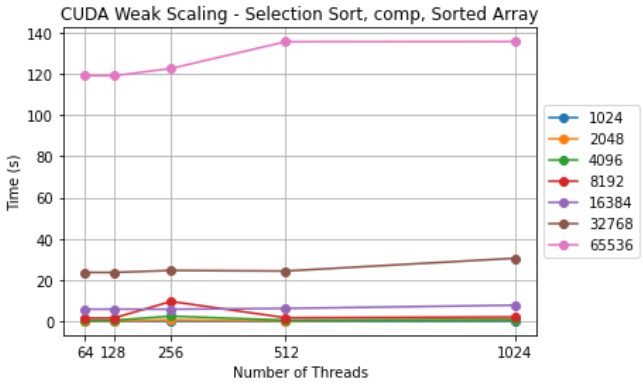




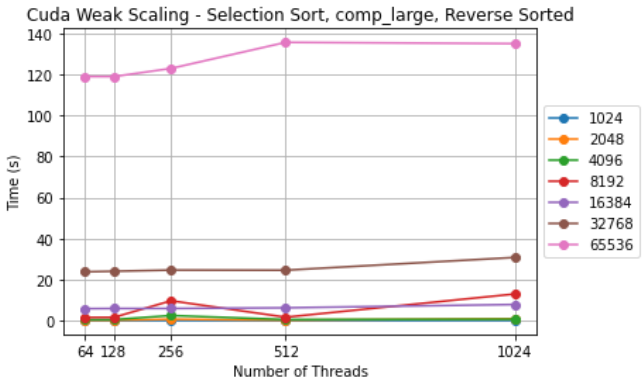


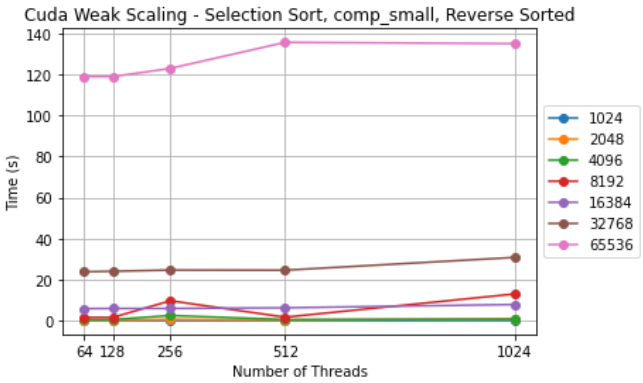


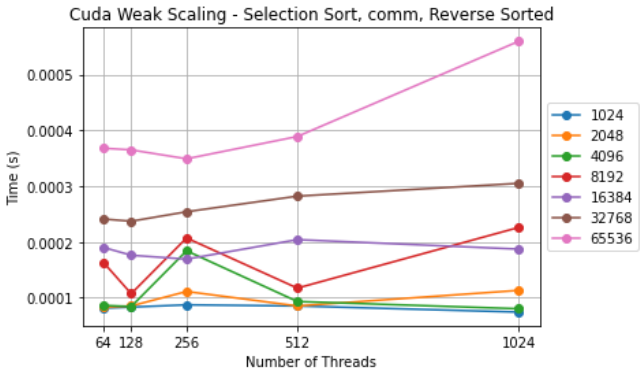


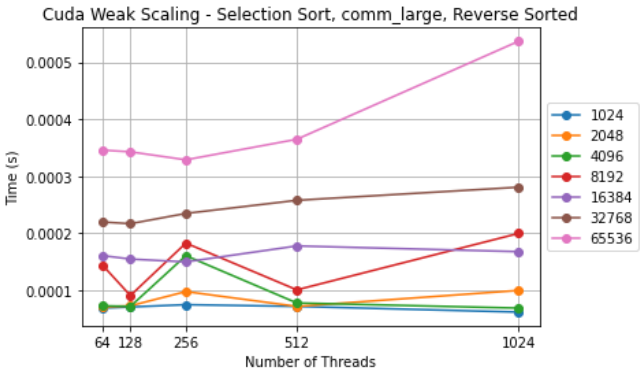


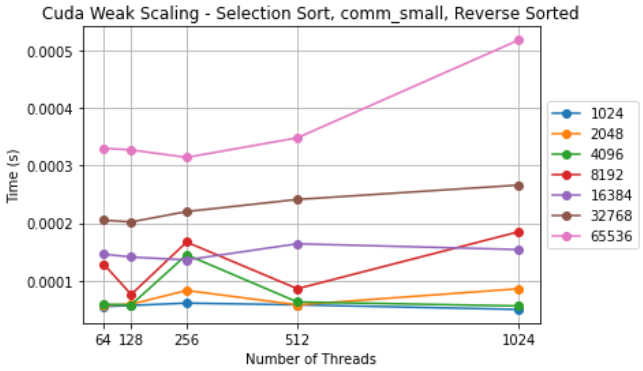
REVERSE SORTED INPUT ARRAY

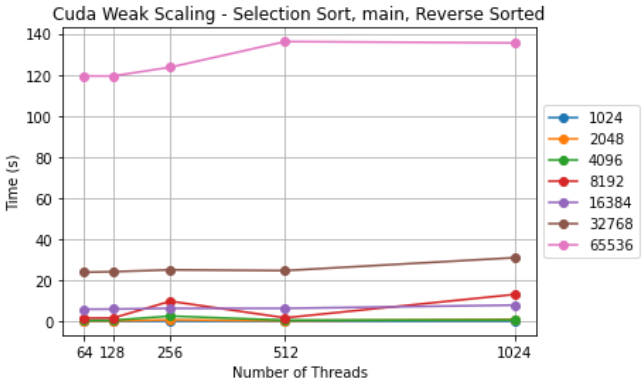


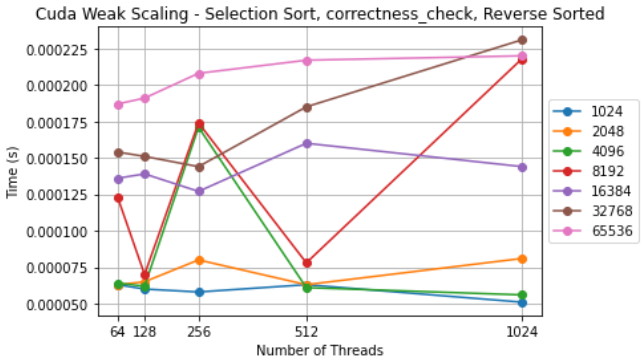


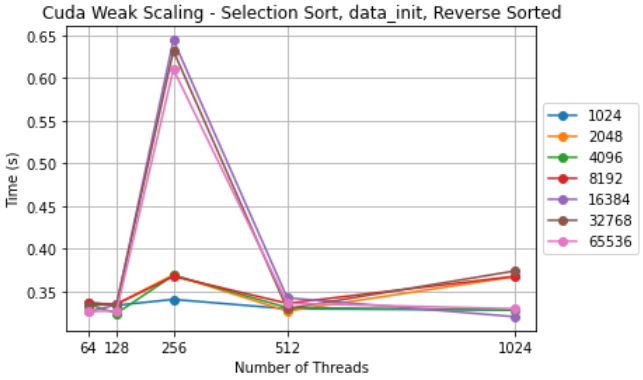


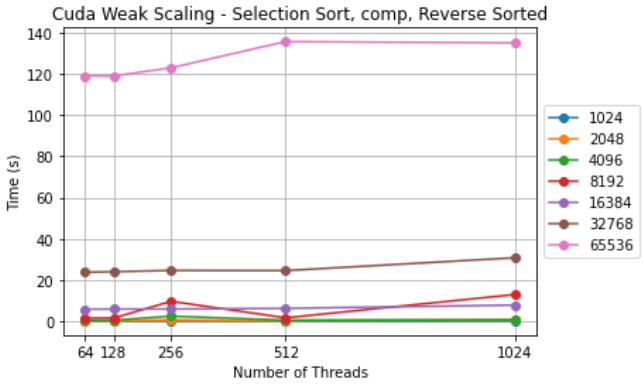




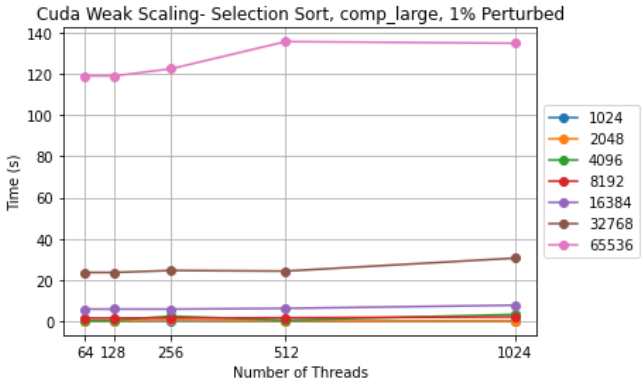


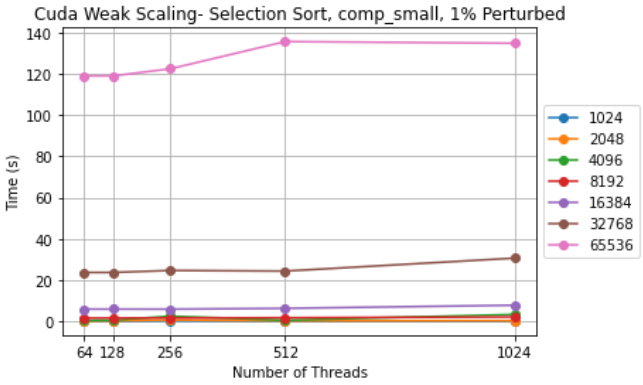


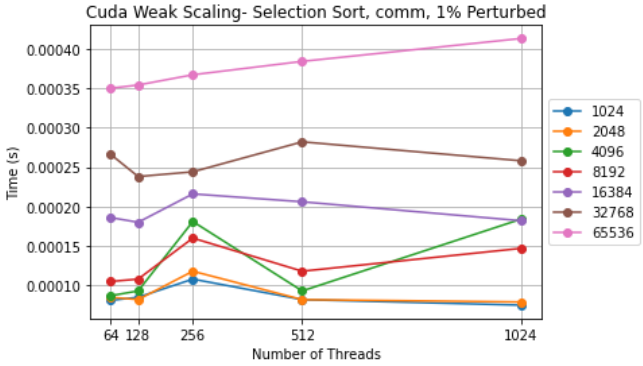


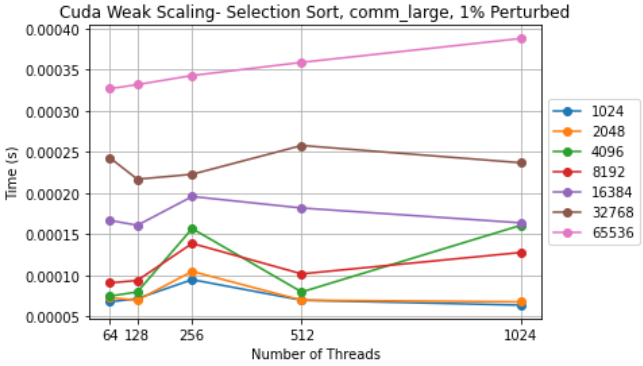


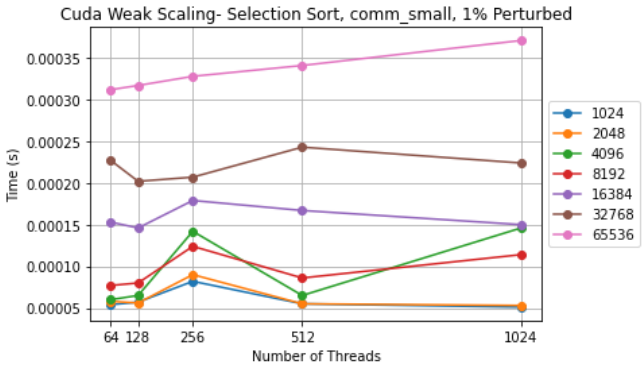
1% PERTURBED INPUT ARRAY

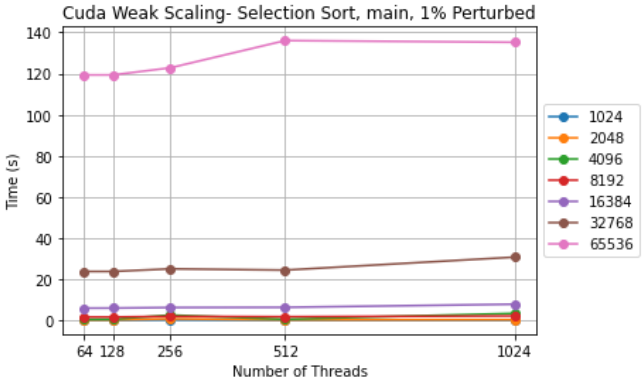

Cuda Weak Scaling- Selection Sort, comp_large, 1% Perturbed
Time (s)
Number of Threads
64 128 256 512 1024
1024
2048
4096
8192
16384
32768
65536

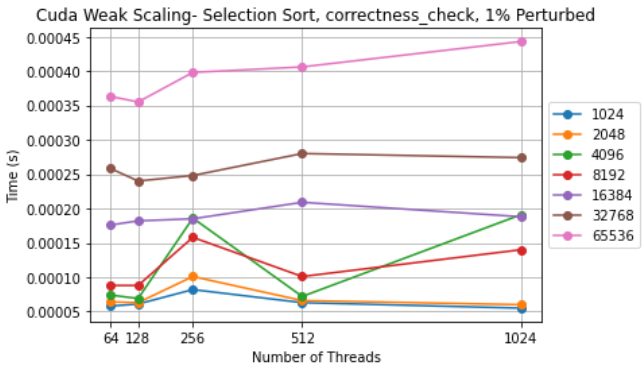

Cuda Weak Scaling- Selection Sort, comp_small, 1% Perturbed
Time (s)
Number of Threads
64 128 256 512 1024
1024
2048
4096
8192
16384
32768
65536

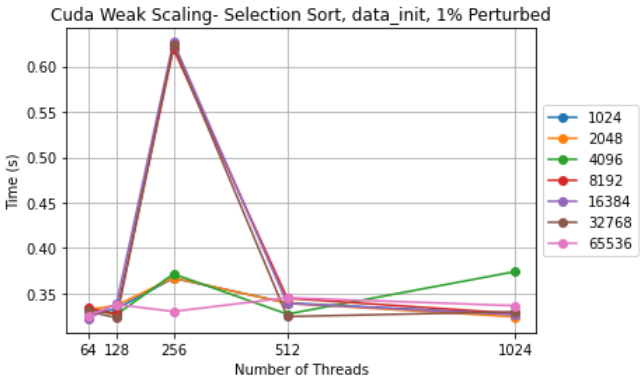

Cuda Weak Scaling- Selection Sort, comm, 1% Perturbed
Time (s)
Number of Threads
64 128 256 512 1024
1024
2048
4096
8192
16384
32768
65536

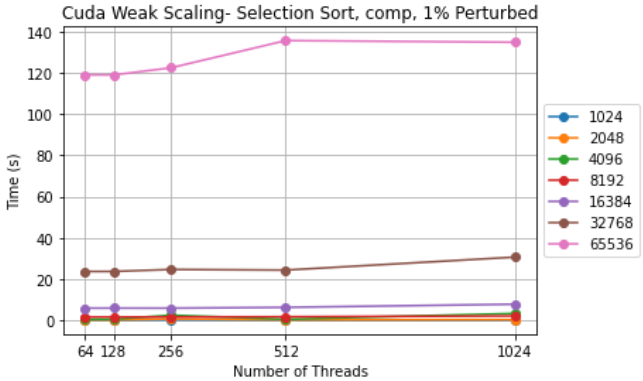

Cuda Weak Scaling- Selection Sort, comm_large, 1% Perturbed
Time (s)
Number of Threads
64 128 256 512 1024
1024
2048
4096
8192
16384
32768
65536


Cuda Weak Scaling- Selection Sort, comm_small, 1% Perturbed
Time (s)
Number of Threads
64 128 256 512 1024
1024
2048
4096
8192
16384
32768
65536


Cuda Weak Scaling- Selection Sort, main, 1% Perturbed
Time (s)
Number of Threads
64 128 256 512 1024
1024
2048
4096
8192
16384
32768
65536


Cuda Weak Scaling- Selection Sort, correctness_check, 1% Perturbed
Time (s)
Number of Threads
64 128 256 512 1024
1024
2048
4096
8192
16384
32768
65536


Cuda Weak Scaling- Selection Sort, data_init, 1% Perturbed
Time (s)
Number of Threads
64 128 256 512 1024
1024
2048
4096
8192
16384
32768
65536


