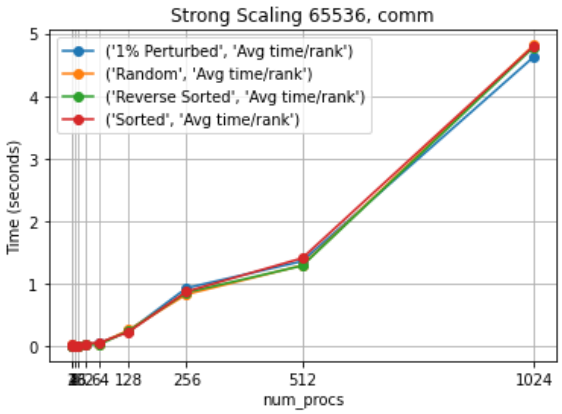
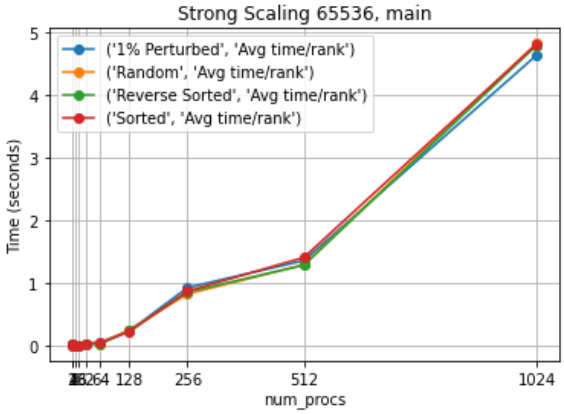
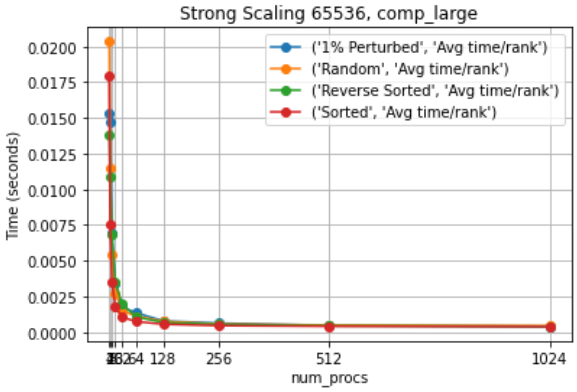
Sample Sort MPI Graphs