Cuda Weak Scaling- Selection Sort, comp, 1% Perturbed
Time (s)
Number of Threads
64 128 256 512 1024
1024
2048
4096
8192
16384
32768
65536

Strong Scaling

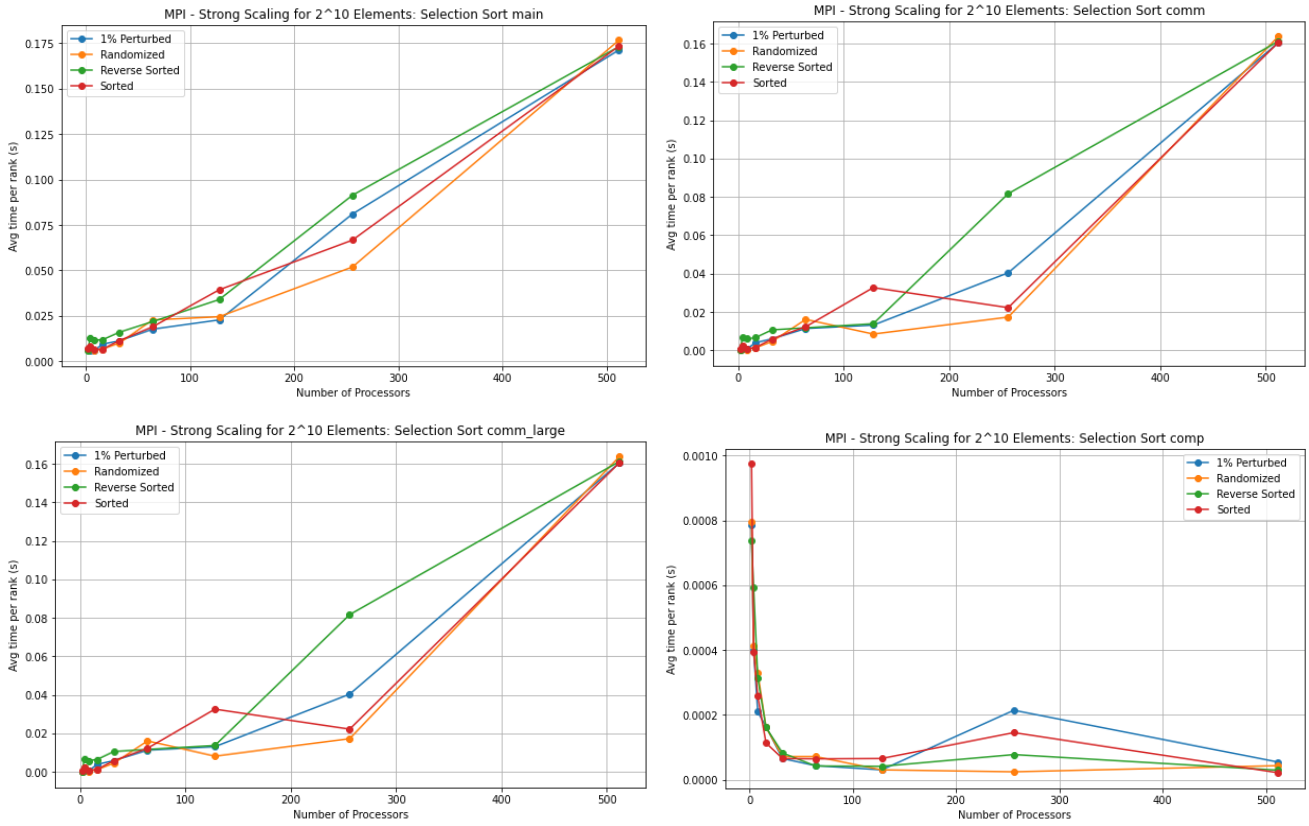
MPI

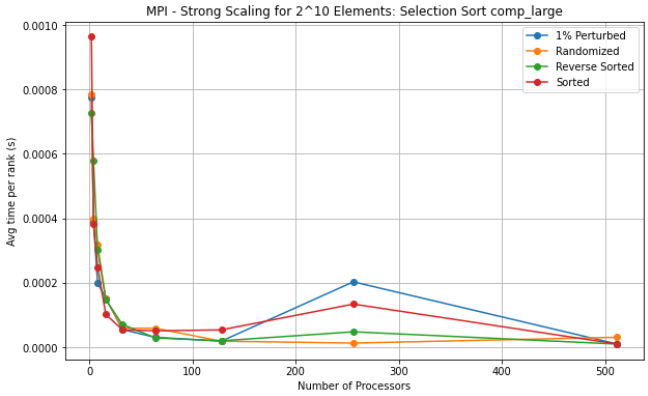
The following graphs are the strong scaling for MPI.

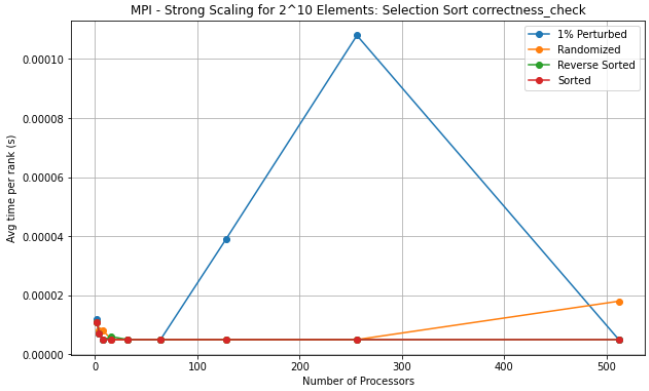
There is a general trend of increase for the comm graphs as the number of processors increases. The relationship can be seen as directly proportional as there are more processors to communicate, therefore, the general trend is an increase. For the computation graphs, there is a decrease in average time as the number of processors increases. This is due to parallelizing as the inputs are spread across multiple processors thus decreasing the average time. The data init graphs are slighty more difficult to find a pattern to as different input sizes are behaving differently and there wasn't a general trend across any of the array types. Therefore, with repeated trials a more general trend might be seen. For correctness check there is a general decrease, however, there are outliers where the average time spikes.

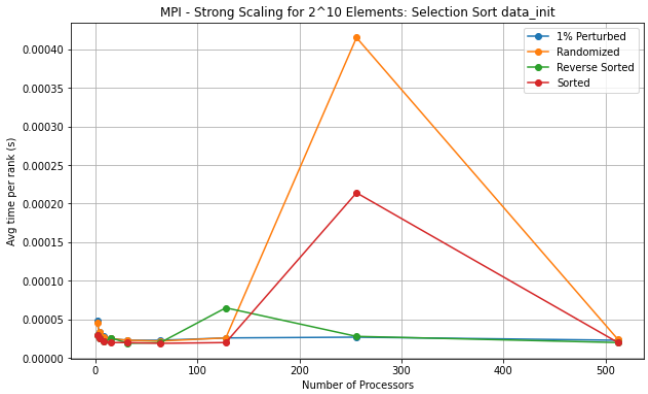
As the input size increase, so does the average time for the computation. Since there are more inputs to handle the time should generally increase. For example the y axis time values for 2^10 are smaller than those for 2^11 or 2^12. Due to the input sizes being smaller, the different types of arrays have similar trends and there is not really a specific one that is seen to perform significantly worse than the rest.

2^10 Elements

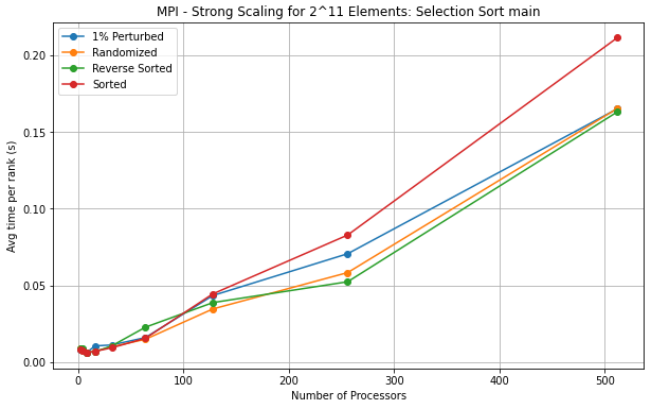


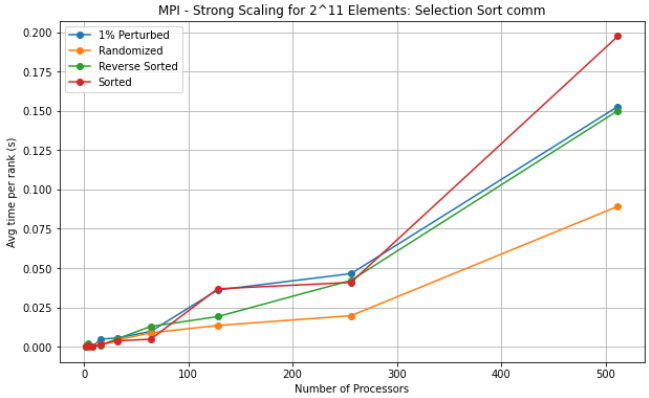


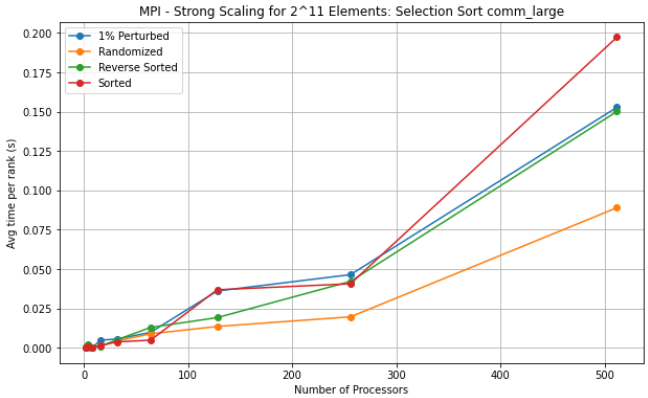


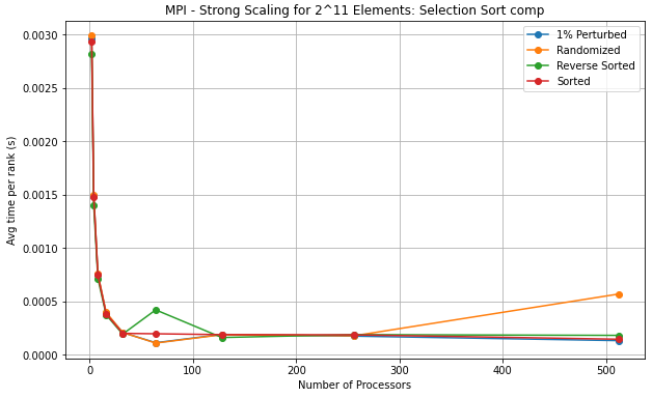


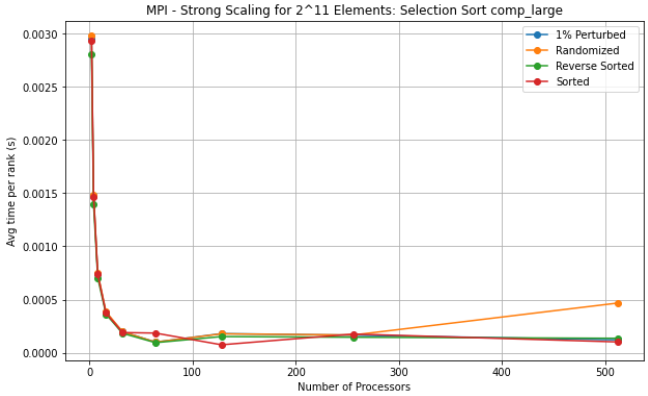
2^11 Elements

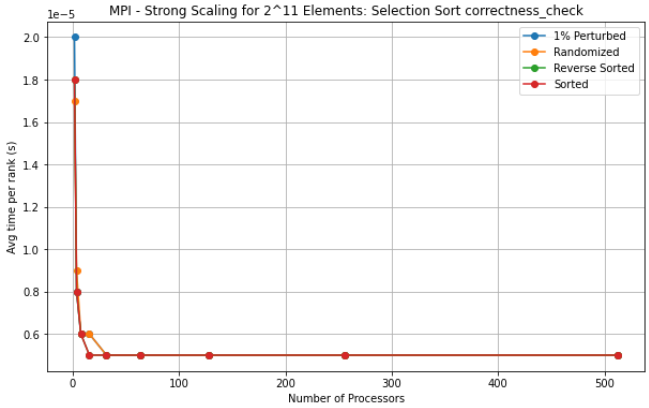


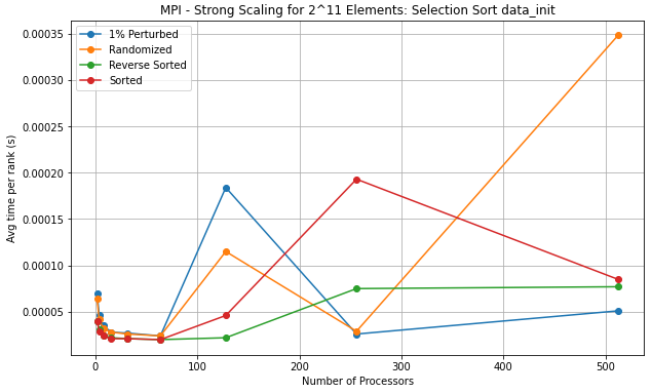




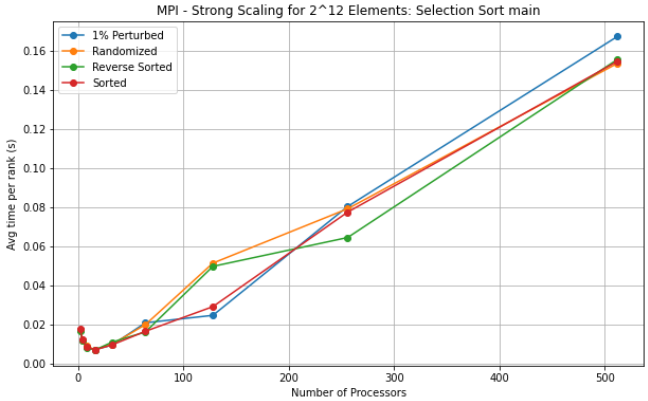


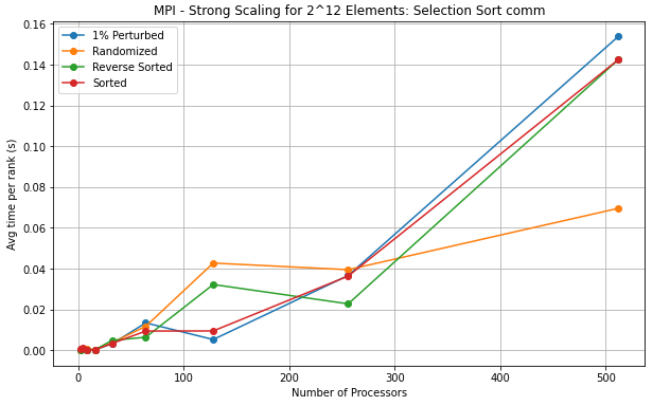


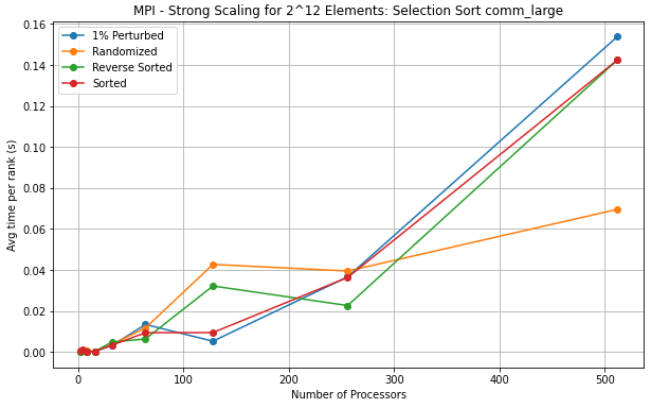


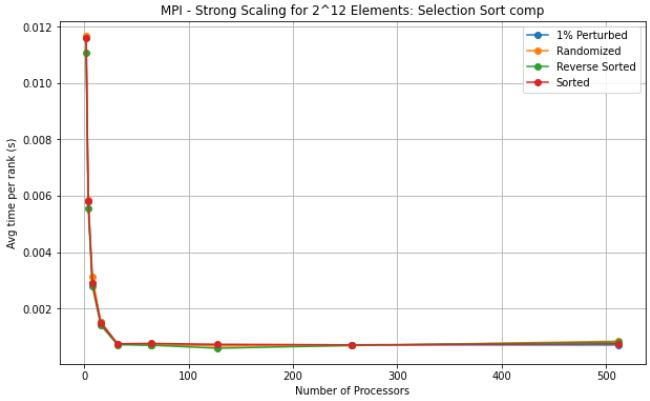


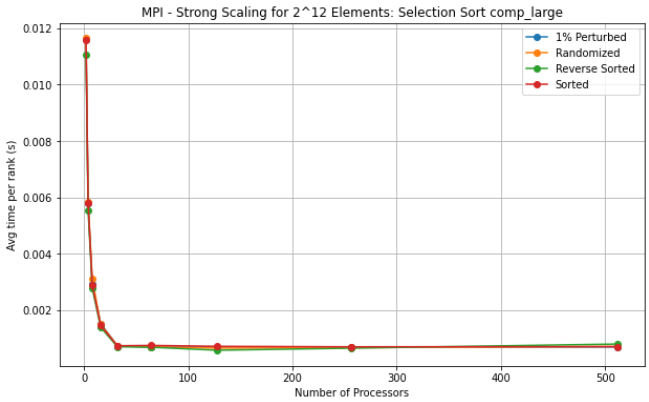
2^12 Elements

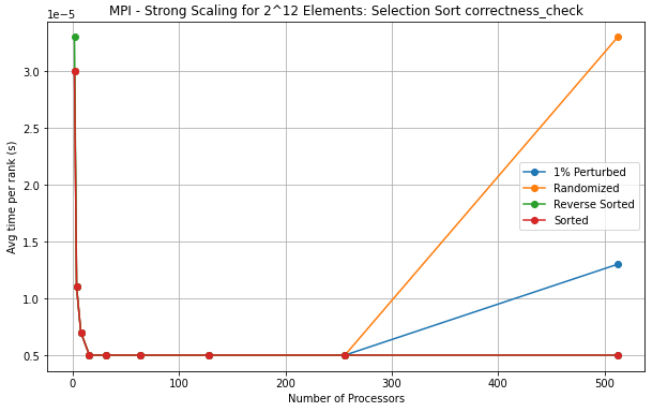


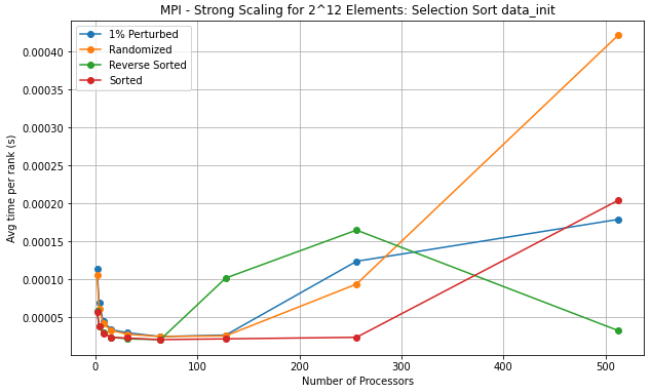




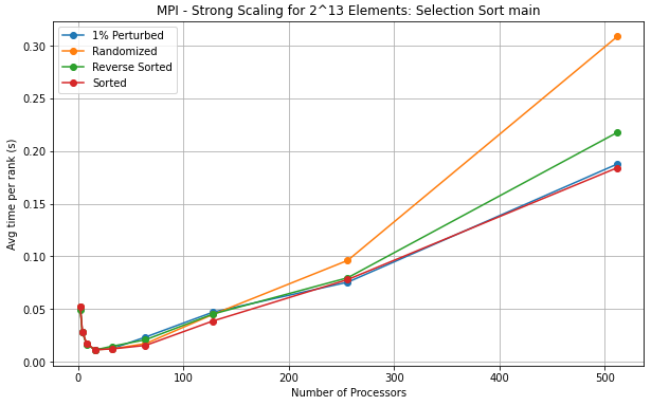


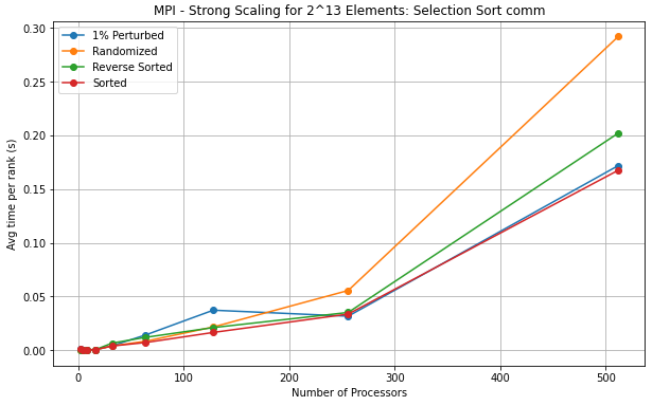


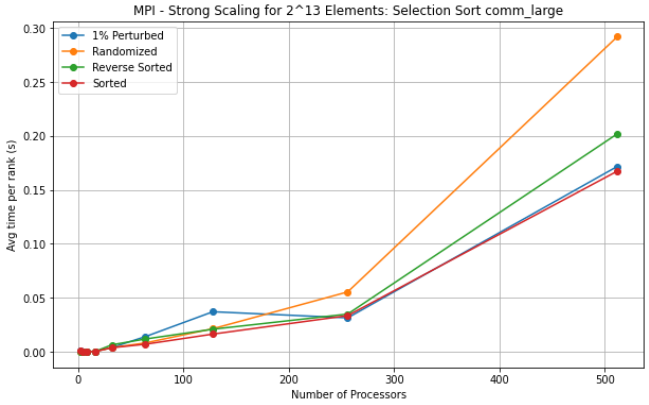


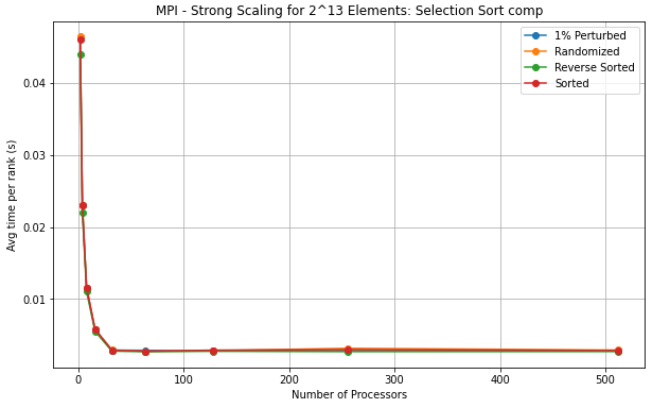


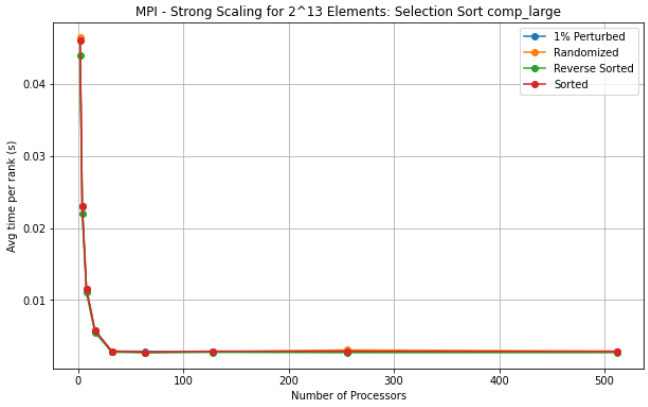
2^13 Elements

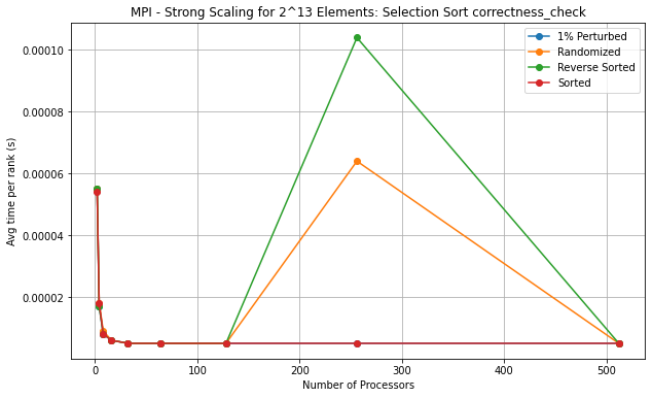


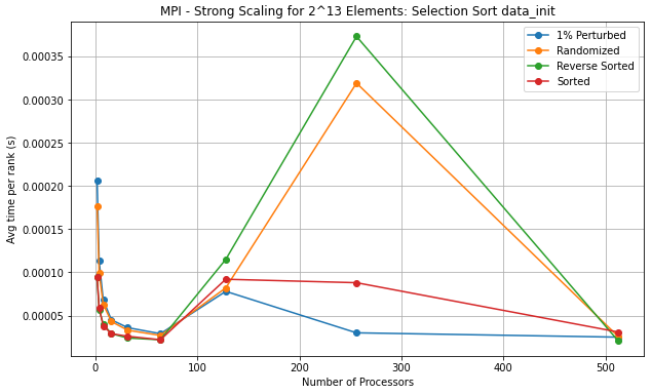




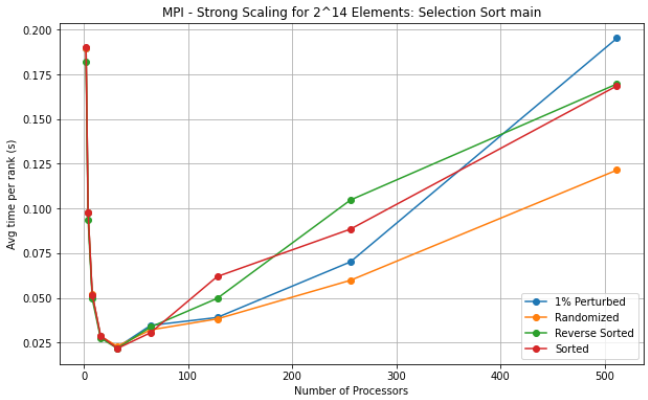


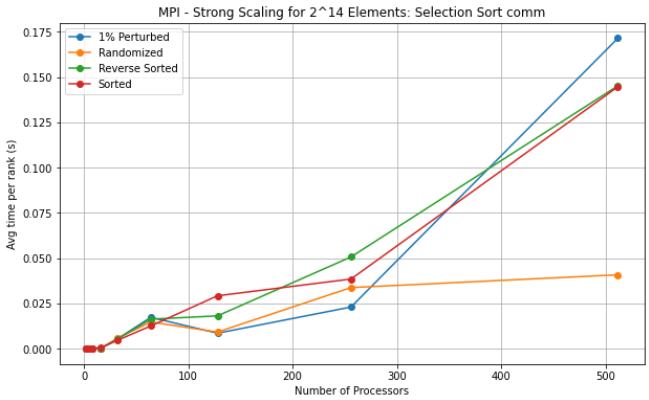


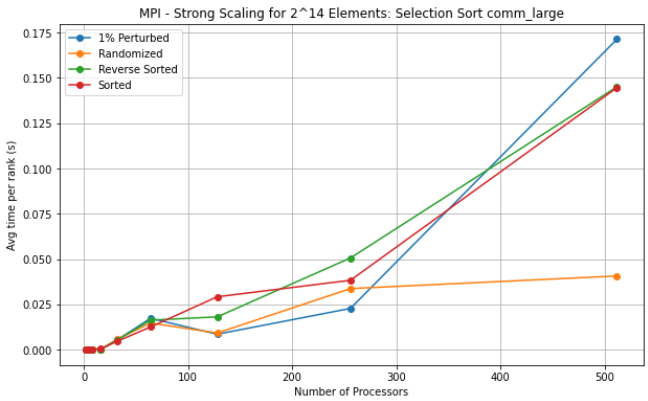


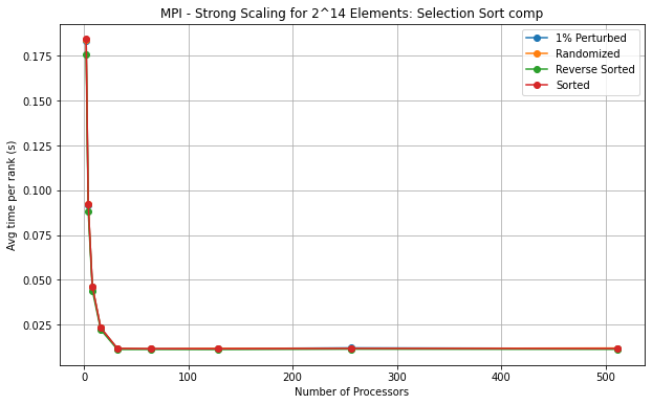


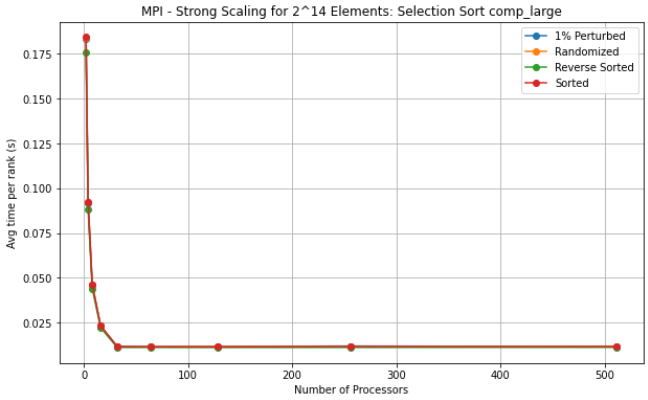
2^14 Elements

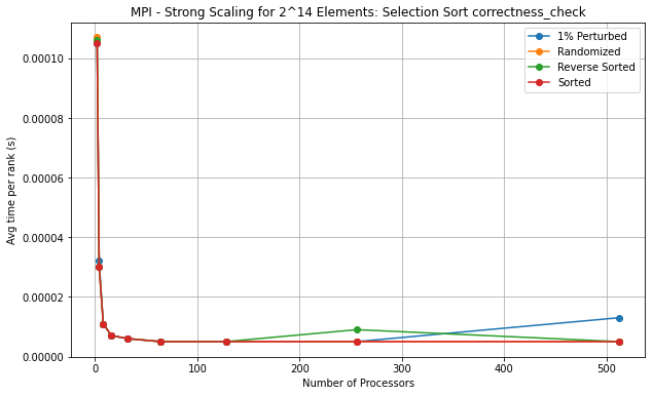


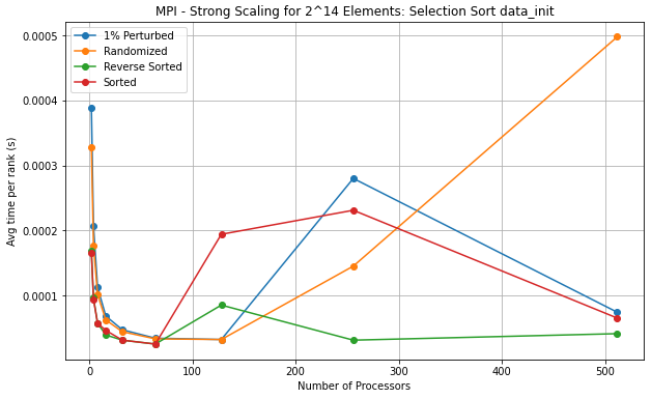




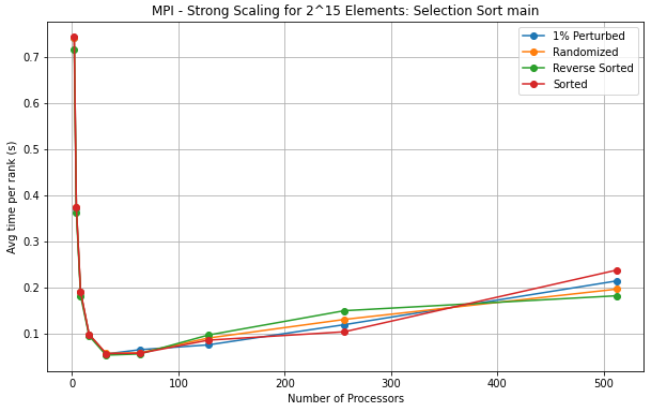


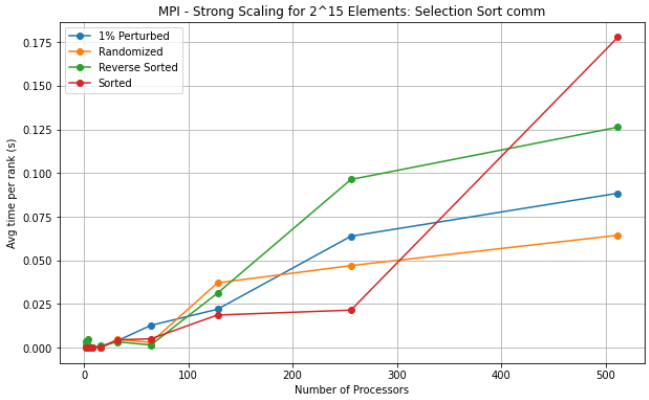


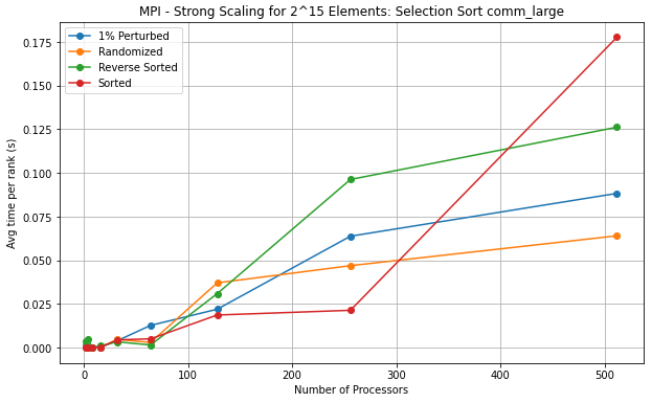


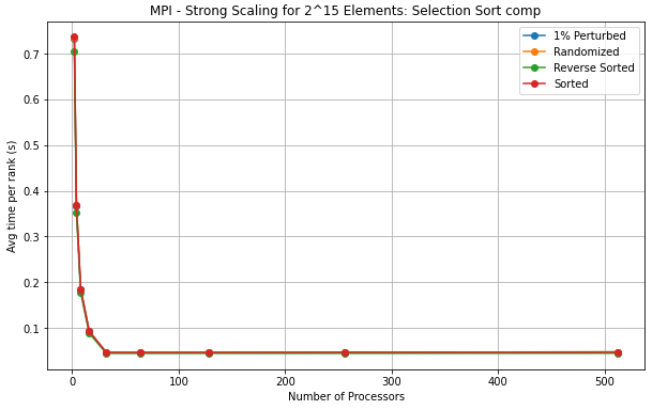


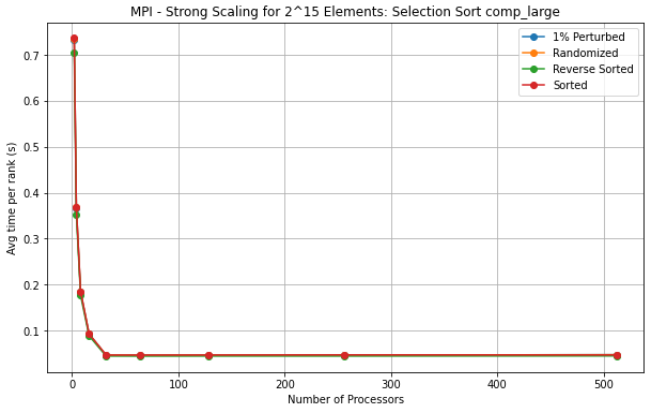
2^15 Elements

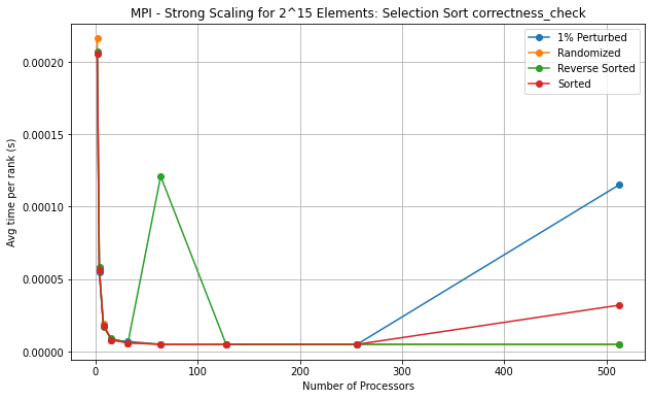