Strong Scaling Plots

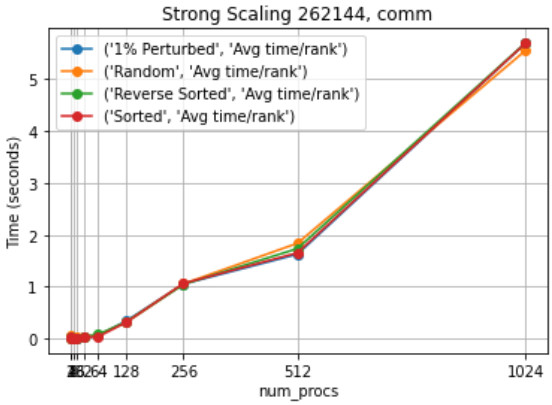
Input Size 65536

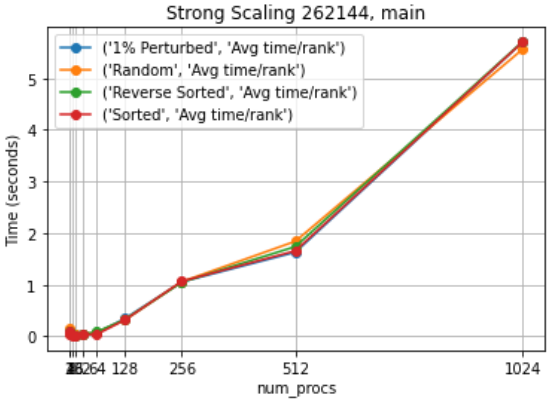


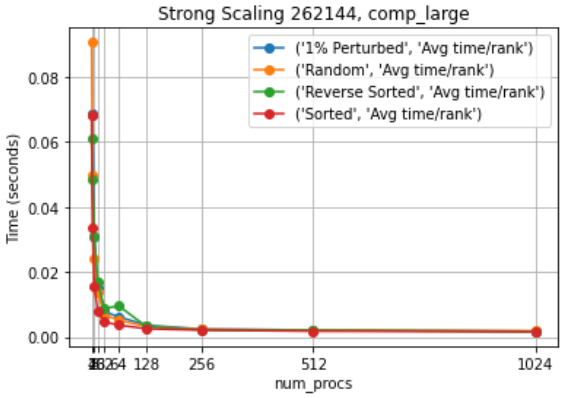




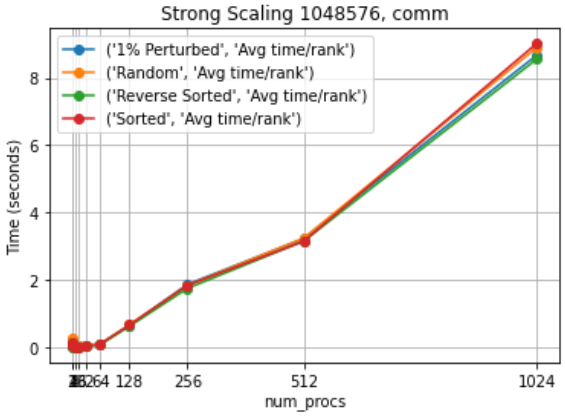
Input Size 262144

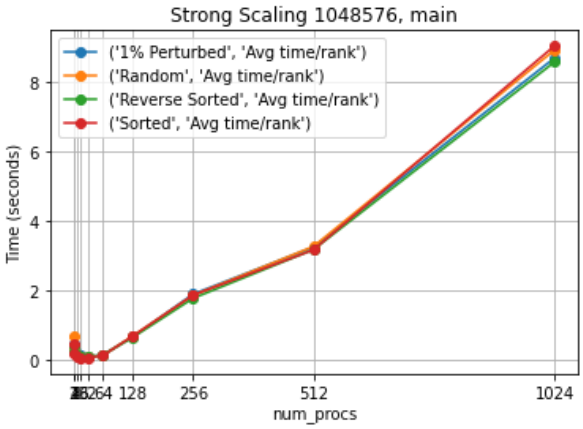


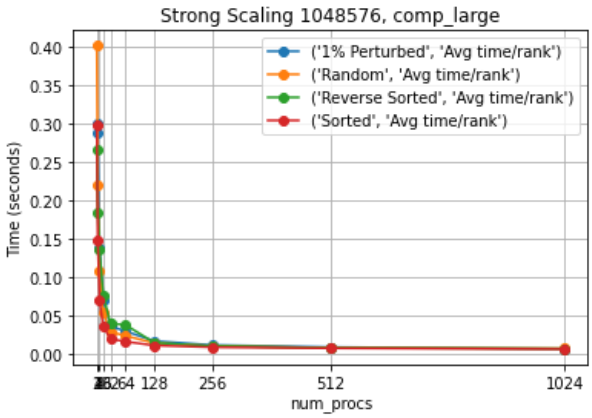




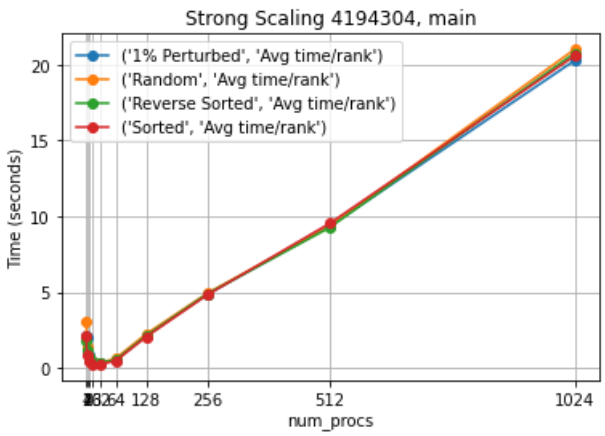
Input Size 1048576

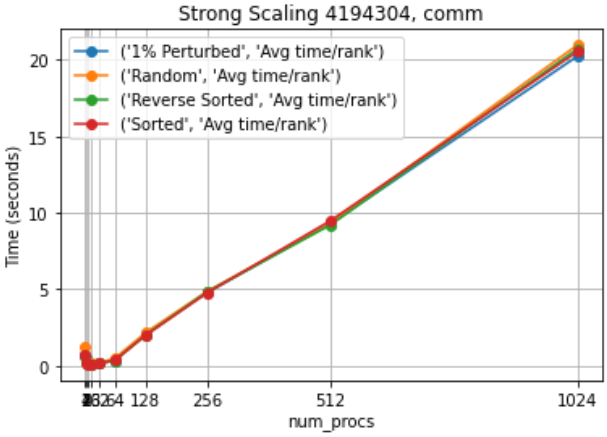


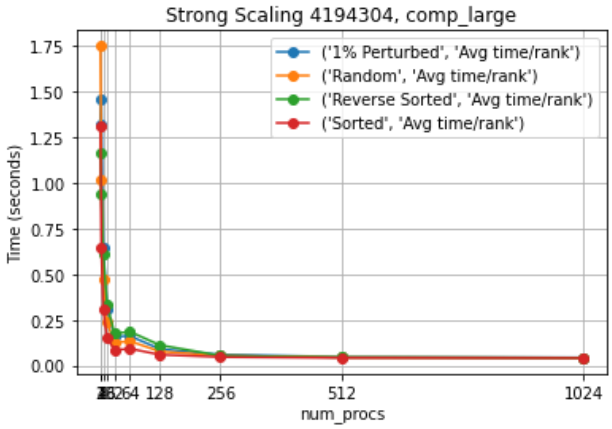




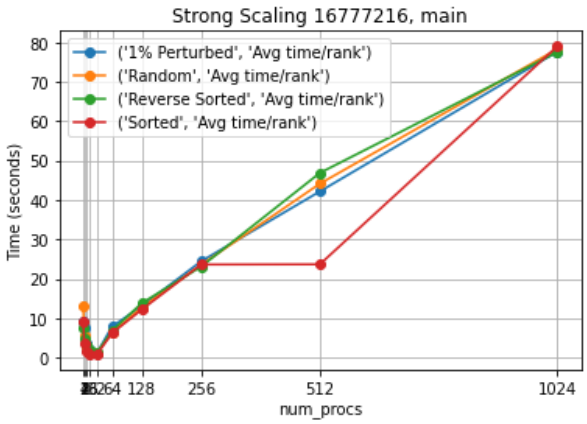
Input Size 4194304

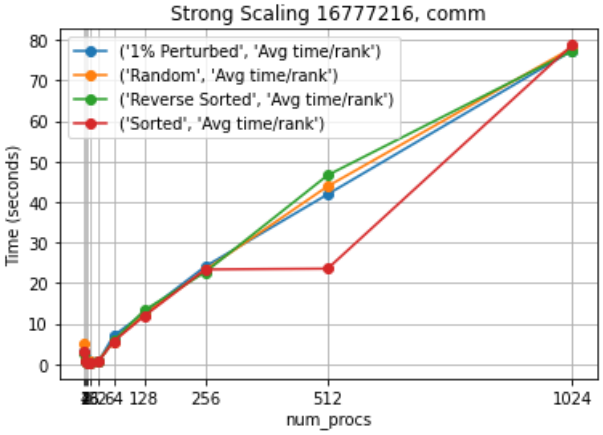


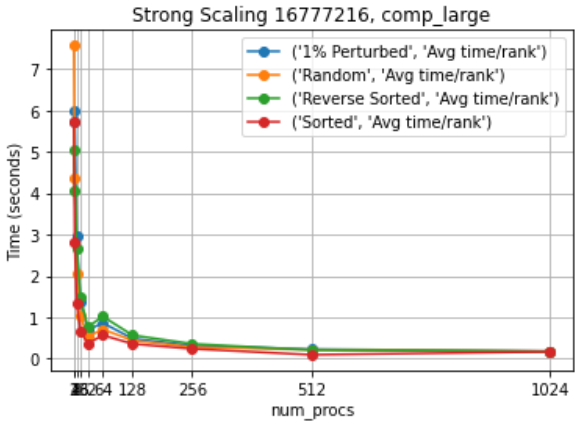




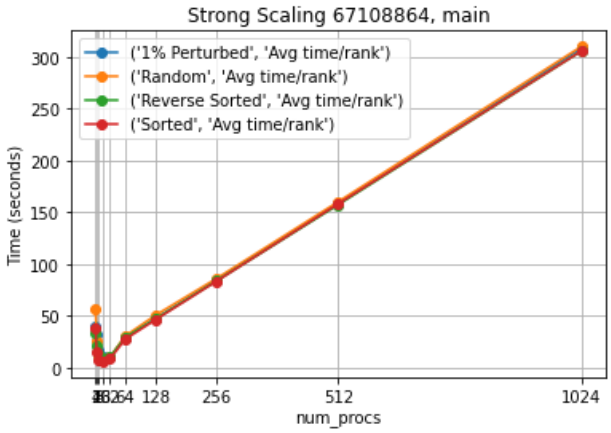
Input Size 16777216

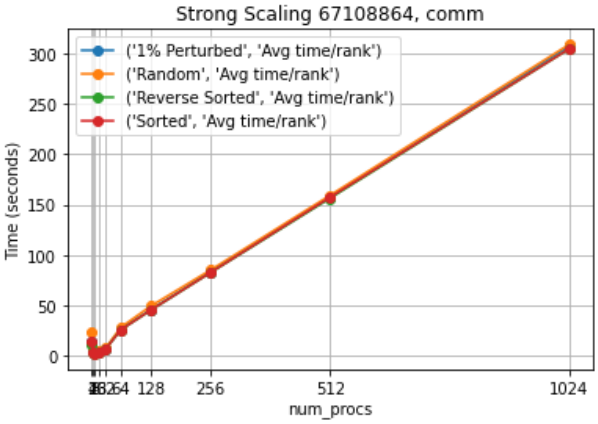


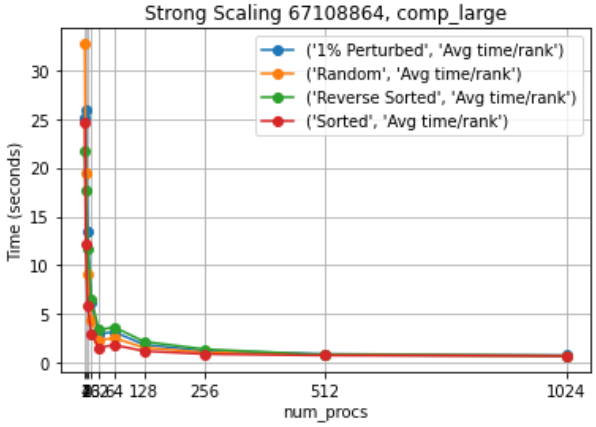




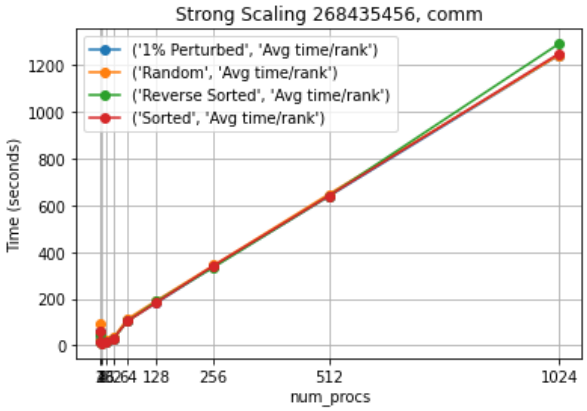
Input Size 67108864

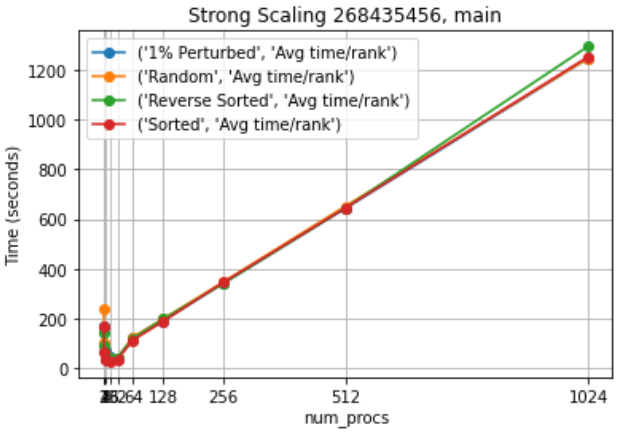


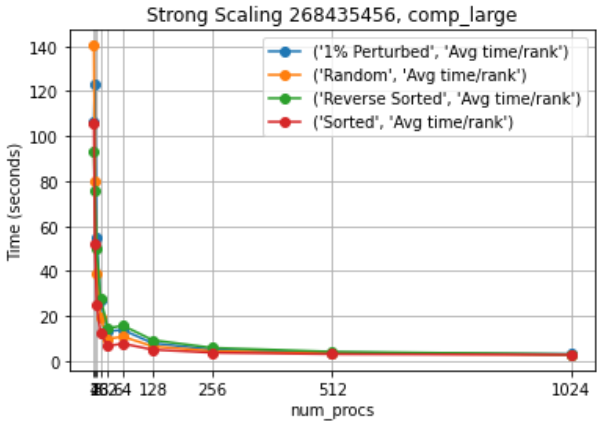




Input Size 268435456







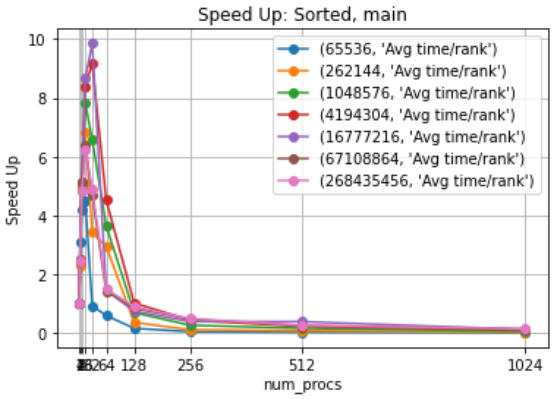
Analysis

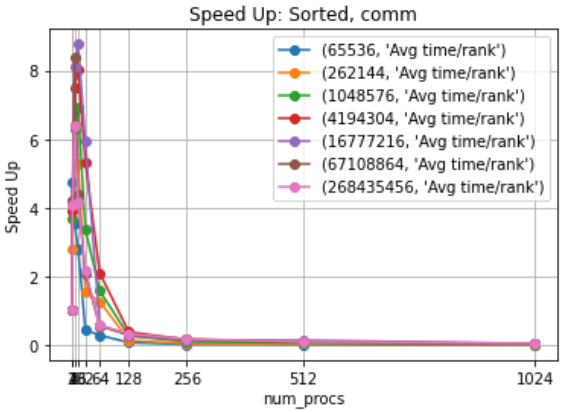
My MPI implementation for sample sort did not scale well with strong scaling. As can be seen, for every input size, the runtime for the overall program (main) increased as the number of processors increased; this trend is the same regardless of different input types. We see that computation time did improve as the number of processors increased, though the benefits of parallelization plateaus after 128 processors. Though computation did scale well, the overall implementation did not because of communication time. As can be seen with the communication strong scaling plots, the runtime for communication increases linearly with the number of processors. The reason that runtime increases in direct proportion to the number of processors is because the master process sends a 2d array to each worker process. The number of rows that this array has is equal to the number of worker processes. Because of this, more worker processors = more rows = larger array size. Not only is the array size larger, but it is also being sent to more worker processors. This is the reason that communication time increases in direct relation to the number of processors. Because of this, we can deduce that the bottleneck for this implementation is communication time. This is further supported by the fact that all of the strong scaling (main) graphs look almost identical to strong scaling (comm)

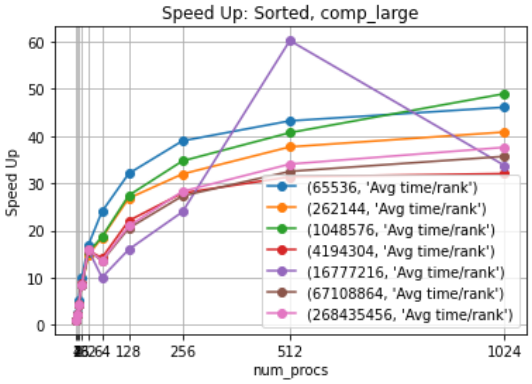
graphs. The benefits from computational parallelism is negligible in comparison to the amount of time communication takes.