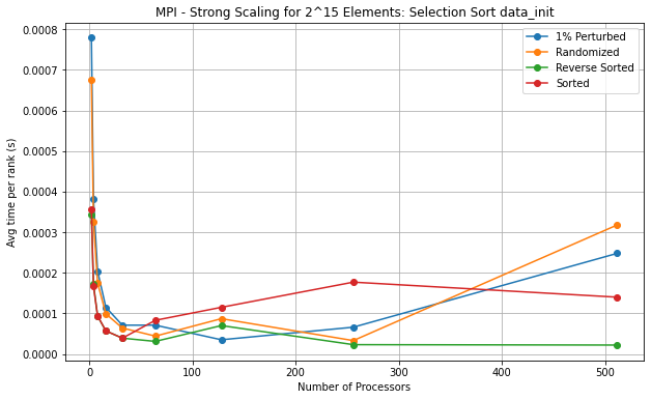




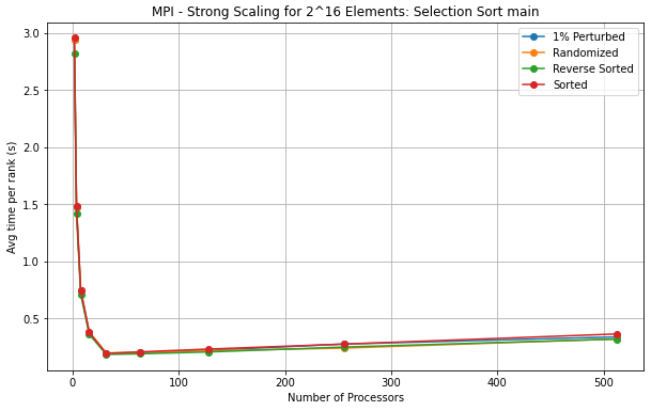


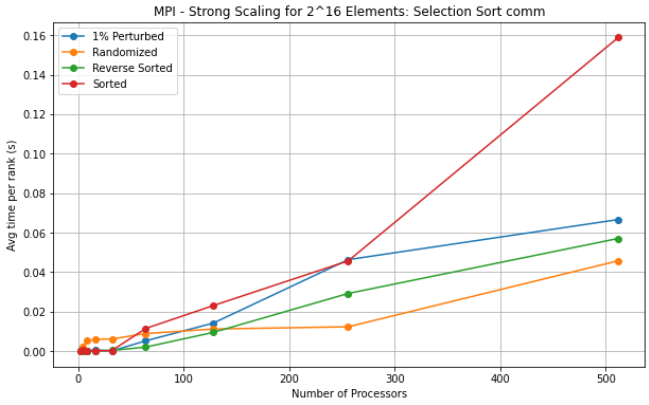


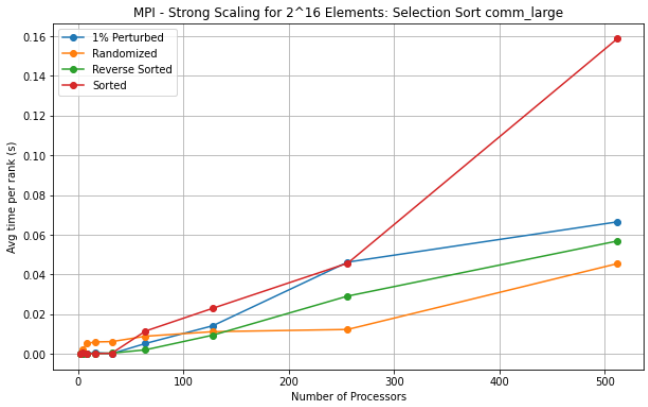


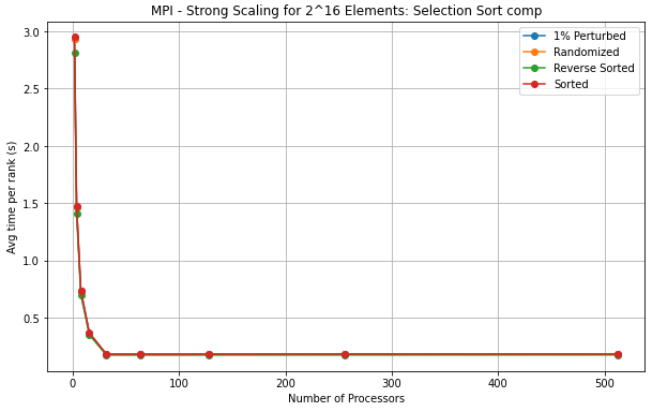


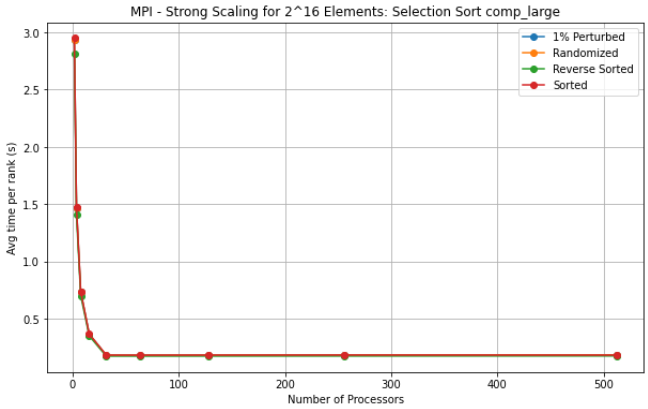
2^16 Elements

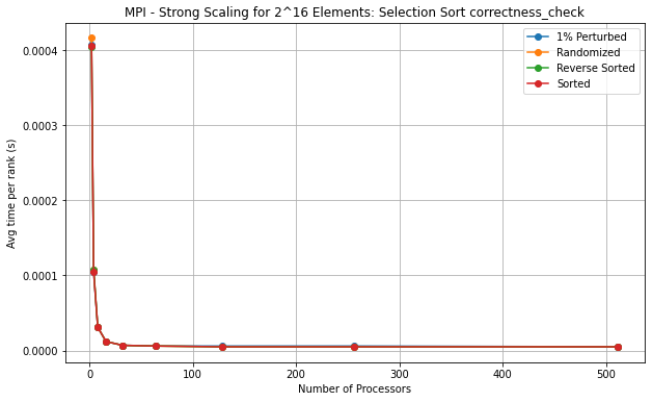


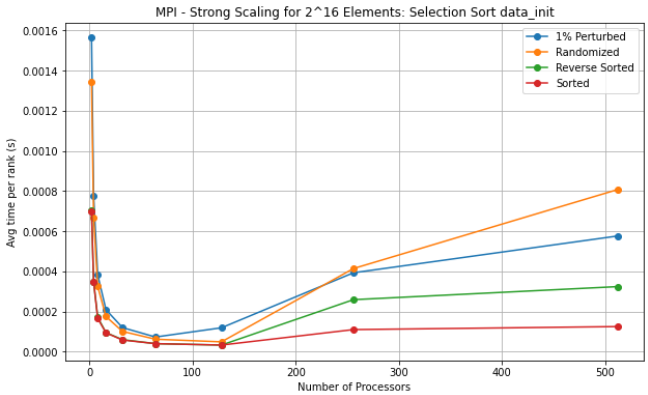










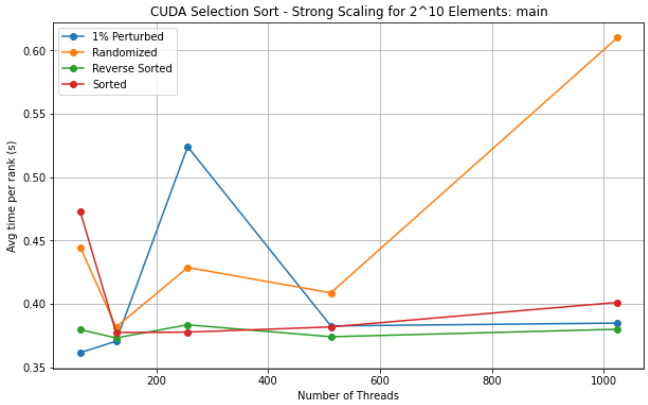


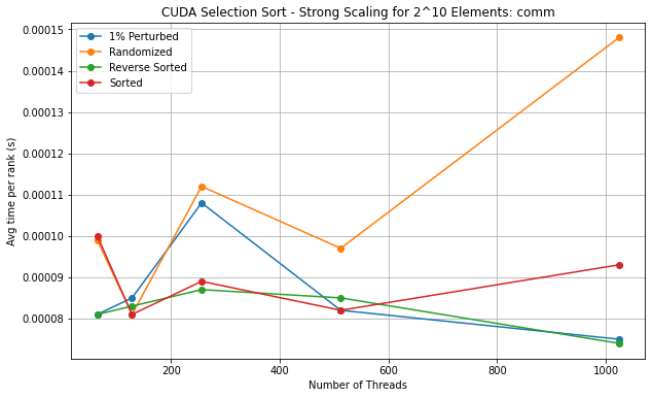
CUDA

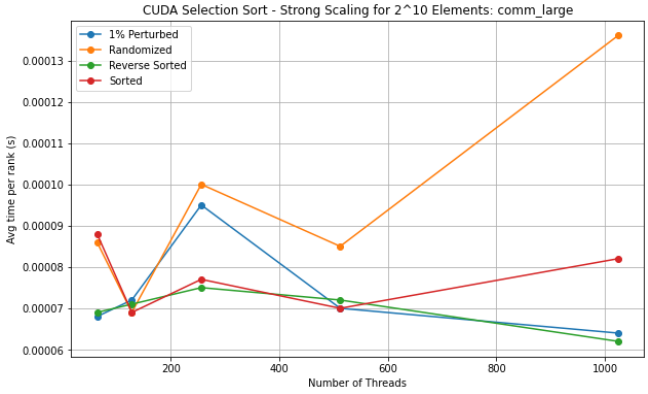
The following graphs are the strong scaling for CUDA.

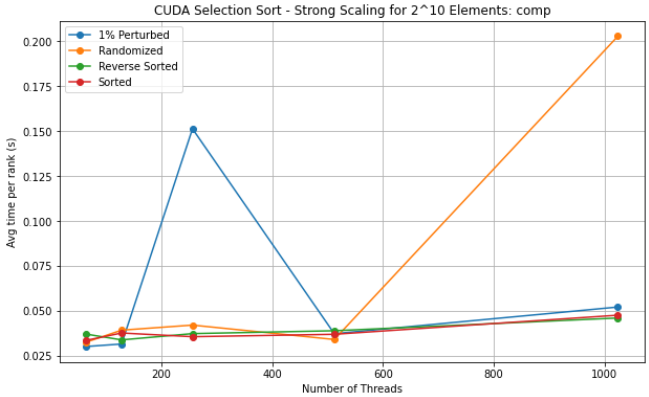
For main, a general decrease in time is shown, however around 512 threads the arrays had an increase in time for the inputs. The trend for the computation graphs was a general decrease until 512 threads with spikes in the middle for some of the inputs, however, at 512 threads there was an increase in the average time. This could be because of overhead. After a certain number of processors, increasing processors does not decrease time but instead takes more time. The same trend repeated for correctness check, comm, and data init graphs, as there was an increase in time starting around 512 threads. There were outliers throughout the graphs, however, with multiple runs of the script an average can be found. The randomized and reverse sorted arrays were doing worse than the sorted array in terms of time over the various input sizes. The reason is likely to be that they require more sorting to take place as the array is not in order.

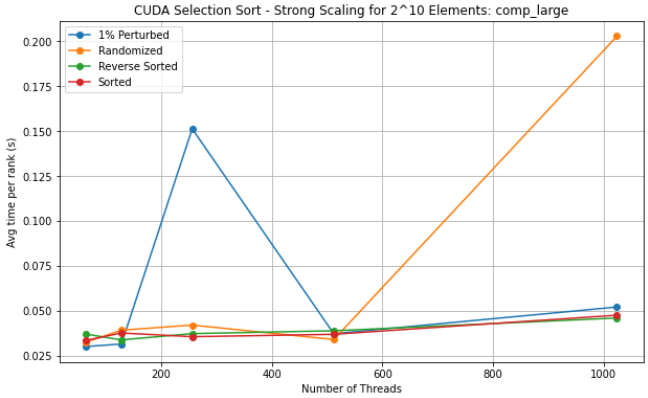
2^10 Elements

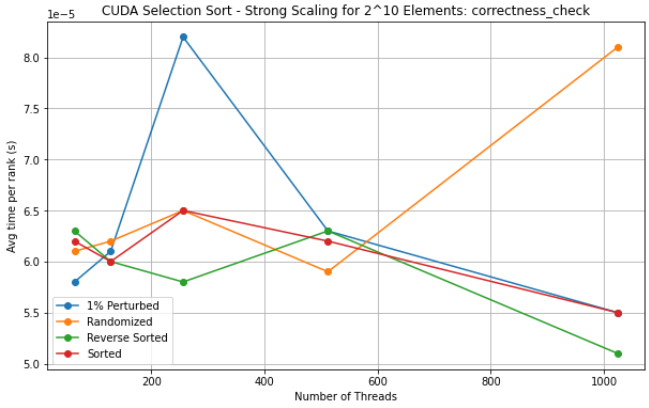


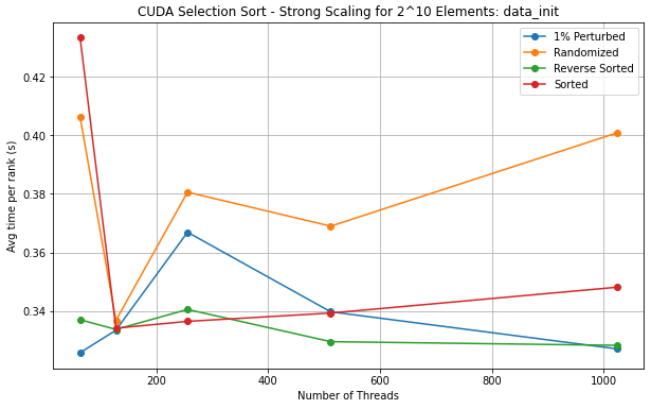




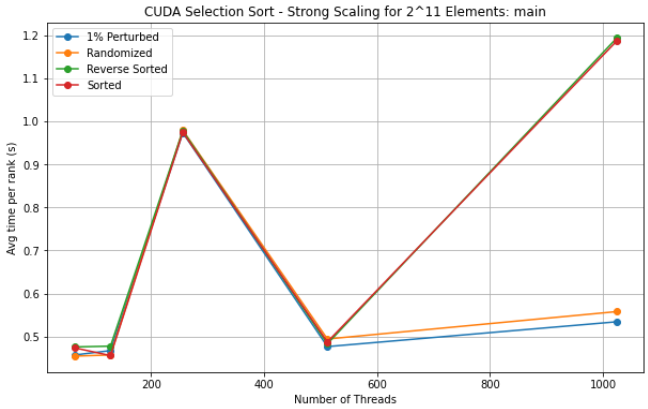


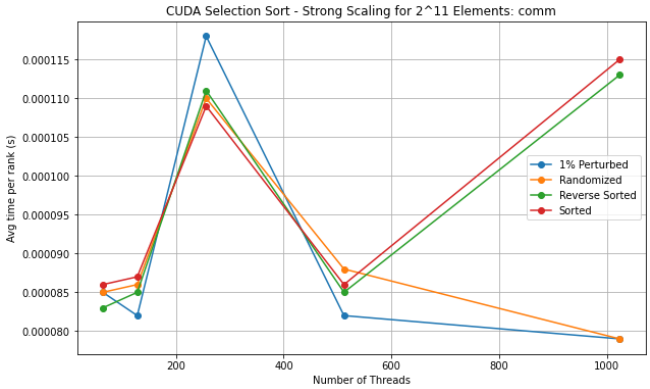


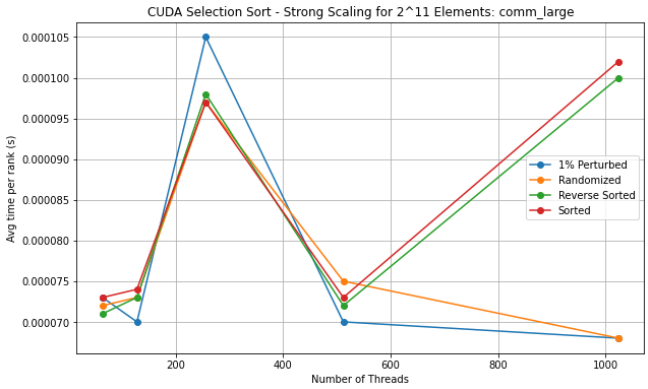


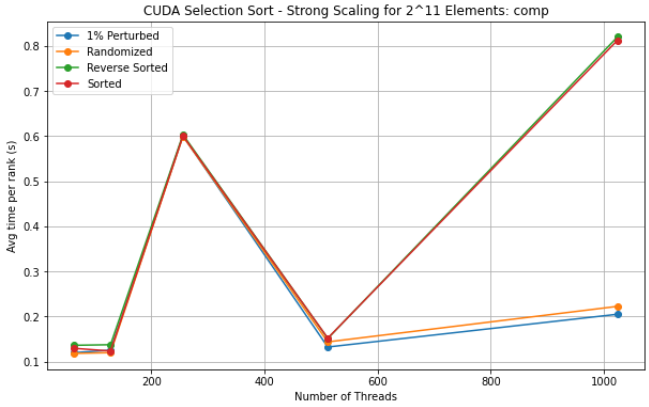


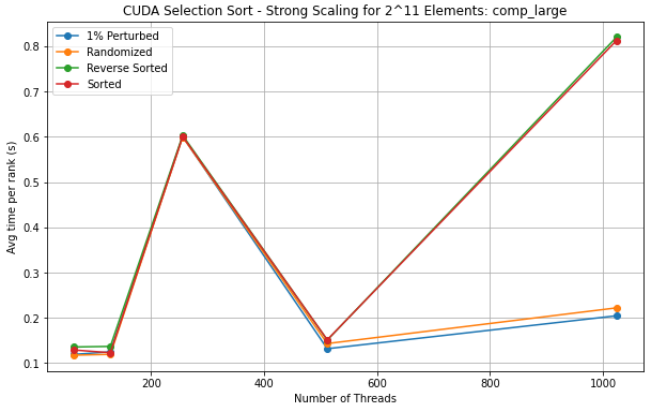
2^11 Elements

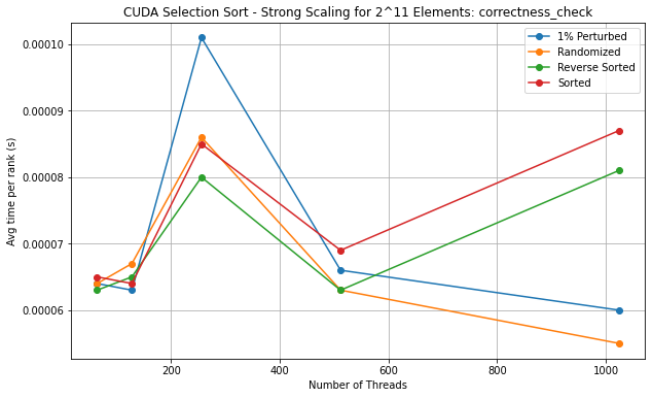


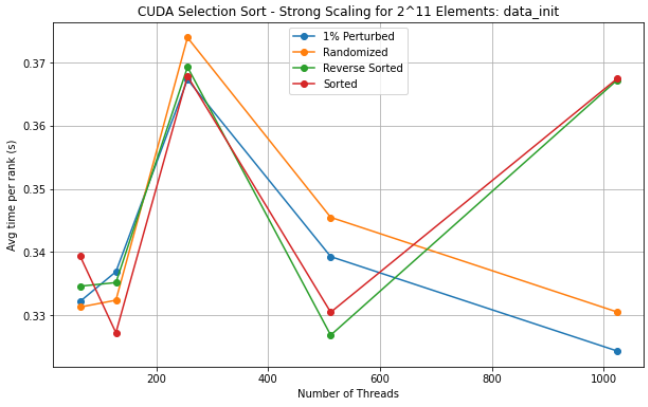




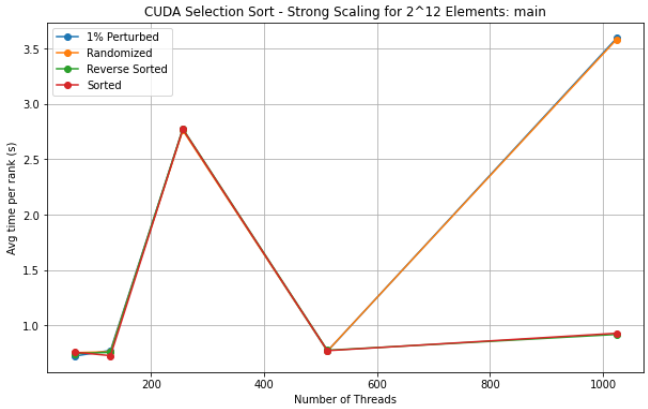


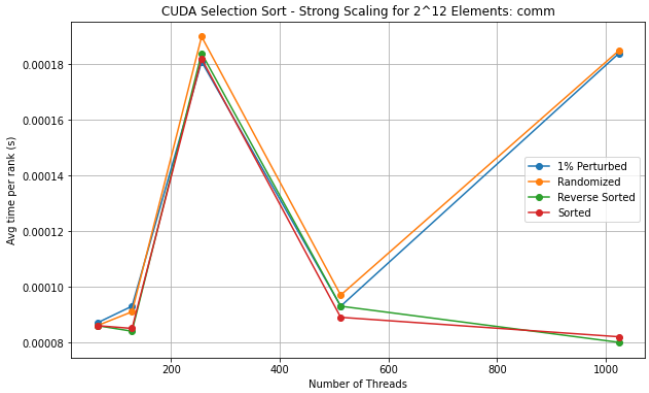


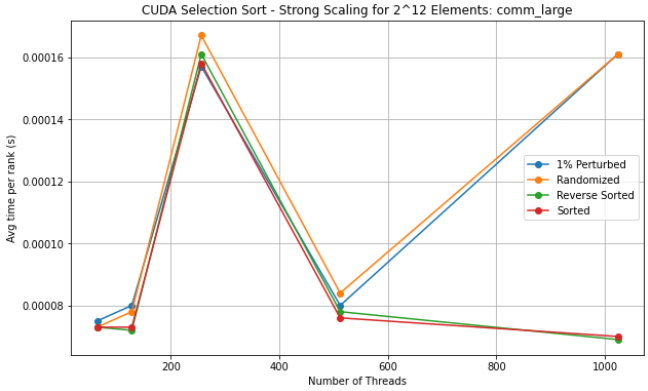


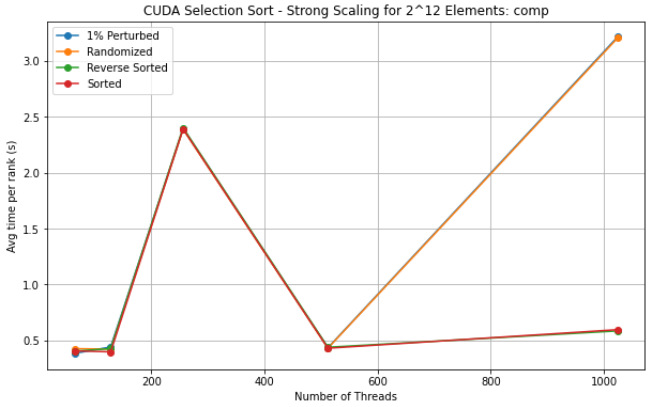


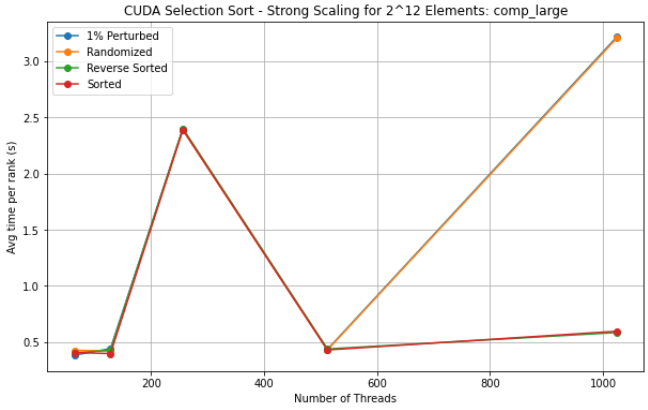
2^12 Elements

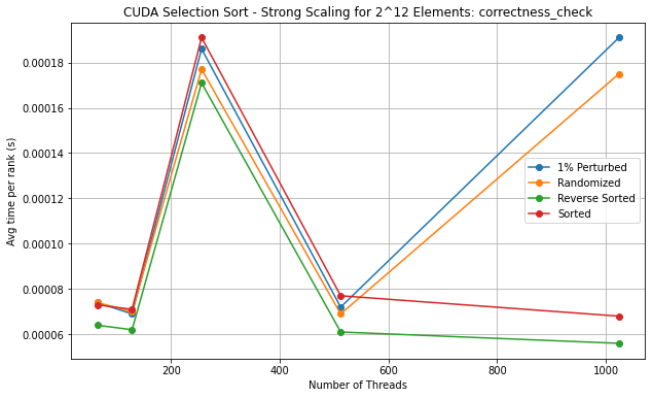


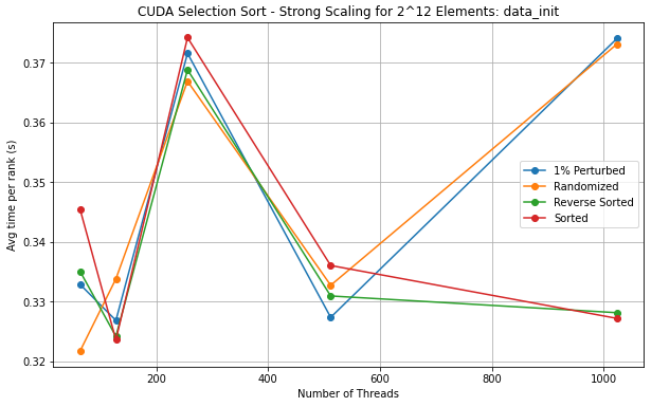




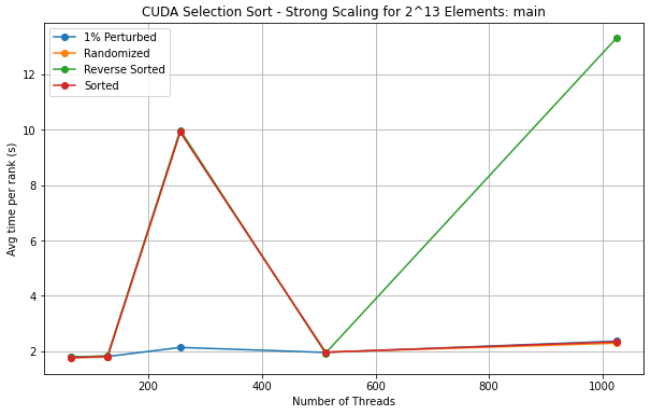


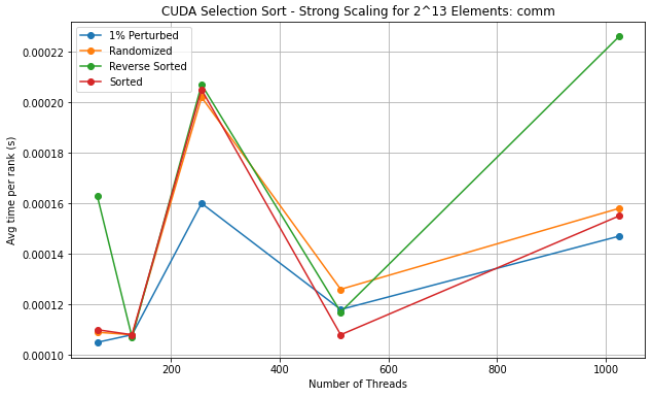


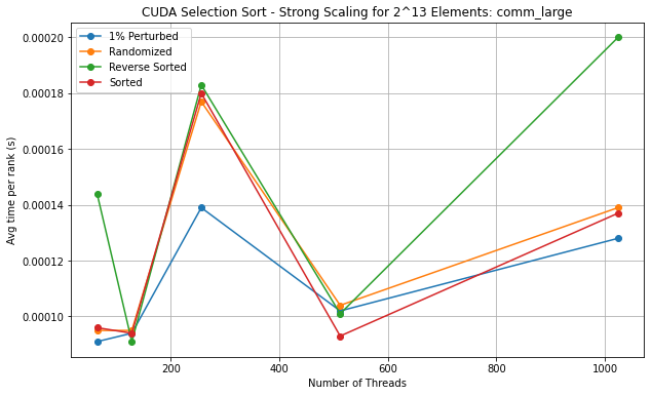


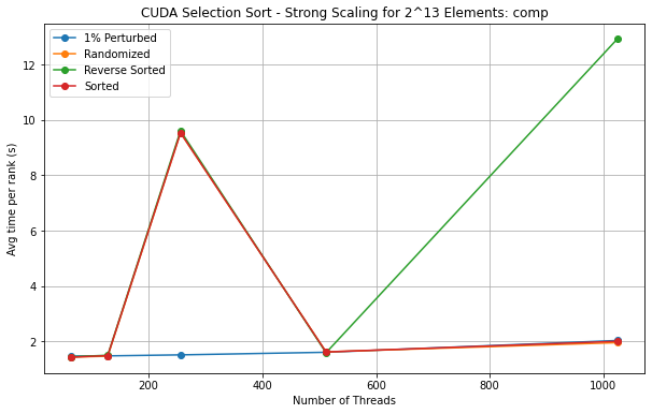


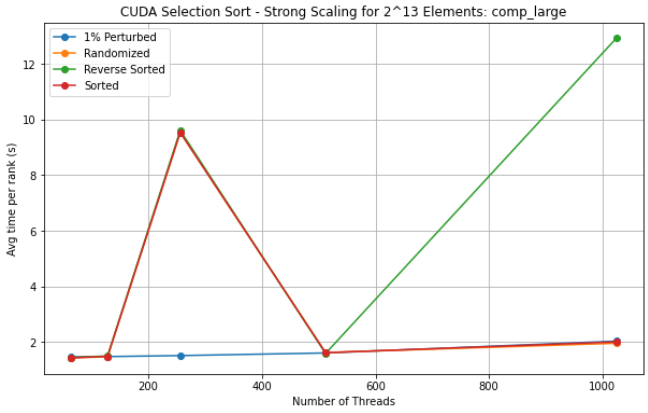
2^13 Elements

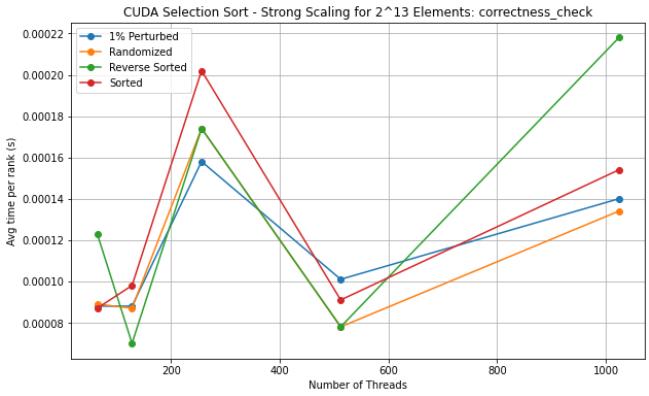


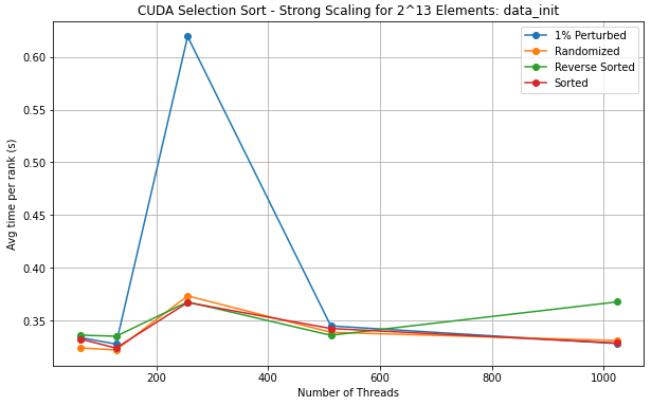




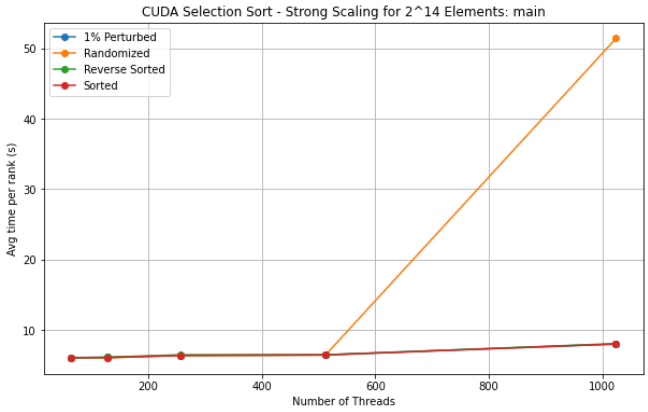


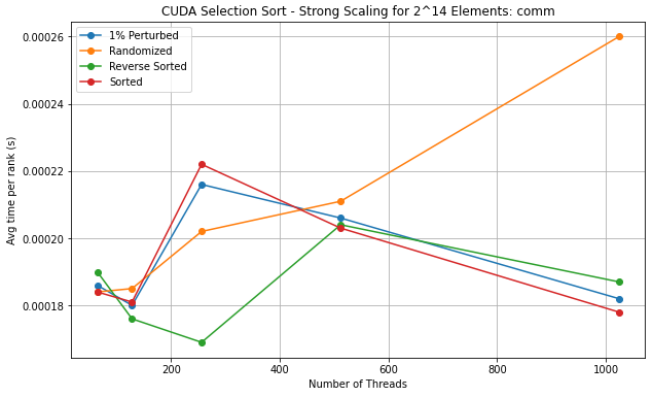


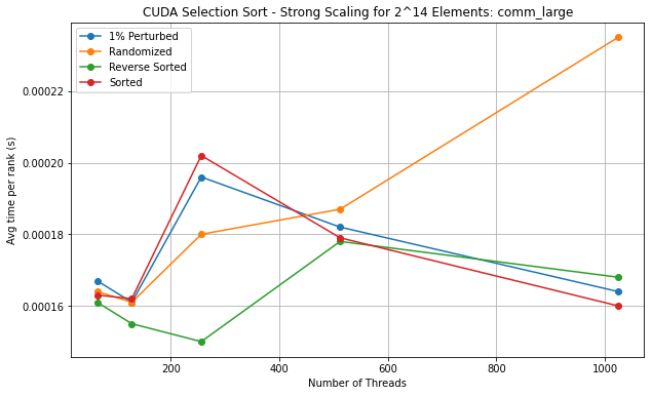


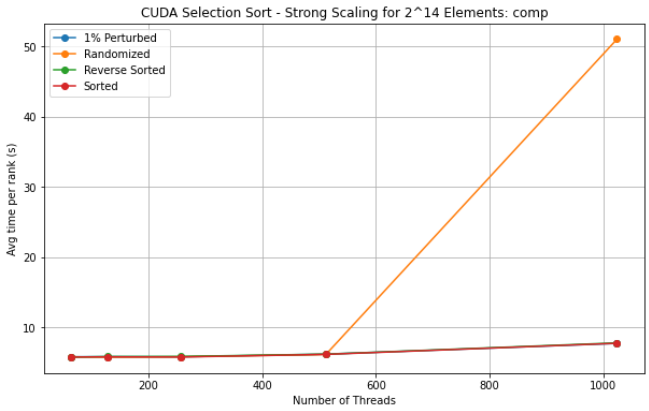


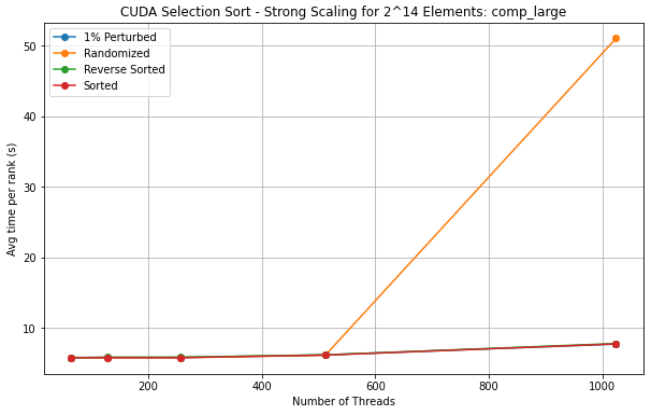
2^14 Elements