Strong Scaling Speedup

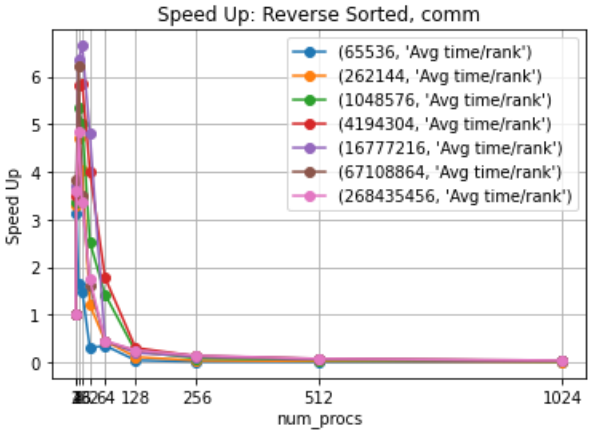
Input Type: Sorted

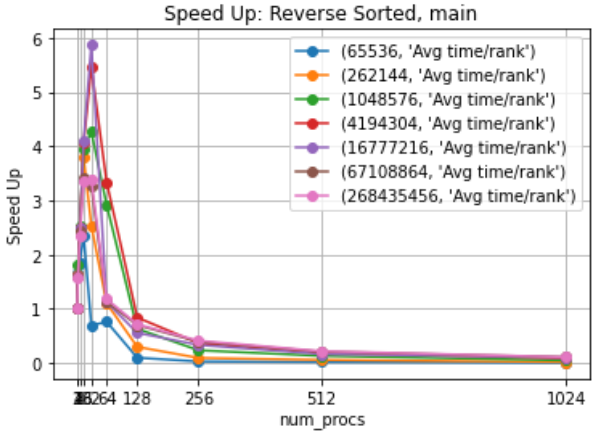


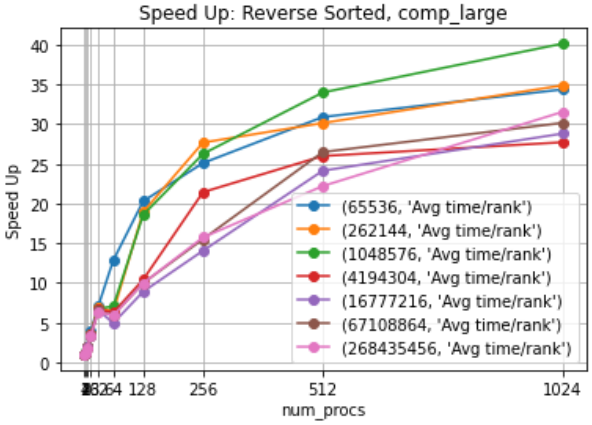




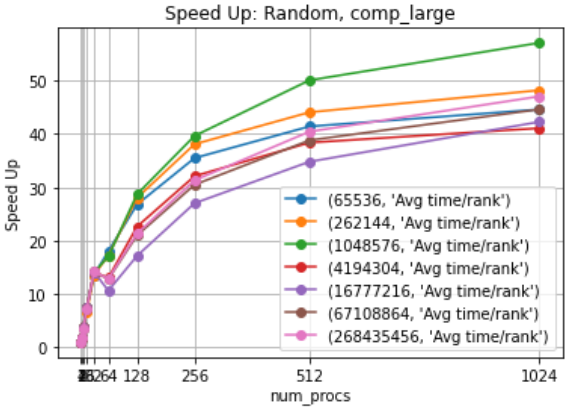
Input Type: Reverse Sorted

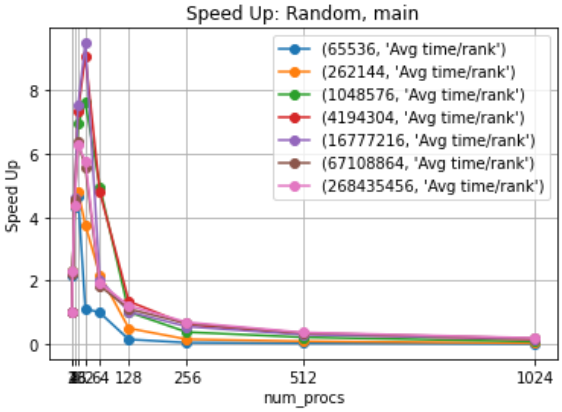


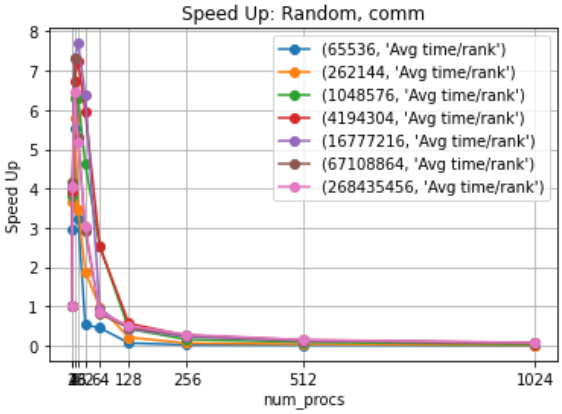




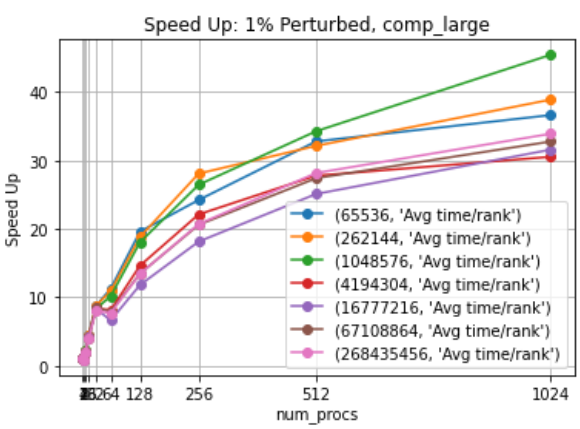
Input Type: Random

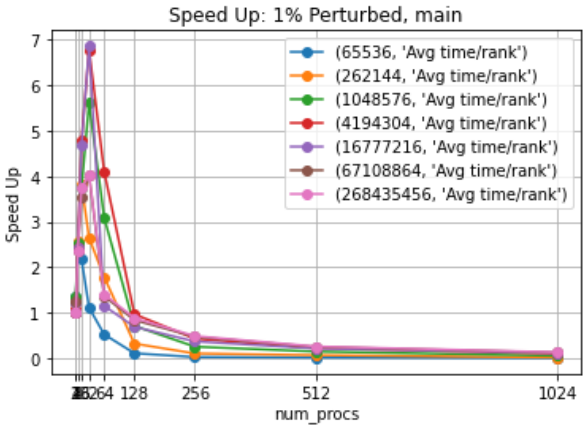


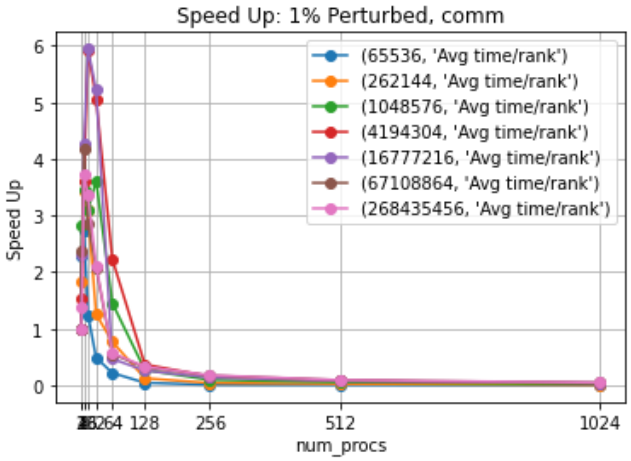




Input Type: 1% Perturbed





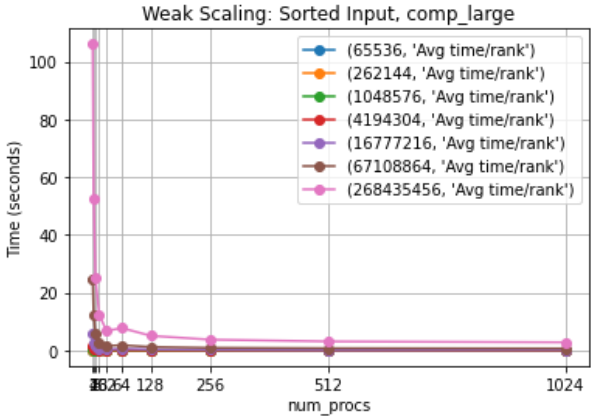


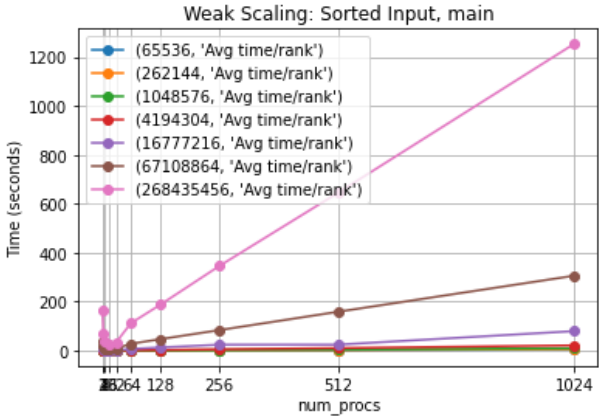
Analysis

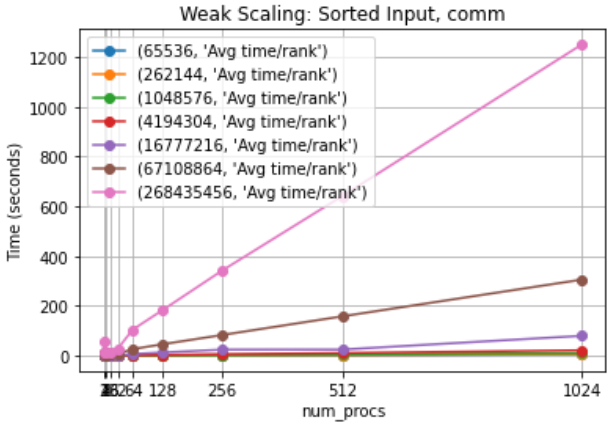
The trends seen with speed up matches perfectly with what was stated/analyzed in the strong scaling analysis. We see that for every input type, computation does see a speed up. We know that the parallelism in this implementation works well for communication because the trend for computation speed up remains the same across all the various input sizes. We see that speed up for communication was terrible. Speed up increased briefly, but once we surpassed 16 processors, the speed up plummeted. This matches up with our strong scaling graphs. If you refer to the strong scaling comm graphs, you will notice that though the runtime increased proportionally with the number of processors, there was a small point where runtime did not increase and instead decreased but a small amount. This point where we see this decrease matches perfectly with where we see an increase in speed up for comm. Similarly to the strong scaling graphs, we can deduce that the bottleneck for speed up here is the communication. Like the strong scaling graphs, the speed up for main follows the same trend as communication. These trends remain consistent among all the different input types.