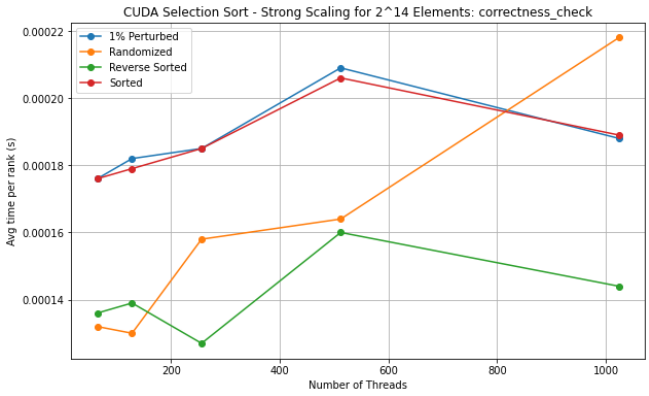


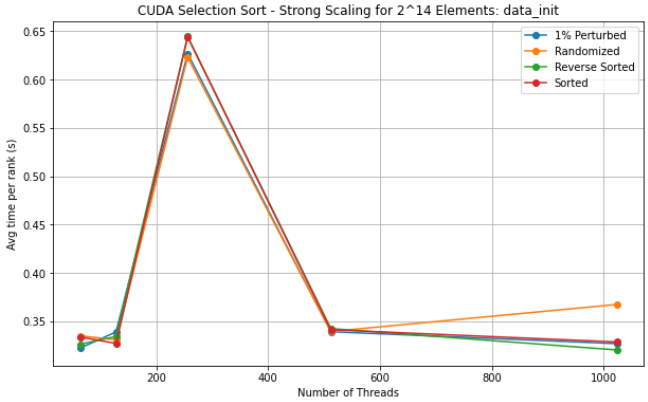




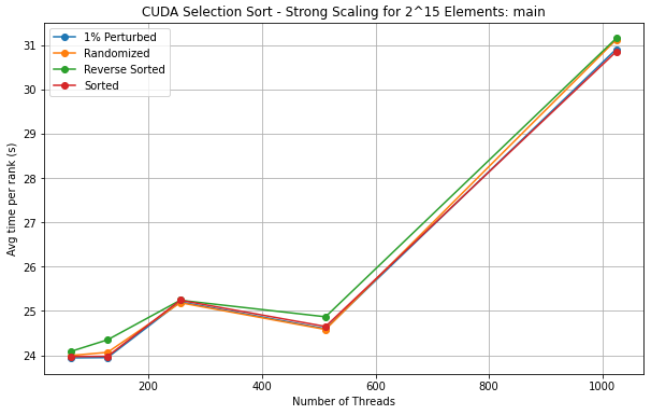


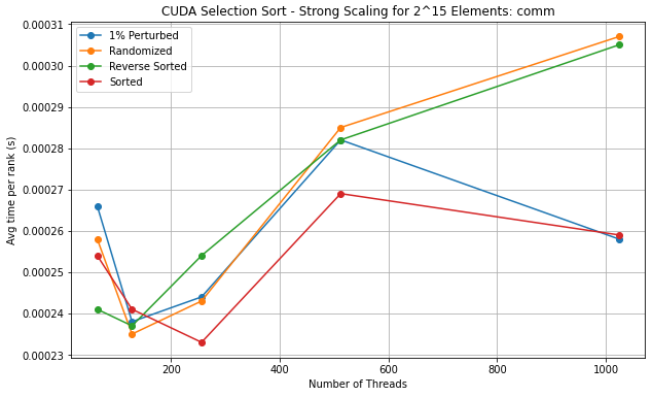


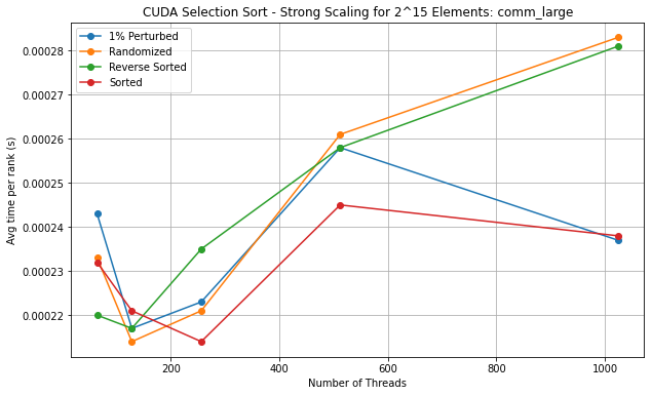


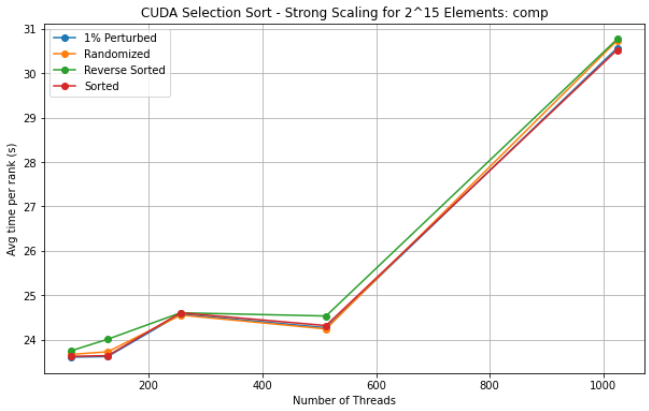


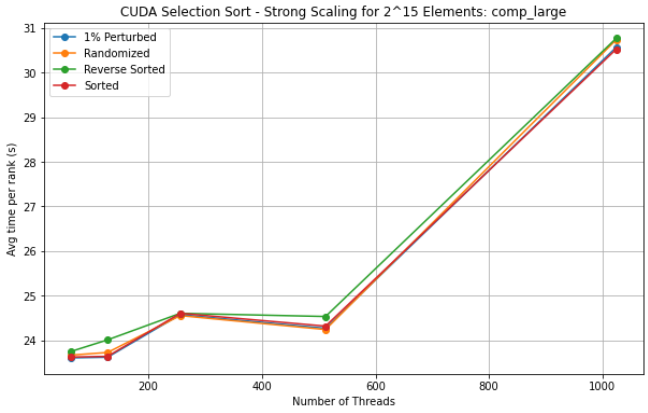
2^15 Elements

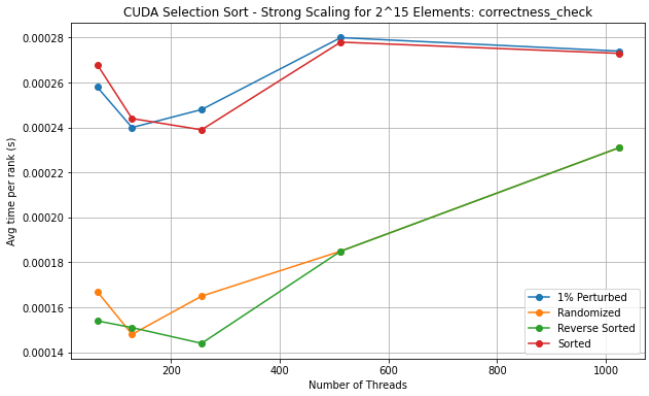


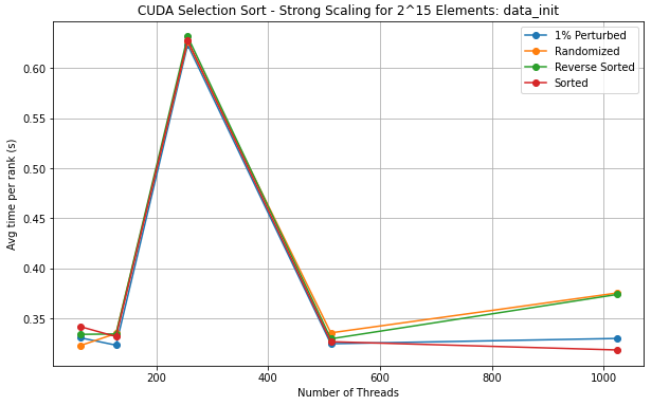




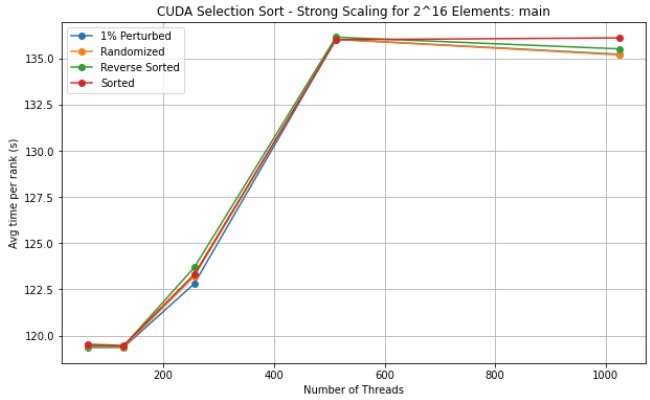


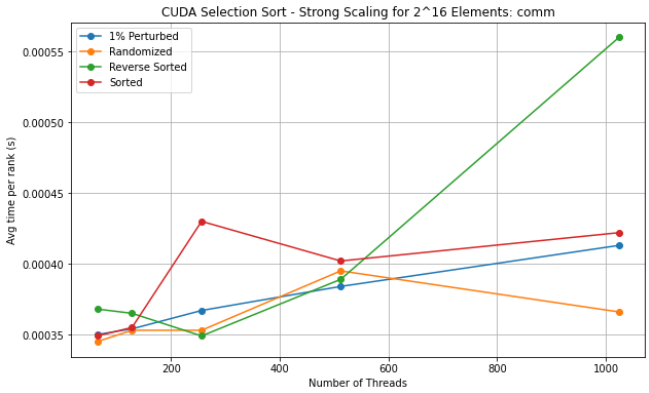


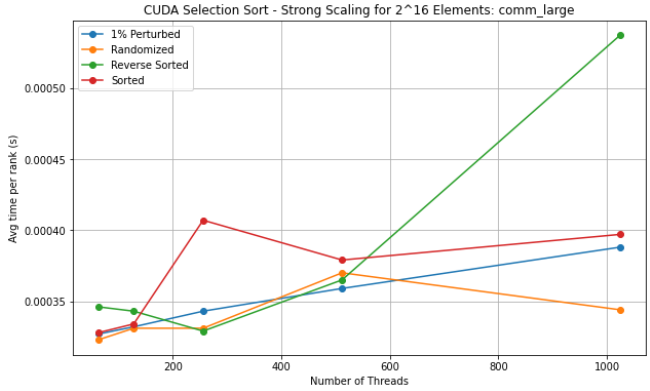


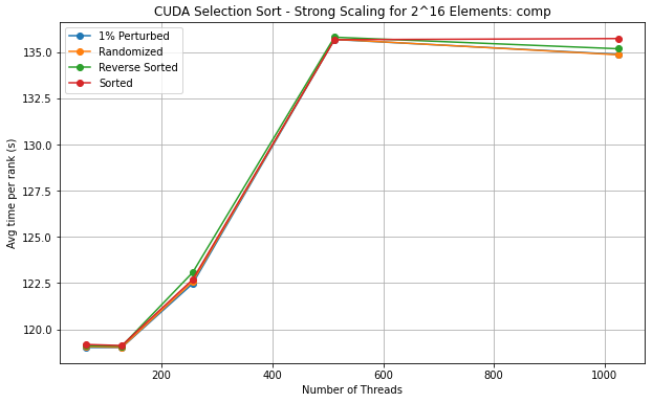


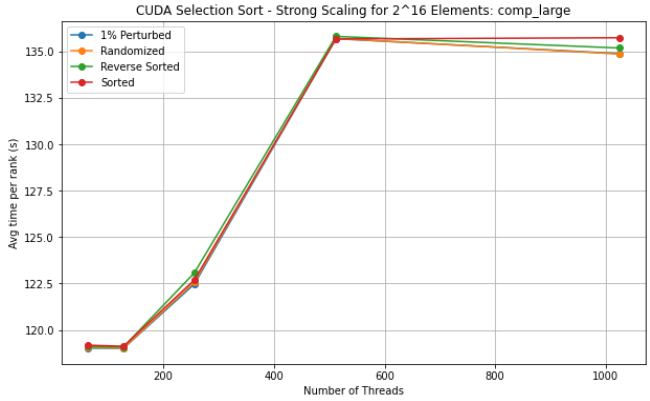
2^16 Elements

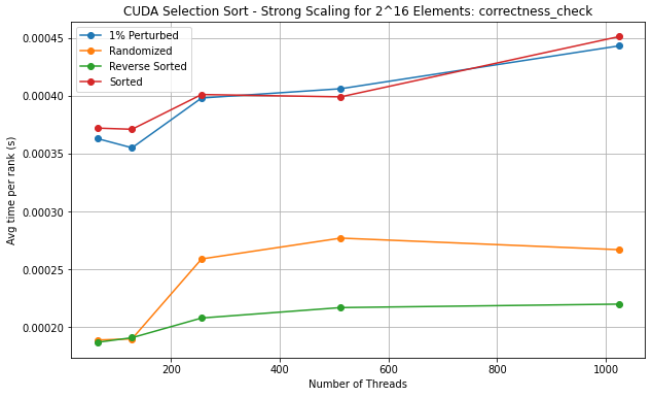


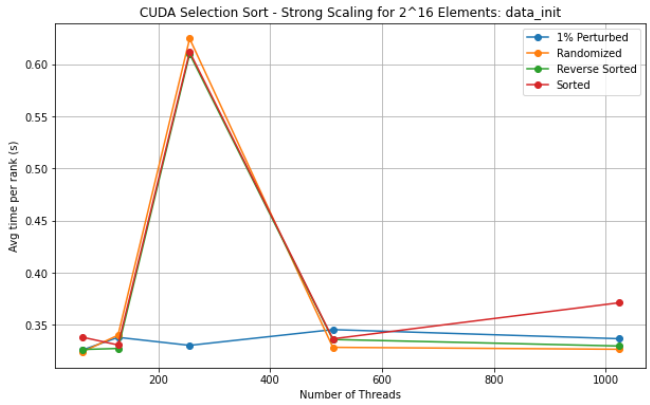












Speedup

MPI

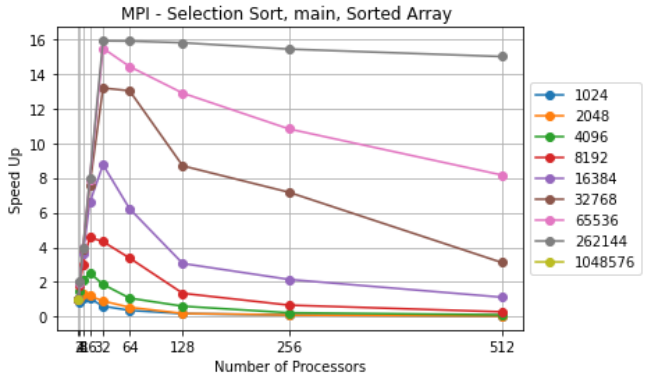
The following graphs are the speed up graphs for MPI.

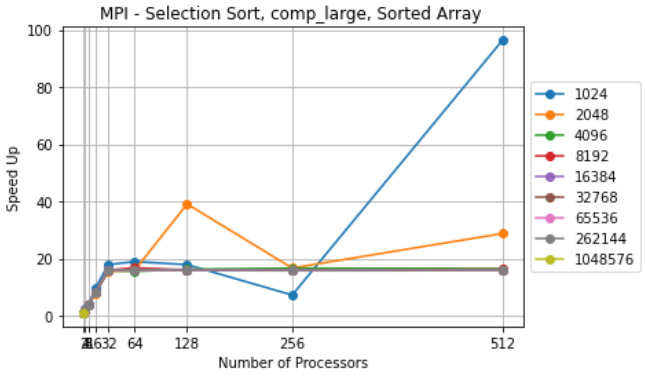
The general trend of the speed up graphs for main was an increase until around 32 processors. After 32 processors, there was a decrease in speed up. This could be because the input sizes that were tested were

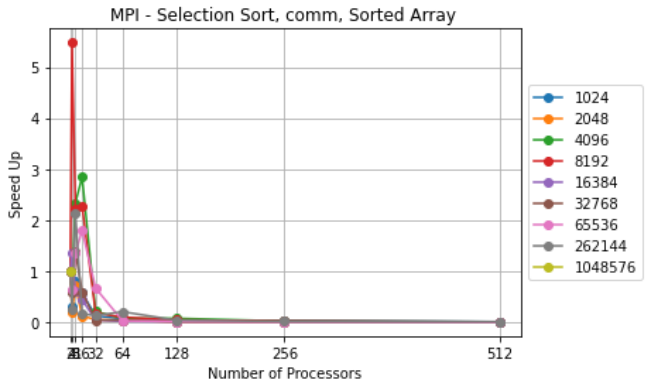
smaller, therefore 100's of processors are not necessary for a significant amount of speed up. The graph also demonstrates that the higher the input size, the higher the speed up increases at first. Although there is a slight decrease in speed up after 32 processors, it is not a decreasing spike but a steady decrease.

For the comp_large graphs, there is a general increase and then decrease in speedup until 128 processors on the various input types. However, as soon as the value is at 256 processors, there is an increase in speedup for the various input types. For the smalles input size 1024, there is a large increasing spike in speedup. For the comm graphs, there is a general decrease after a slight increase up to 16-32 processors. The graphs has a sharp decline and then flattens out for the various input sizes.

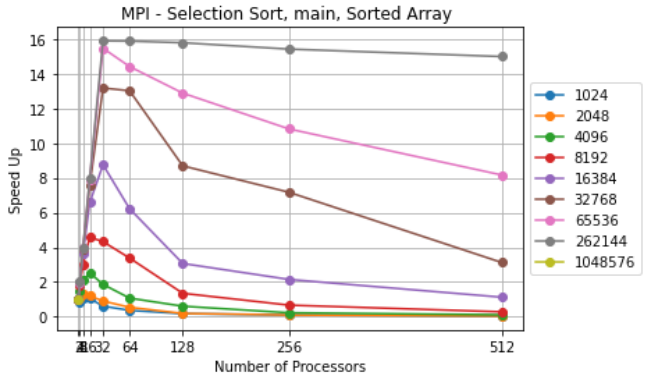
RANDOM INPUT ARRAY

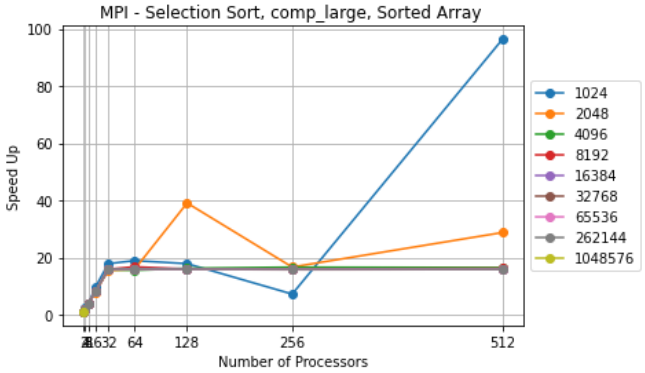


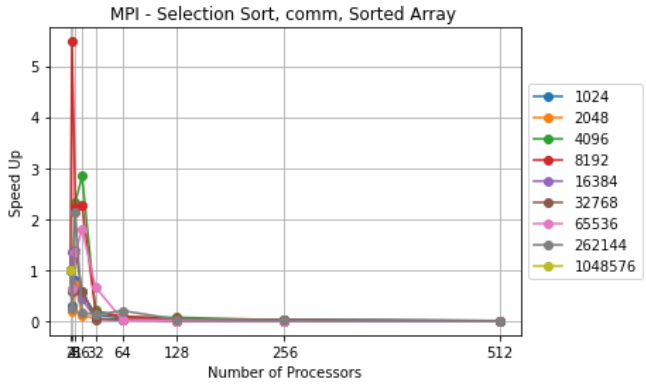




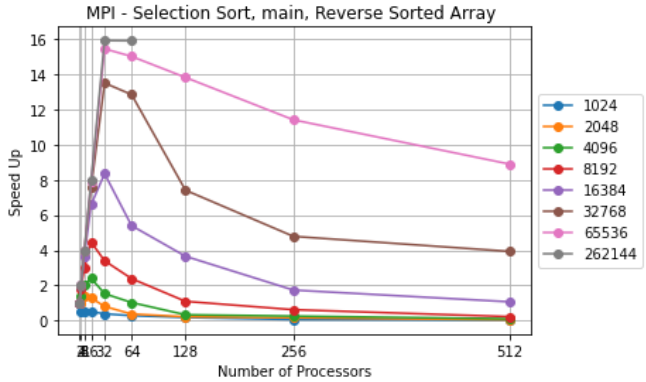
SORTED INPUT ARRAY

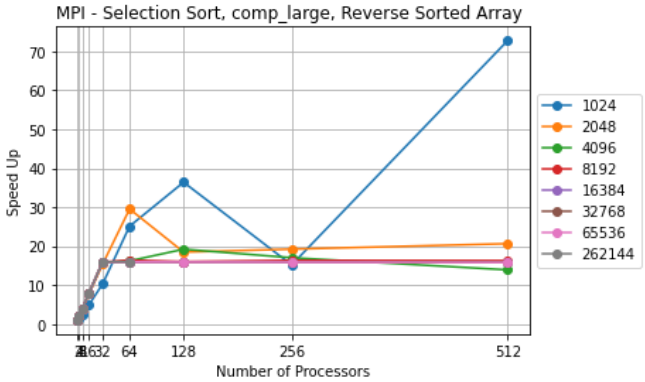


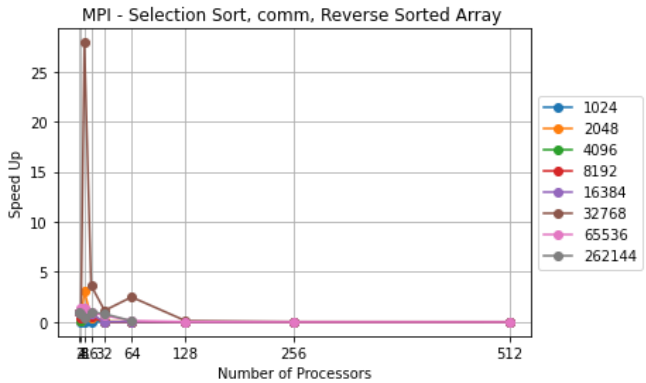




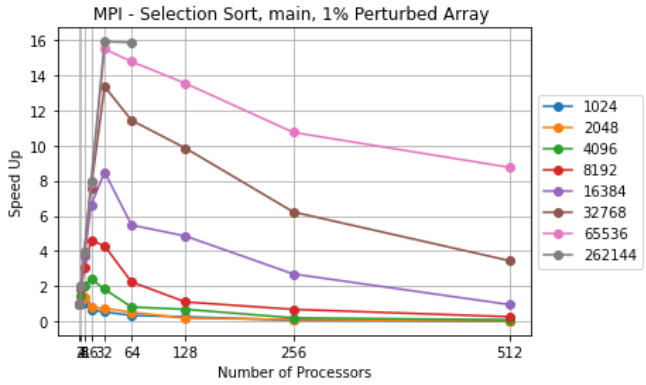
REVERSE SORTED INPUT ARRAY

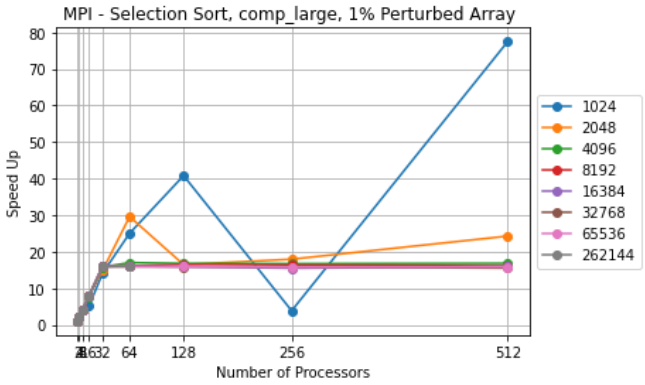


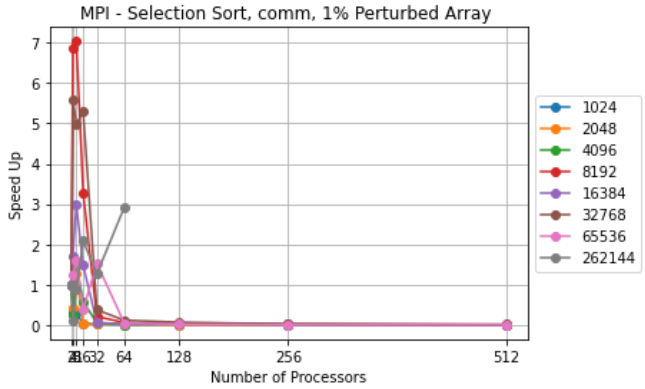




1% PERTURBED INPUT ARRAY





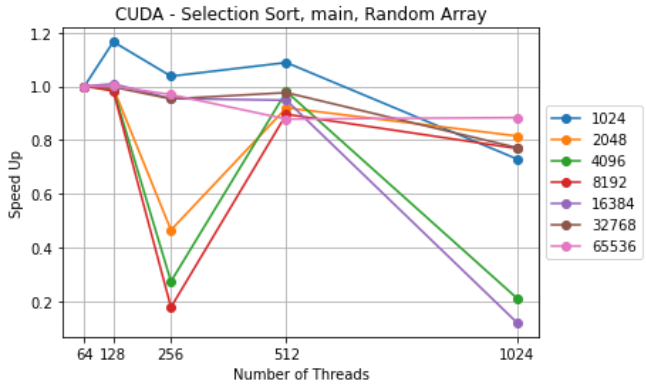


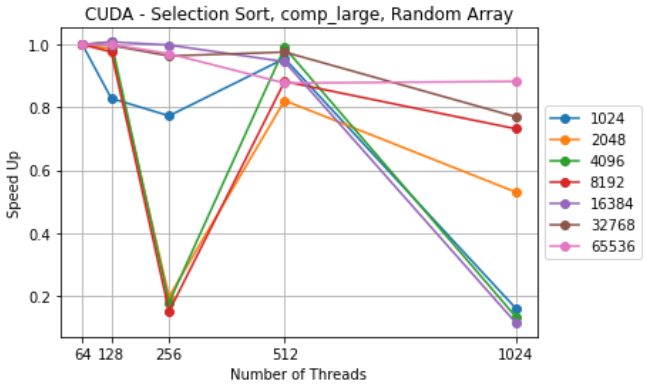
CUDA

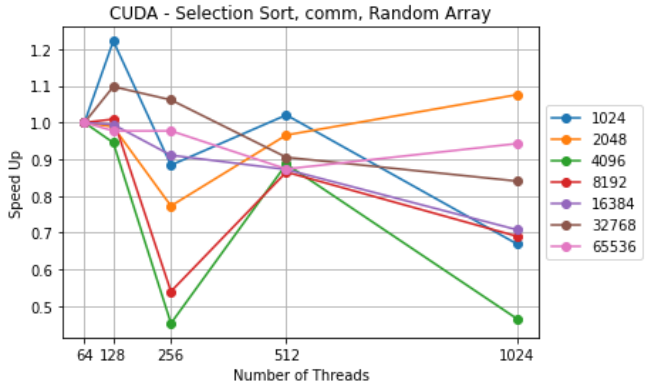
The following graphs are the speed up graphs for CUDA.