Weak Scaling

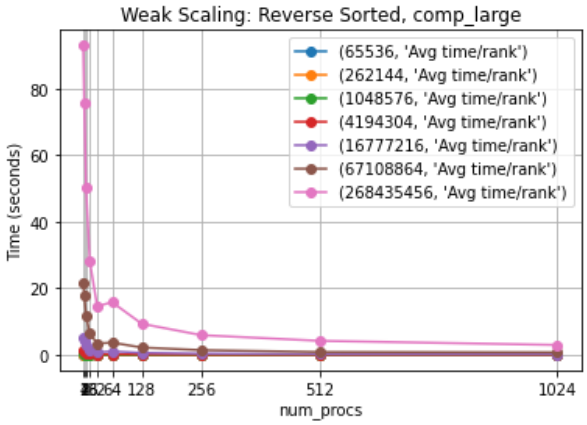
Input Type: Sorted

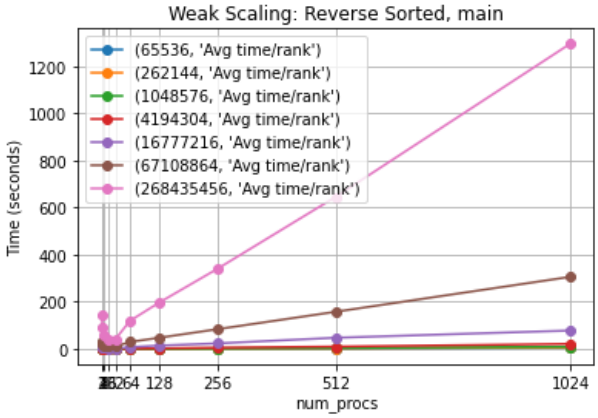


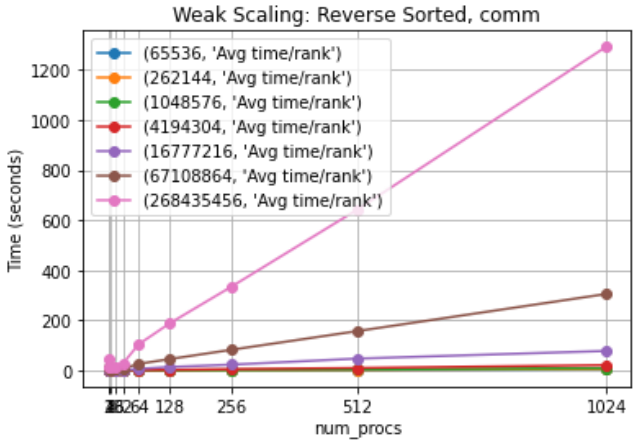




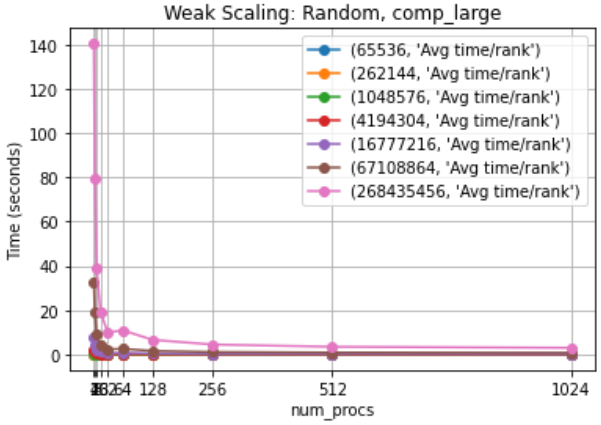
Input Type: Reverse Sorted

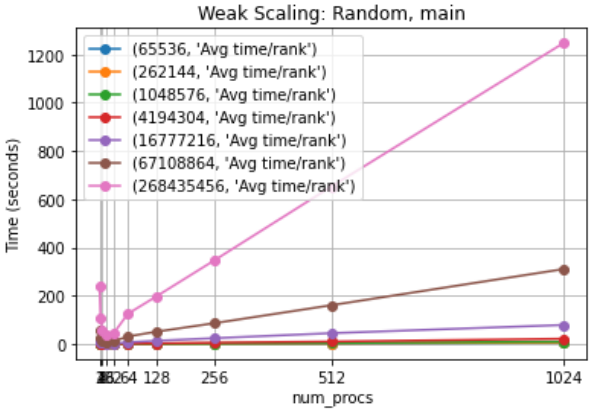


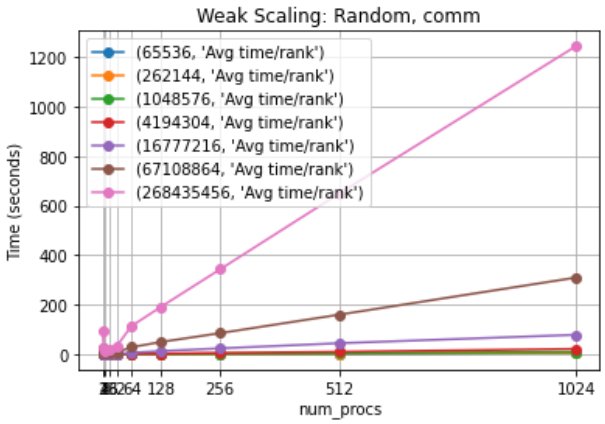




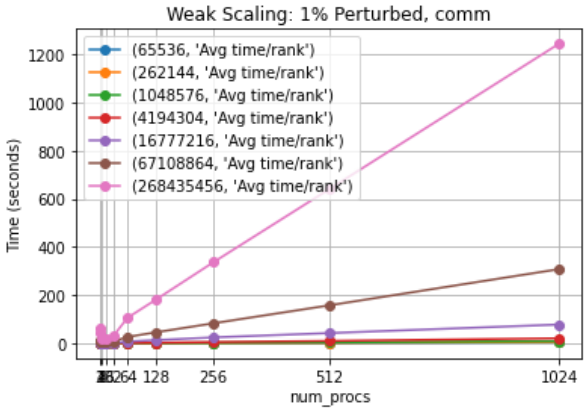
Input Type: Random

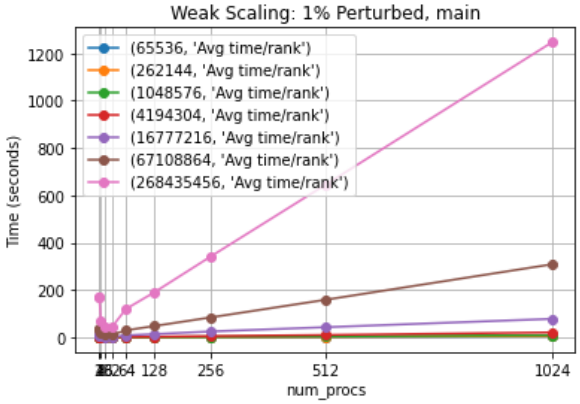


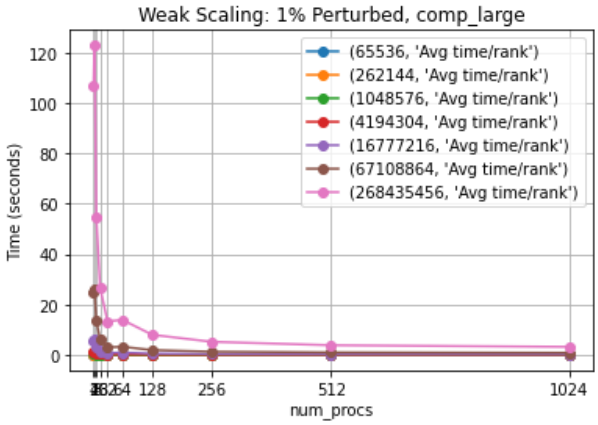




Input Type: 1% Perturbed







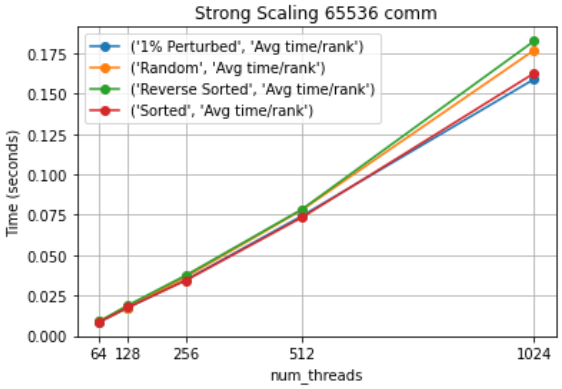
Analysis

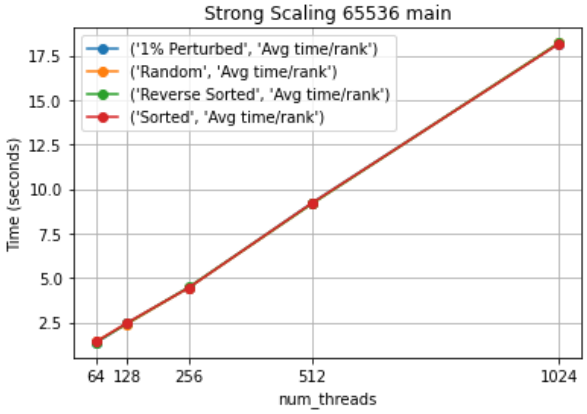
Among all the different input types, we notice the same trends for communication, computation, and the overall program (main). For computation, there is a spike at two processors, but the runtime then decreases and plateaus afterwards. Seeing that runtime plateaus is ideal and expected, because that means that the workload is being distributed equally among processors. The reason that we see a spike at 2 processors is because the time taken between the master process and worker process is vastly different. Unlike computation, we see that comm and main did not scale as well for weak scaling. This is likely due to the size of the array that processors are sending increasing with processor size. This is an issue because the master process sends this array to each processor and each processor sends it back to master the master process. Even though the work load is balanced among all processors, it does not scale well weakly because they are dealing with a larger and larger array size. Similar to the other analyses, we see that weak scaling for main follows the same trend as comm. The reason for this is also because comm is the main bottleneck, and the computation time is negligible in comparison to communication.

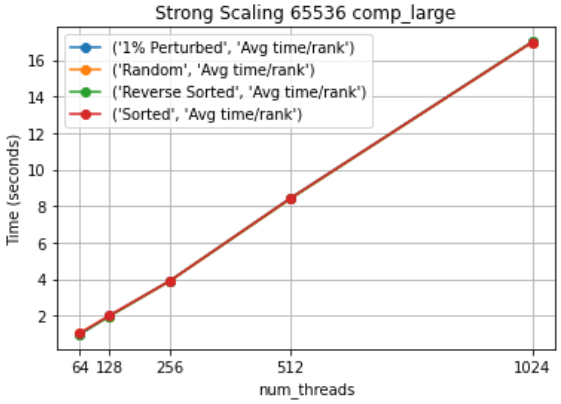
Sample Sort CUDA Graphs

Strong Scaling Plots

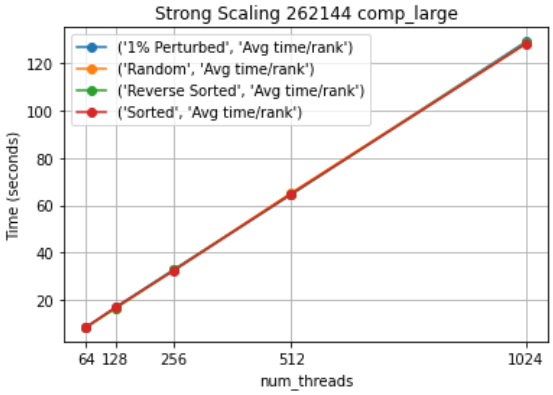
Input Size: 65536

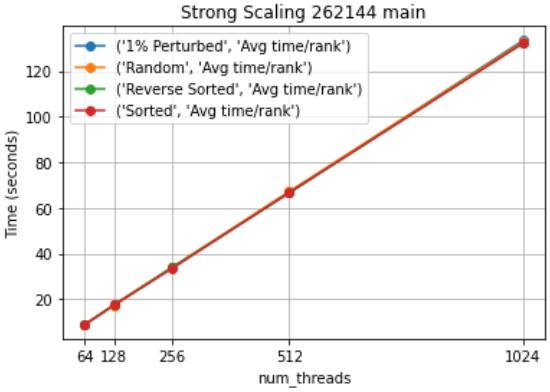


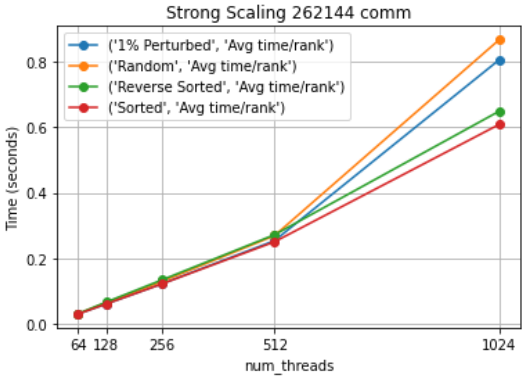




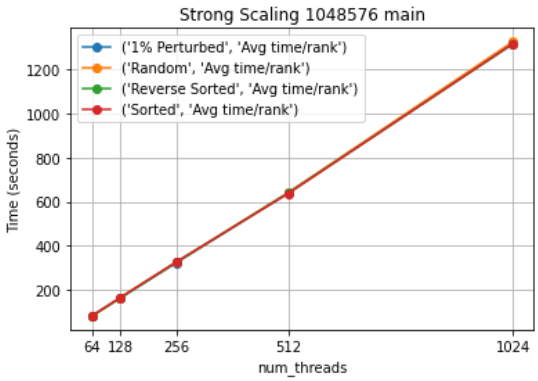
Input Size: 262144

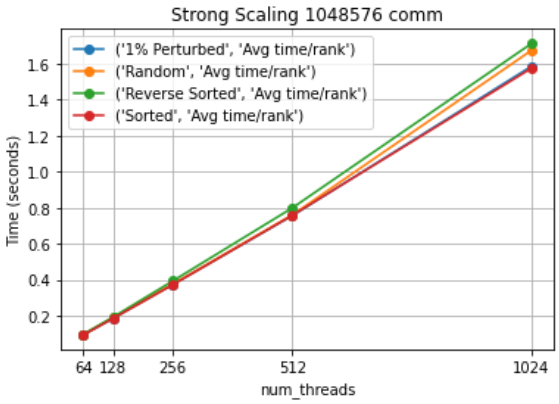






Input Size: 1048576



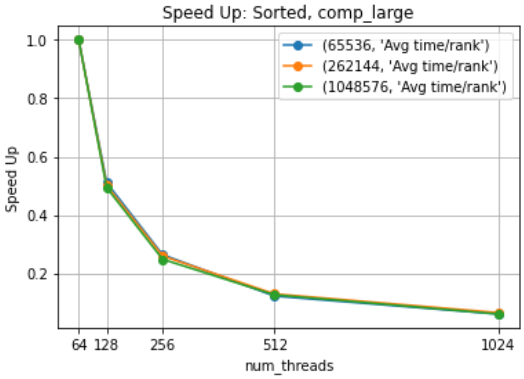


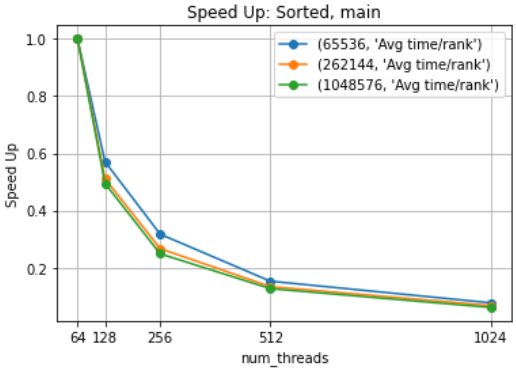
Analysis

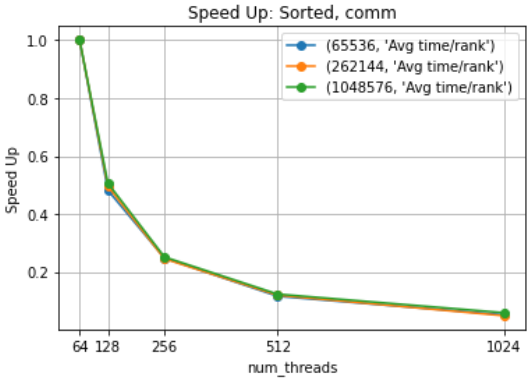
It can be seen that runtime increases for comp, comm, and main as the number of threads increases. Runtime increasing as the number of threads increases is to be expected. The reason for this is because the array used to store bucket values for sample sort is based on the number of threads. This means that as the number of threads increases, the size of the array being copied from host to device and device to host is increasing as well. Because a larger array is copied, that means that runtime would increase in conjunction with size. Unexpectedly, it can be noted that computation runtime also increases with the number of threads. Though it was expected that more threads would decrease runtime, the reason that runtime increased is because of how this implementation calculated the number of blocks. The number of blocks was calculated as the number of values divided by the number of threads. This means that the more threads there were, the less blocks there would be, meaning that there were less threads being run concurrently. This explains why computation runtime not only increased, but it increased linearly with the number of threads. The strong scaling graphs for main look exactly like the computation graphs because computation time was vastly higher than communication. Because of this, we can see that the bottleneck for this implementation was computation. We can see from these plots that this implementation for sample sort did not scale well.

Strong Scaling Speedup

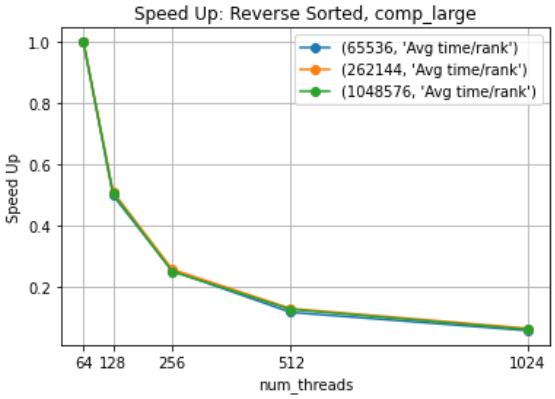
Input Type: Sorted

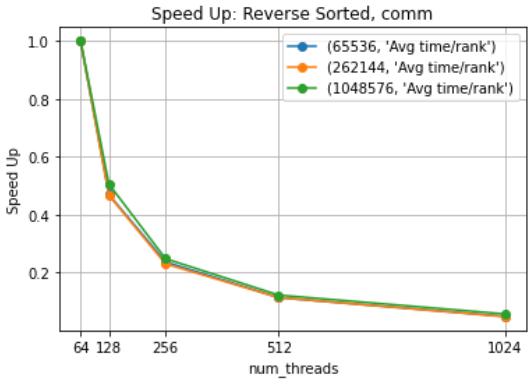


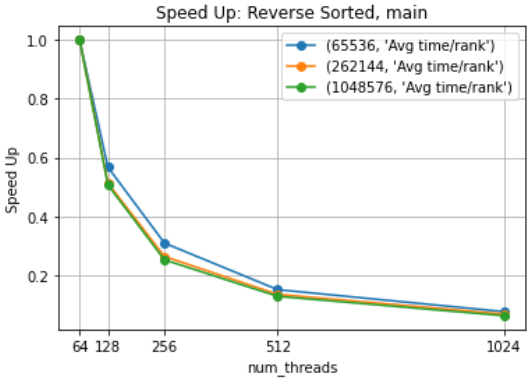




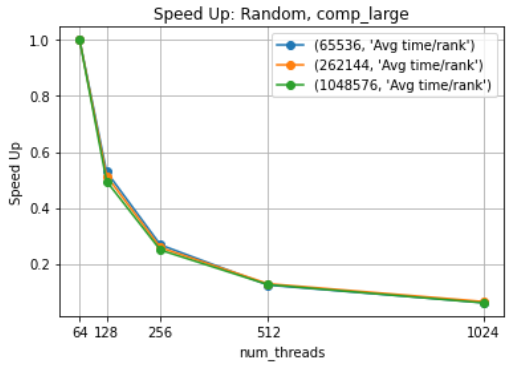
Input Type: Reverse Sorted

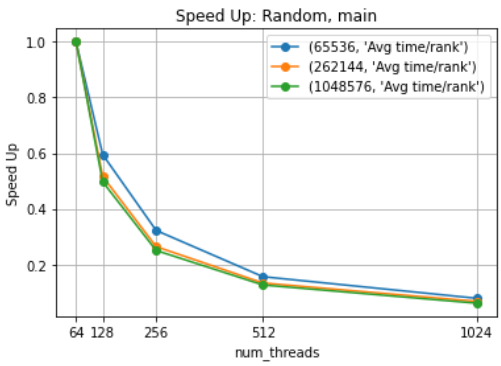




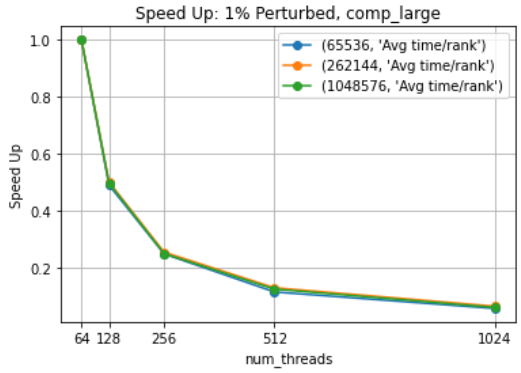


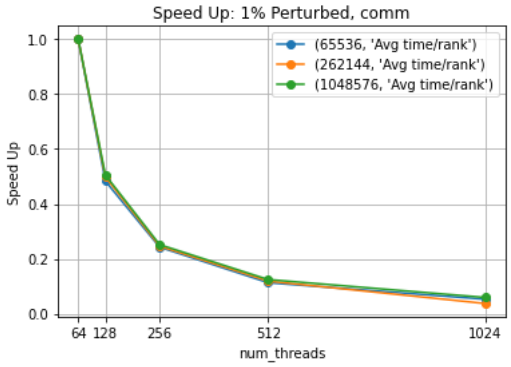
Input Type: Random

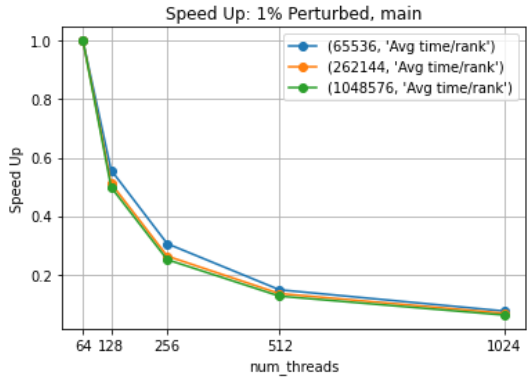




Input Type: 1% Perturbed





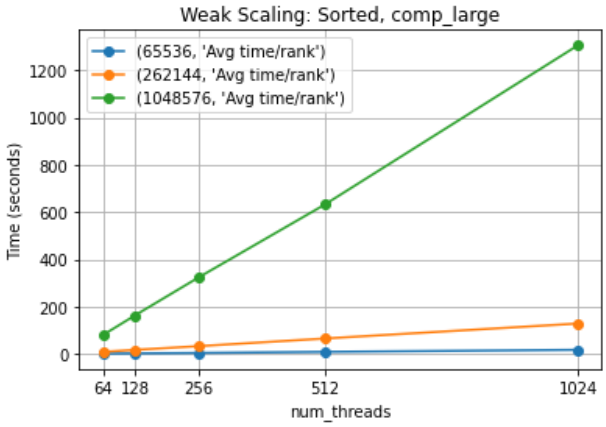


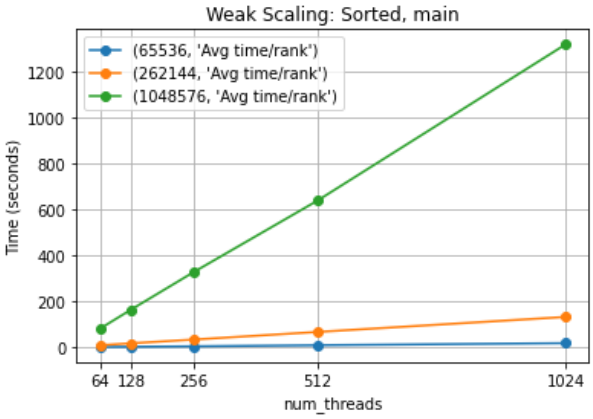
Analysis

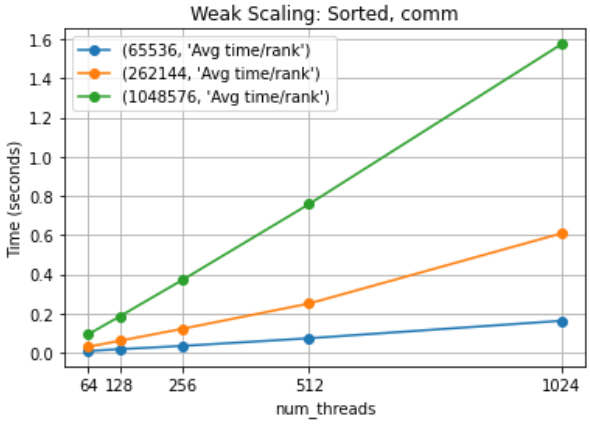
There are no notable differences for speedup among different input types; this means that input type did not have a large impact on the CUDA implementation of sample sort since the trends remained consistent regardless of input type. Speed up for computation, communication, and main can be seen to follow the same trend; they all decrease at an exponential rate. Seeing that speed up decreases exponentially for computation, communication, and main further supports the conclusion that this implementation did not scale well. These plots match up with the trends noted in the strong scaling graphs. In strong scaling, runtime increased (for comp, comm, and main) as the number of threads increased. This matches because more time being taken means that the program is getting slower, not speeding up as the number of threads increases. Because of this, it matches that the speed up graphs would decrease as the number of threads increased.