There is a general decrese across the graphs until around 256 threads. Then there is a general increase in all the input lines until 512 threads. Finally, there is a decrease in all the threads after 512 threads. For comm, however, the same trend was followed, but for the 1% perturbed array some of the input sizes had an increase from 512 threads. The input sizes were 2^10, 2^11, 2^14, and 2^15 had an increase. There were a few outliers in the graphs. For example, for the comm of the reverse sorted array. 2^13 input size had an increase from 64 to 128 in speedup. However, one average for all of the graphs, the general trend of increase and decrease mentioned above was followed.

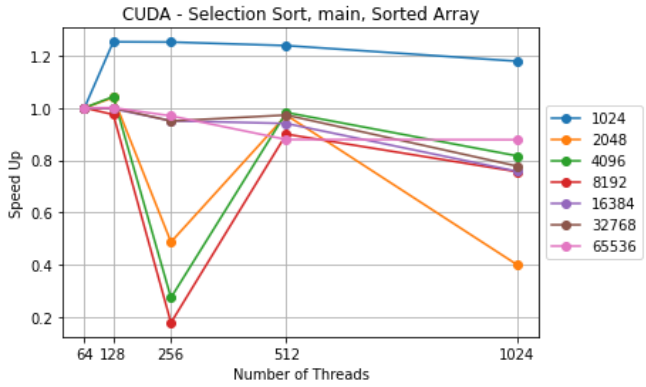
RANDOM INPUT ARRAY

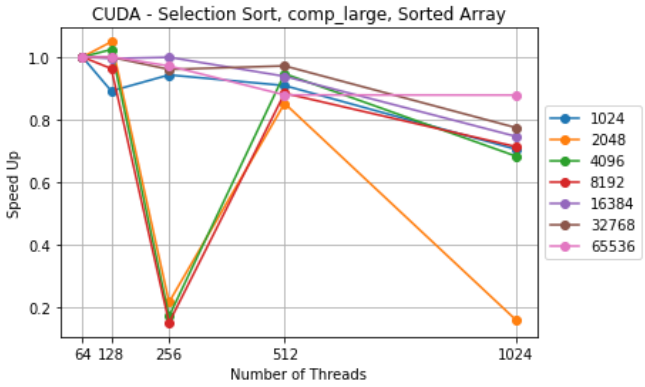


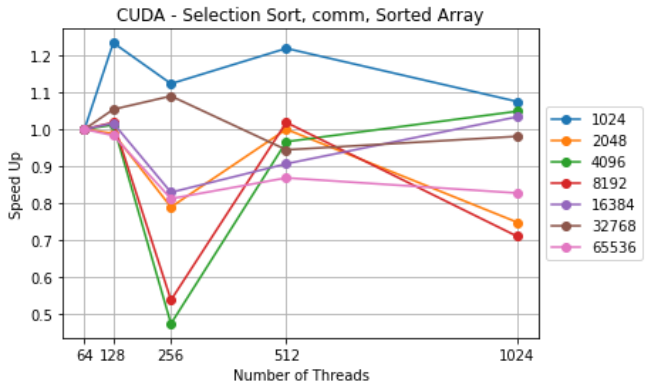




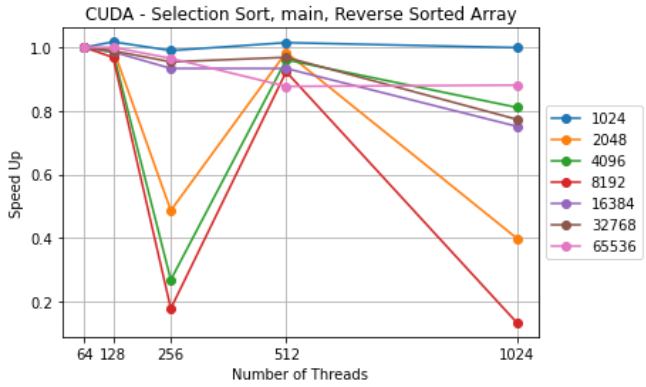
SORTED INPUT ARRAY

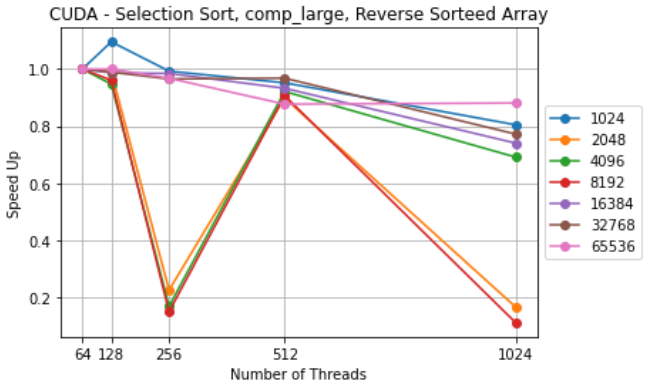


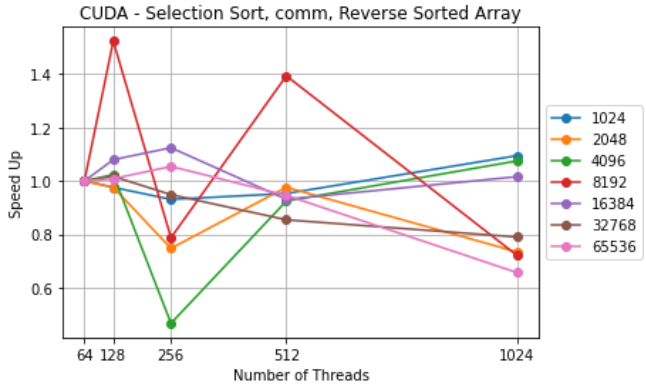




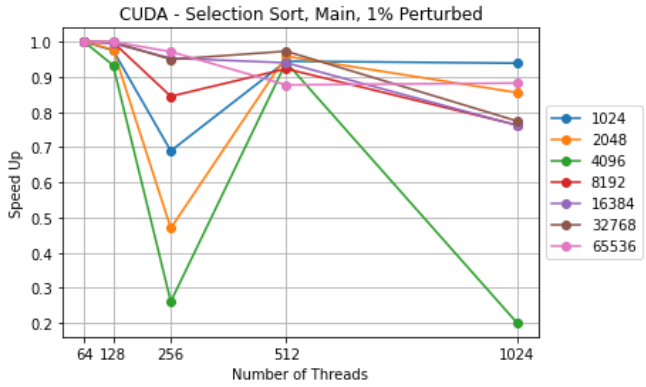
REVERSE SORTED INPUT ARRAY

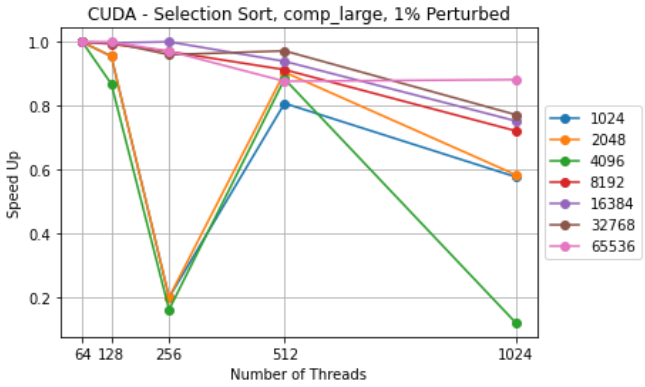


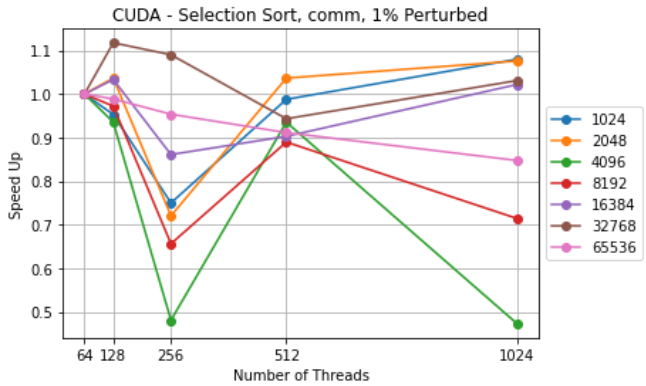




1% PERTURBED INPUT ARRAY





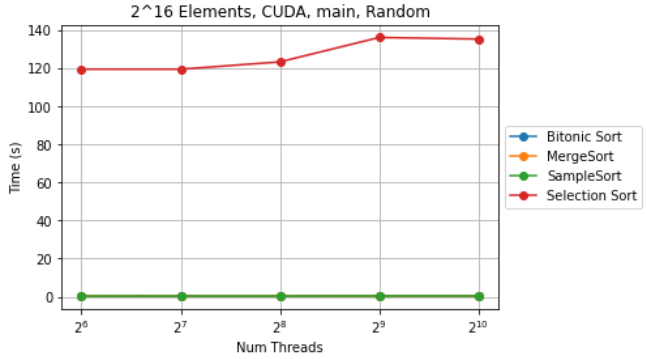


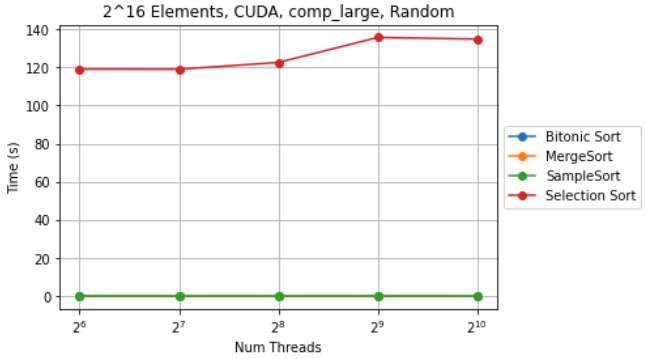
Comparison Analysis

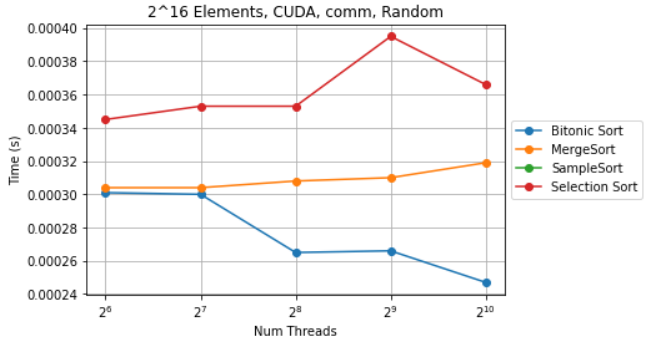
For the comparison analysis, we analyzed an input size of 2^16 elements since selection sort was having issues at larger sizes. Starting with CUDA, we noticed that selection sort was much slower for all aspects of total (main), communication, and computation time. To conduct a more thorough analysis, we looked at both

the graphs with all of the sorting algorithms, as well as graphs that excluded selection sort. This was needed as selection sort's time was on such a large scale that it would wash out any discernable graph shape for the other three sorts making it impossible to tell which ones performed better. We can see the main and comp graphs are completely overtaken by the selection sort because it is not inherently a parallel algorithm and therefore performs the worst by far. A good comparison with selection sort is easiest to see in the communication graph since this was mainly cudaMemcpy and is relatively quick. Sample sort did not include a comm region, and therefore is not present on the graph.

CUDA - All

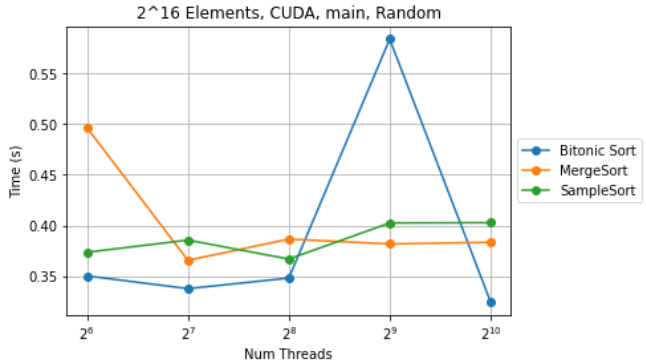


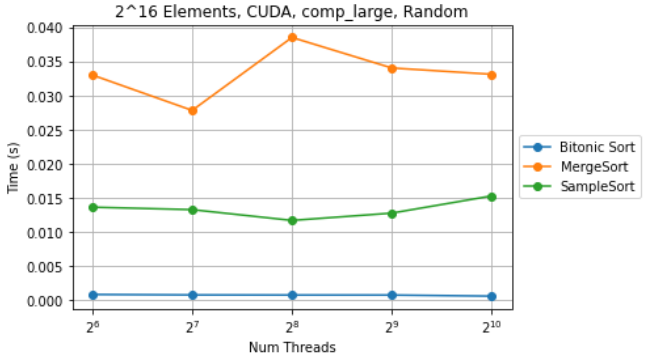




CUDA - Bitonic, Merge, and Sample Sort

When looking at the main and computation graphs without selection sort, we see they are relatively constant, despite the rises and falls in the graphs, as the y-axis scale is to the order of tenths of a second. We see a spike for bitonic sort at 512 threads in the main region, which could likely be an outlier considering its other data points. When looking at just computation, we see that it remains fairly constant for all three graphs. From this analysis, we can conclude that for CUDA, the order from best to worst scaling sorting algorithms for our implementations is bitonic, sample, merge, and then selection sort.





MPI

The graph of main shows mostly similar behavior across algorithms outside of merge sort, with the other three sorts hovering well below one for most number processes. Sample sort has a spike in the comm time at higher process counts, which explains the upward jump on the far right. Merge sort's graph shape for main isn't explained by either comp or comm, so other areas such as data initialization likely played a role in its poor performance. Looking at the computation graph, we see that selection sort is by far the worst at lower process counts, which indicates it is a poor-performing algorithm. However, it quickly converges towards 0 starting around 32 processes so it seems to take better advantage of more resources. It is difficult to see on the comp graph, but merge sort is the best performing of our sorting algorithm in MPI. When looking at the comm graph, we see most of the sorting algorithms slightly increase as more processes are added, except for sample sort which jumps up at 1024 processes, as previously discussed, due to the communication overhead of sending both the data and samples between processes.