Weak Scaling

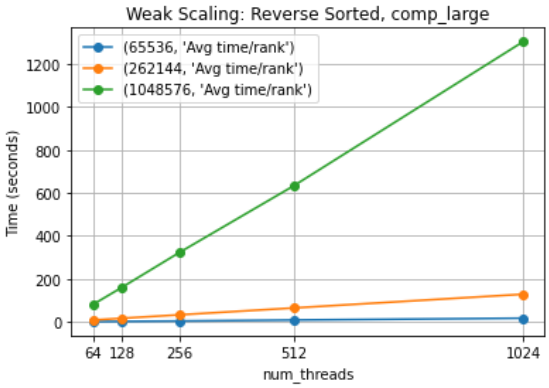
Input Type: Sorted

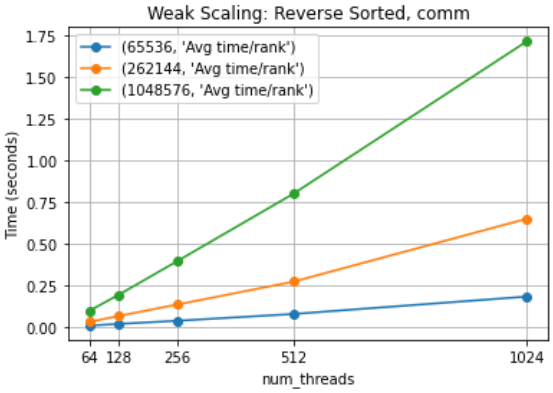


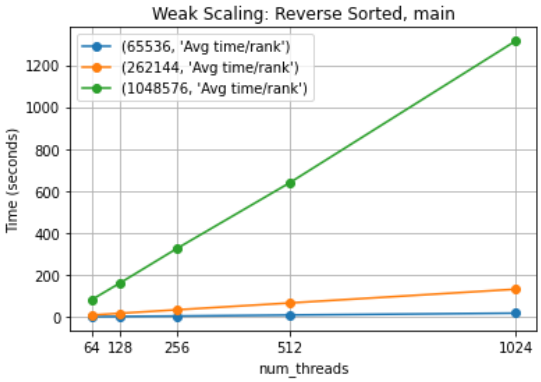




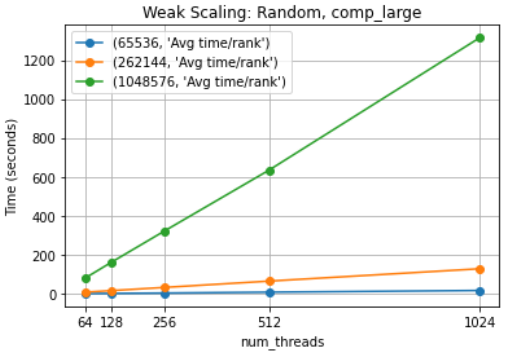
Input Type: Reverse Sorted

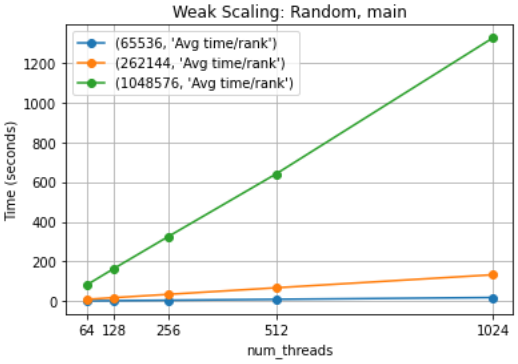


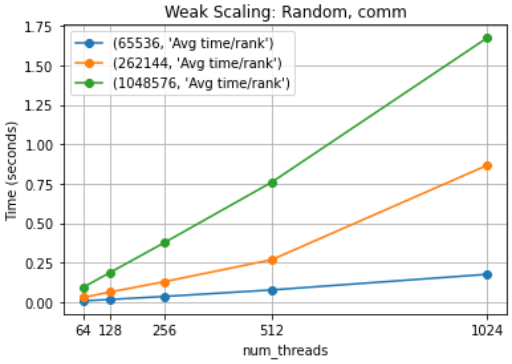




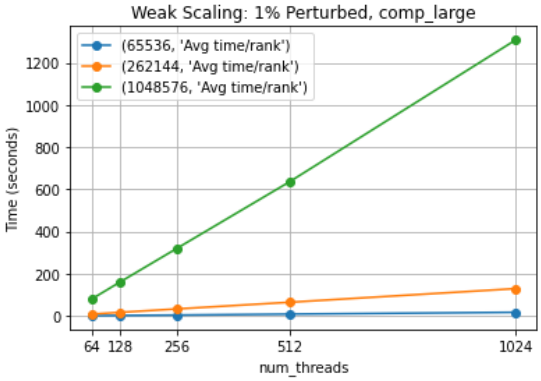
Input Type: Random

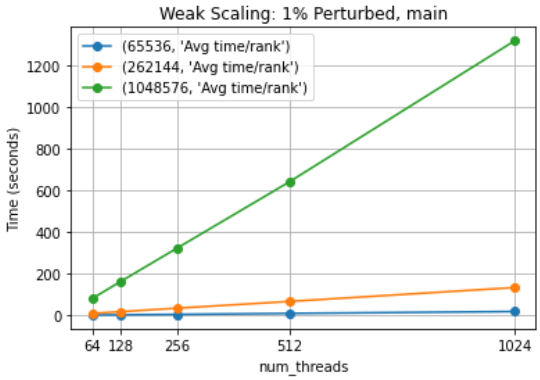


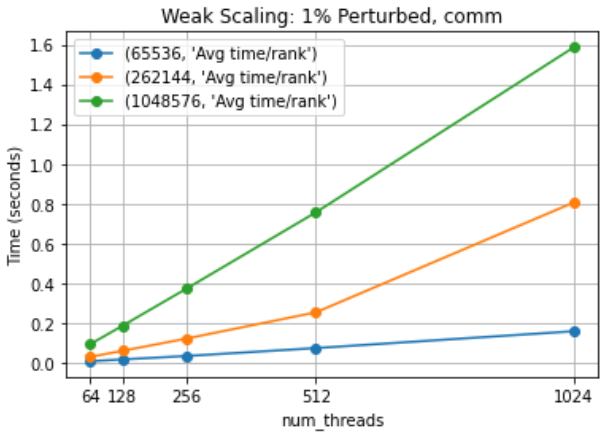




Input Type: 1% Perturbed







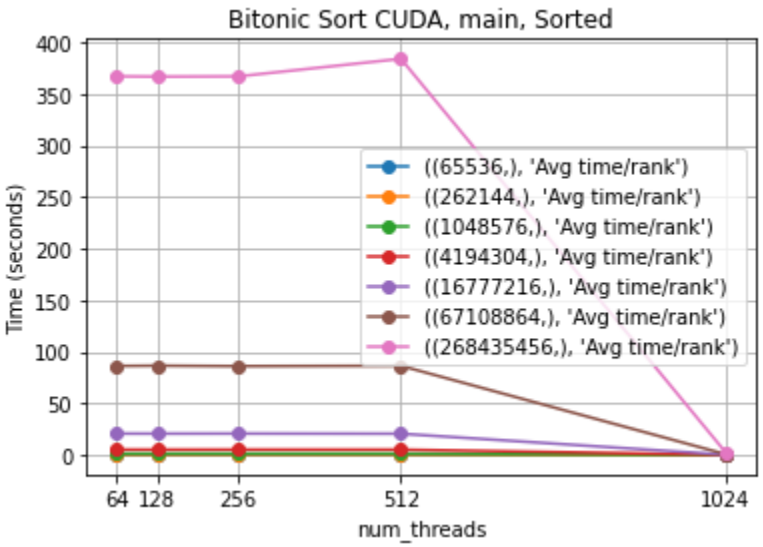
Analysis

Similar to the speed up graphs, there are no notable differences in the weak scaling plots across different input types. This further supports the conclusion that input type did not make an impact on the performance of this implementation. Time increased as the number of threads increased for computation, communication, and overall (main) time. Having computation time increase indicates that the work was not divided evenly among all threads, but I do not think that is the case for this implementation. In this implementation, the number of blocks decreased as the number of threads increased. So, though each thread had the same amount of work to do, there was less overall concurrency because these threads had to wait to be run in warps. Having less blocks meant that there were less warps being run concurrently. We can see from these graphs that this implementation did not scale well for weak scaling.

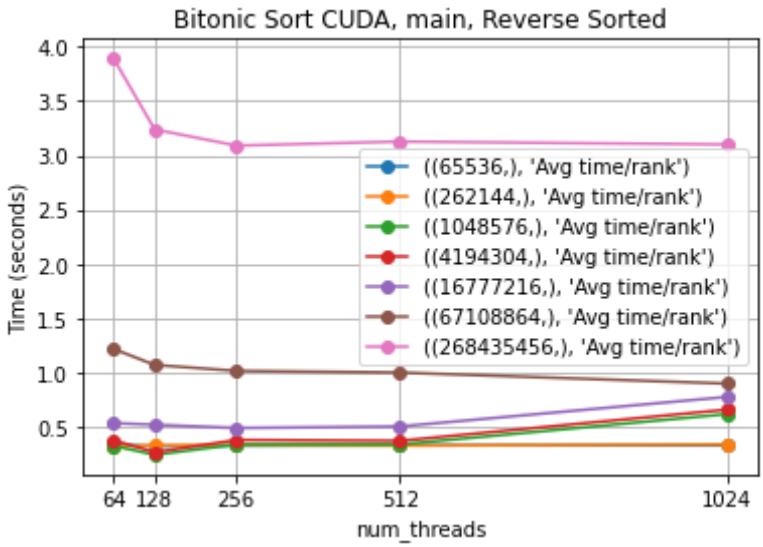
Bitonic Sort Cuda

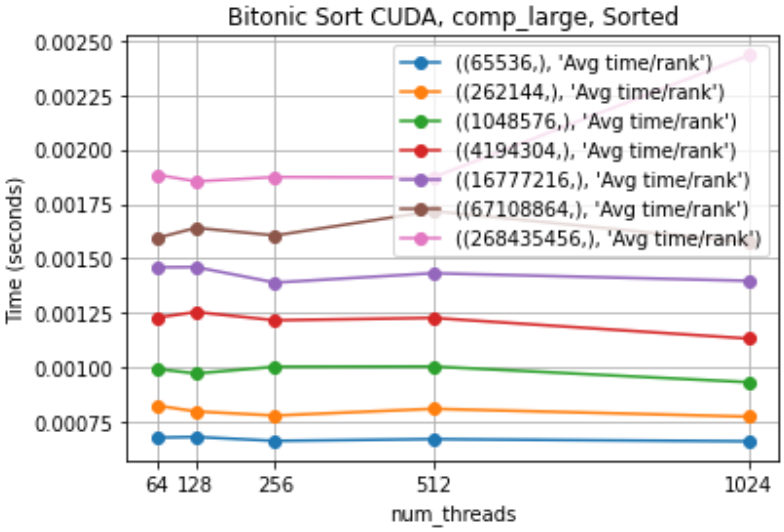
Weak Scaling

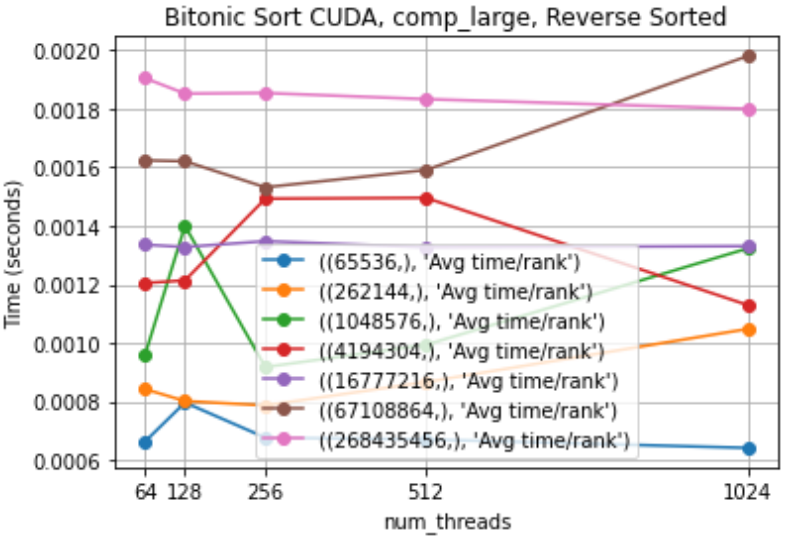
Sorted

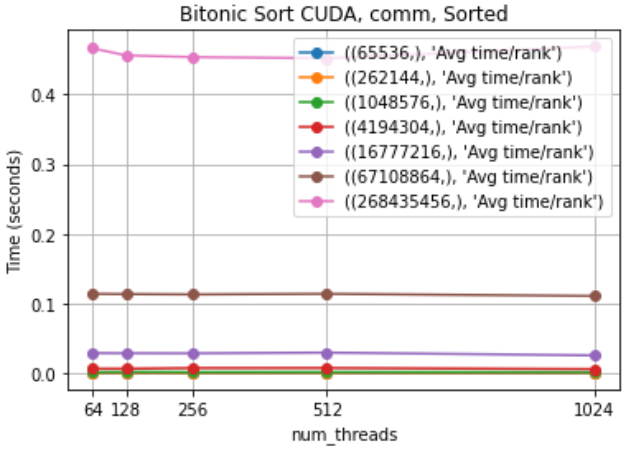


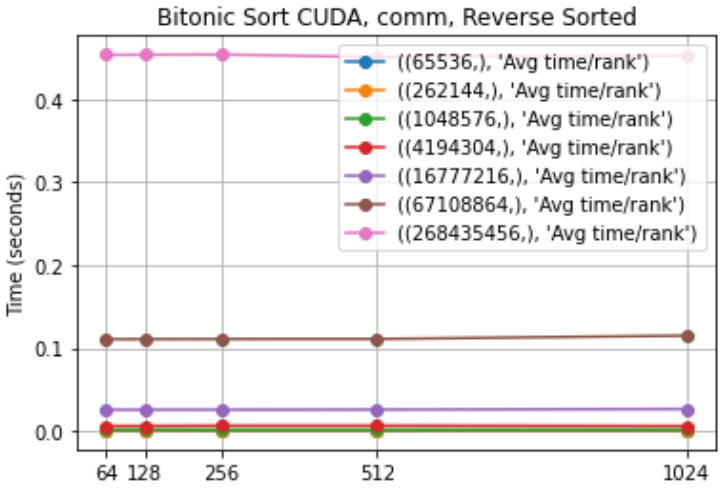
Reverse Sorted

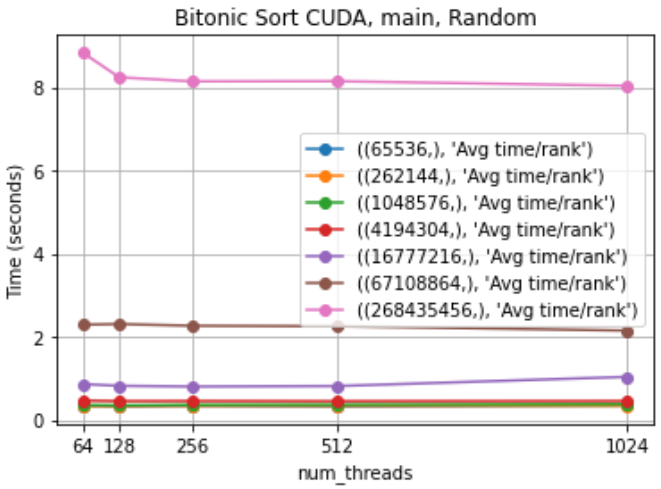


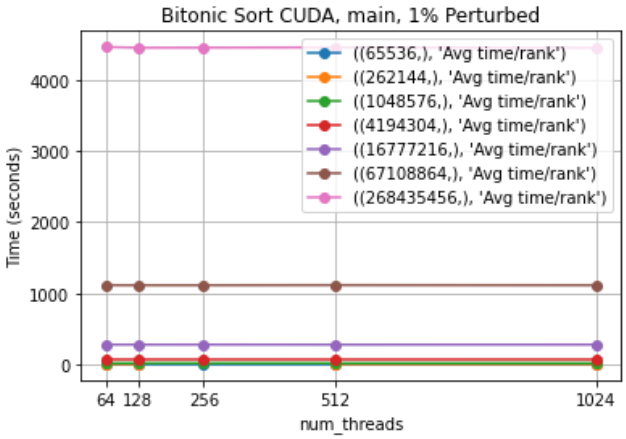


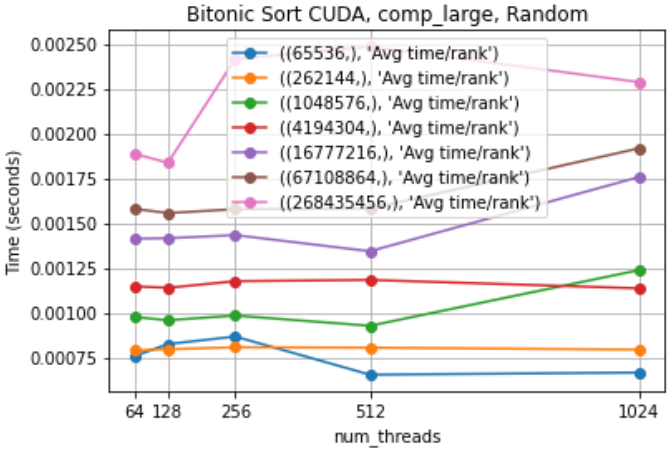


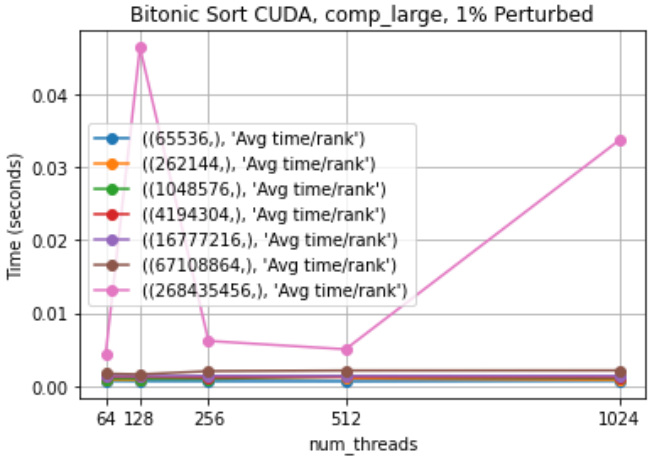


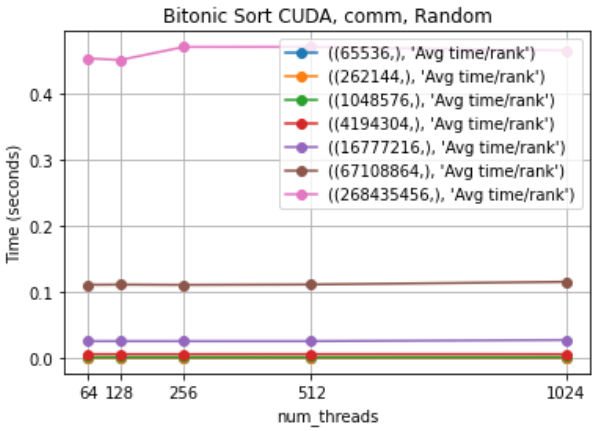


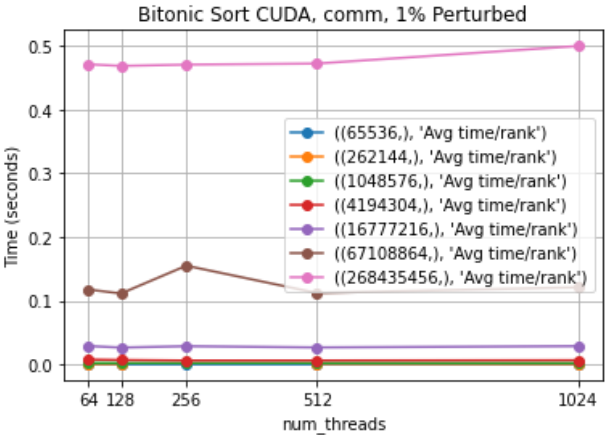

Bitonic Sort CUDA, main, Random
Time (seconds)
num_threads
((65536,), 'Avg time/rank')
((262144,), 'Avg time/rank')
((1048576,), 'Avg time/rank')
((4194304,), 'Avg time/rank')
((16777216,), 'Avg time/rank')
((67108864,), 'Avg time/rank')
((268435456,), 'Avg time/rank')


Bitonic Sort CUDA, main, 1% Perturbed
Time (seconds)
num_threads
((65536,), 'Avg time/rank')
((262144,), 'Avg time/rank')
((1048576,), 'Avg time/rank')
((4194304,), 'Avg time/rank')
((16777216,), 'Avg time/rank')
((67108864,), 'Avg time/rank')
((268435456,), 'Avg time/rank')

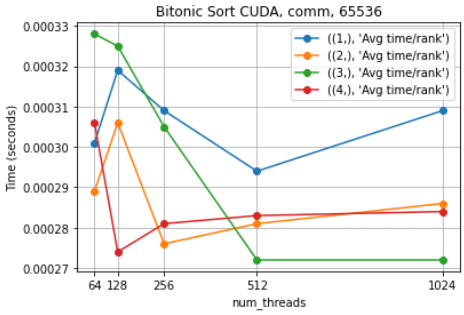

Bitonic Sort CUDA, comp_large, Random
Time (seconds)
num_threads
((65536,), 'Avg time/rank')
((262144,), 'Avg time/rank')
((1048576,), 'Avg time/rank')
((4194304,), 'Avg time/rank')
((16777216,), 'Avg time/rank')
((67108864,), 'Avg time/rank')
((268435456,), 'Avg time/rank')

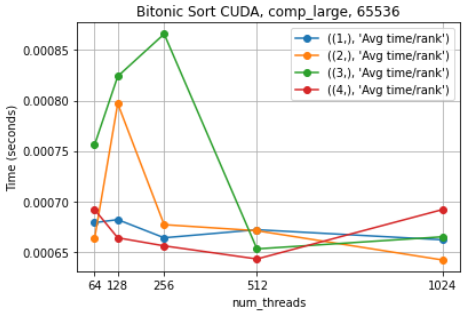

Bitonic Sort CUDA, comp_large, 1% Perturbed
Time (seconds)
num_threads
((65536,), 'Avg time/rank')
((262144,), 'Avg time/rank')
((1048576,), 'Avg time/rank')
((4194304,), 'Avg time/rank')
((16777216,), 'Avg time/rank')
((67108864,), 'Avg time/rank')
((268435456,), 'Avg time/rank')


Bitonic Sort CUDA, comm, Random
Time (seconds)
num_threads
((65536,), 'Avg time/rank')
((262144,), 'Avg time/rank')
((1048576,), 'Avg time/rank')
((4194304,), 'Avg time/rank')
((16777216,), 'Avg time/rank')
((67108864,), 'Avg time/rank')
((268435456,), 'Avg time/rank')

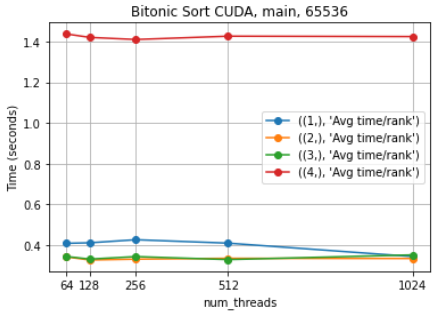

Bitonic Sort CUDA, comm, 1% Perturbed
Time (seconds)
num_threads
((65536,), 'Avg time/rank')
((262144,), 'Avg time/rank')
((1048576,), 'Avg time/rank')
((4194304,), 'Avg time/rank')
((16777216,), 'Avg time/rank')
((67108864,), 'Avg time/rank')
((268435456,), 'Avg time/rank')

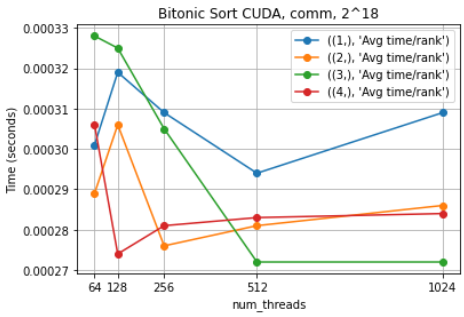
Analysis: The weak scaling graphs for Bitonic Sort Cuda are very ideal because as the number of threads double in size and as input size increases you can see that the lines for the different sorts are pretty horizontal. This shows that the goal of weak scaling is accomplished and shows that to keeping the workload per thread constant as there are more threads being added. This si because each thread is handling the same amount of “work” as it was before. However some graphs, such as the comp_large 1% perturbed graph has a large spike, this can be because the communication between threads is not being efficiently completed, this can be due to communication overhead and have a large input size.

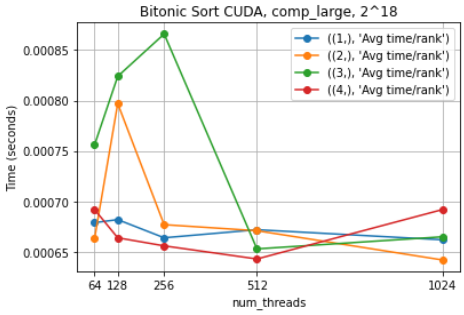


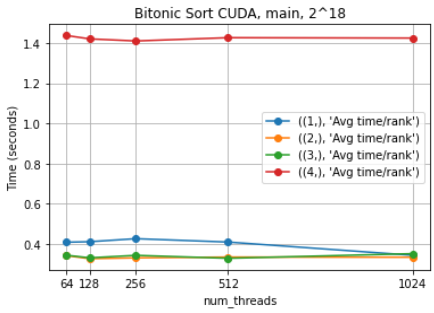


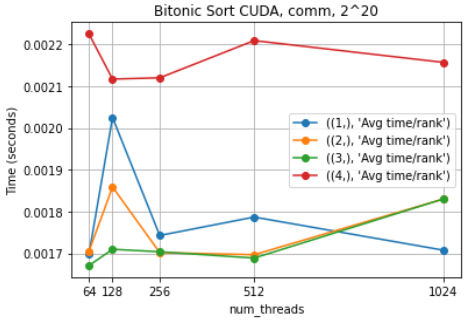
Strong Scaling

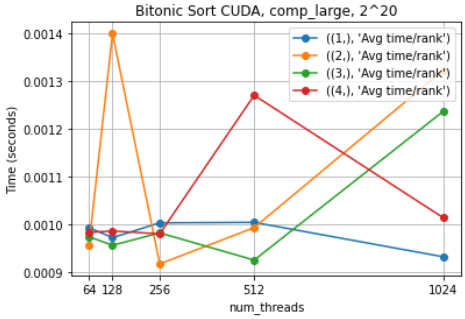


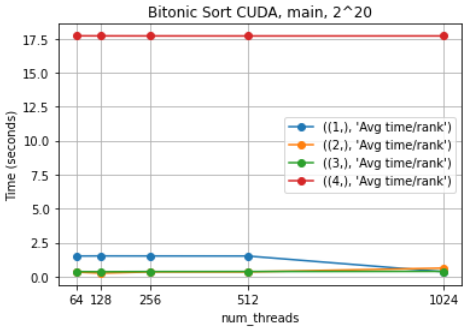


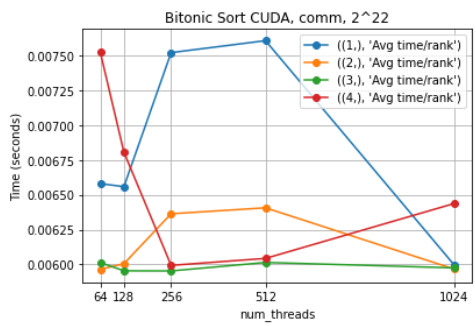


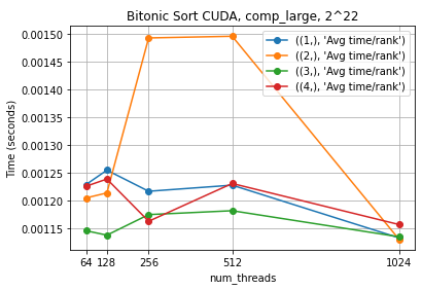


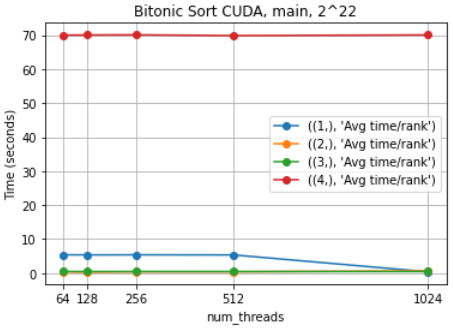


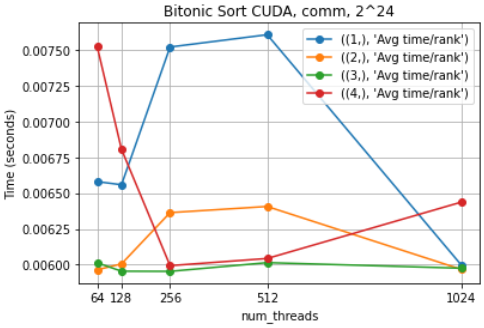


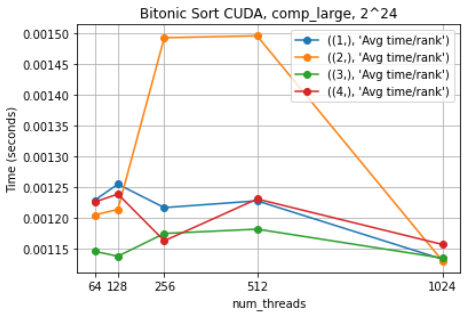


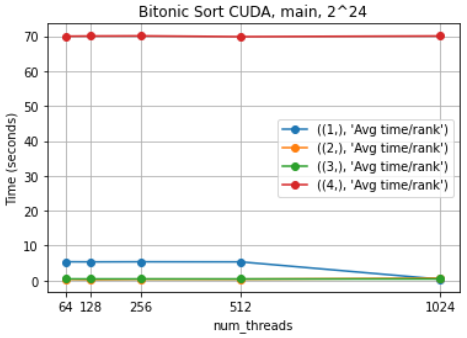


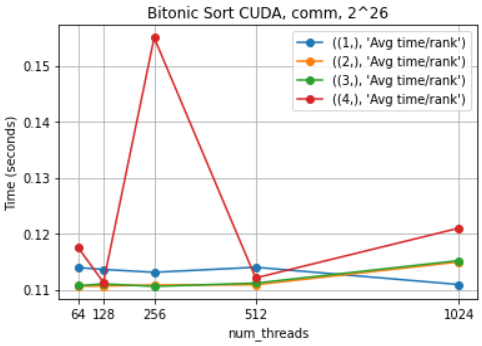


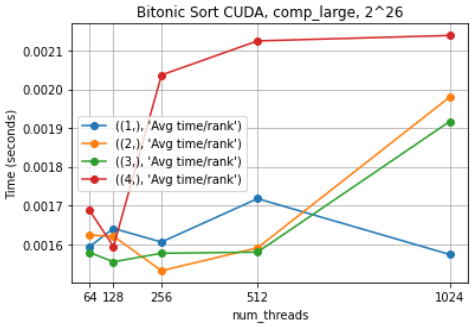


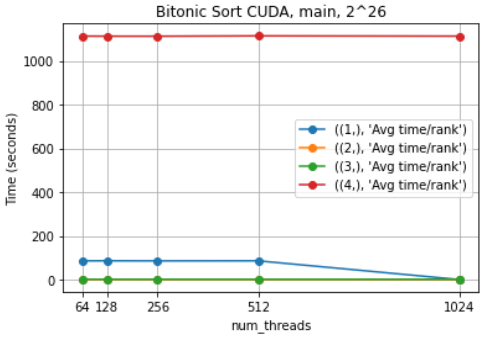

Bitonic Sort CUDA, comm, 2^24
Time (seconds)
num_threads
((1,), 'Avg time/rank')
((2,), 'Avg time/rank')
((3,), 'Avg time/rank')
((4,), 'Avg time/rank')

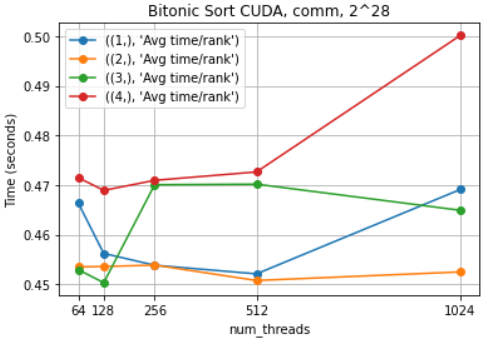

Bitonic Sort CUDA, comp_large, 2^24
Time (seconds)
num_threads
((1,), 'Avg time/rank')
((2,), 'Avg time/rank')
((3,), 'Avg time/rank')
((4,), 'Avg time/rank')

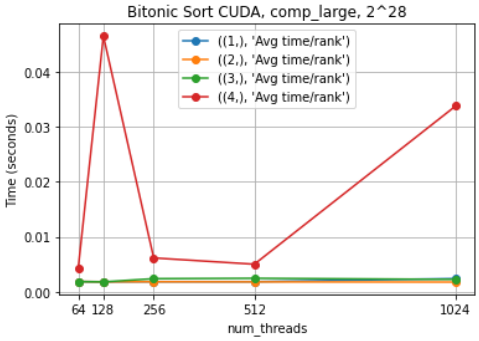

Bitonic Sort CUDA, main, 2^24
Time (seconds)
num_threads
((1,), 'Avg time/rank')
((2,), 'Avg time/rank')
((3,), 'Avg time/rank')
((4,), 'Avg time/rank')

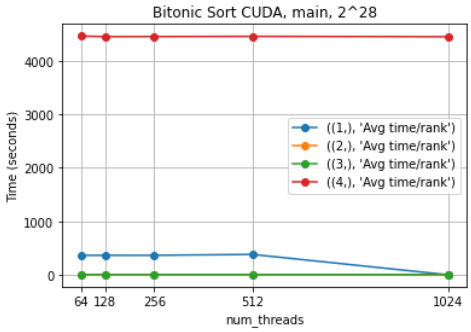

Bitonic Sort CUDA, comm, 2^26
Time (seconds)
num_threads
((1,), 'Avg time/rank')
((2,), 'Avg time/rank')
((3,), 'Avg time/rank')
((4,), 'Avg time/rank')


Bitonic Sort CUDA, comp_large, 2^26
Time (seconds)
num_threads
((1,), 'Avg time/rank')
((2,), 'Avg time/rank')
((3,), 'Avg time/rank')
((4,), 'Avg time/rank')


Bitonic Sort CUDA, main, 2^26
Time (seconds)
num_threads
((1,), 'Avg time/rank')
((2,), 'Avg time/rank')
((3,), 'Avg time/rank')
((4,), 'Avg time/rank')


Bitonic Sort CUDA, comm, 2^28
Time (seconds)
num_threads
((1,), 'Avg time/rank')
((2,), 'Avg time/rank')
((3,), 'Avg time/rank')
((4,), 'Avg time/rank')

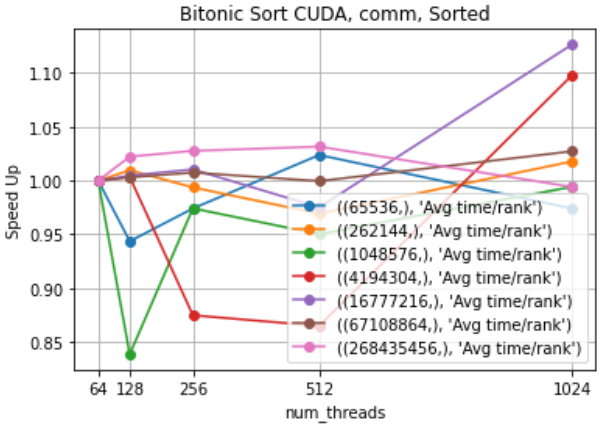

Bitonic Sort CUDA, comp_large, 2^28
Time (seconds)
num_threads
((1,), 'Avg time/rank')
((2,), 'Avg time/rank')
((3,), 'Avg time/rank')
((4,), 'Avg time/rank')

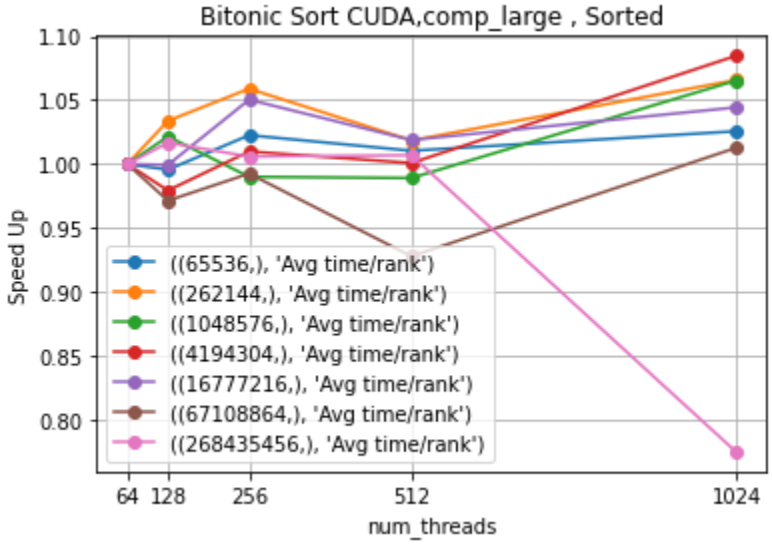

Bitonic Sort CUDA, main, 2^28
Time (seconds)
num_threads
((1,), 'Avg time/rank')
((2,), 'Avg time/rank')
((3,), 'Avg time/rank')
((4,), 'Avg time/rank')

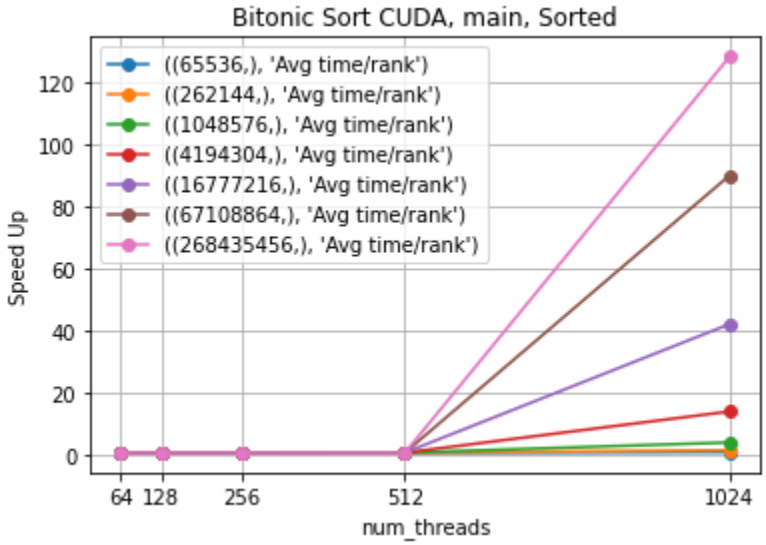
Analysis:
Prerequisite. For the legend, 1 is for sorted, 2 is for Reverse sorted, 3 is for Random and 4 is for 1% perturbed.

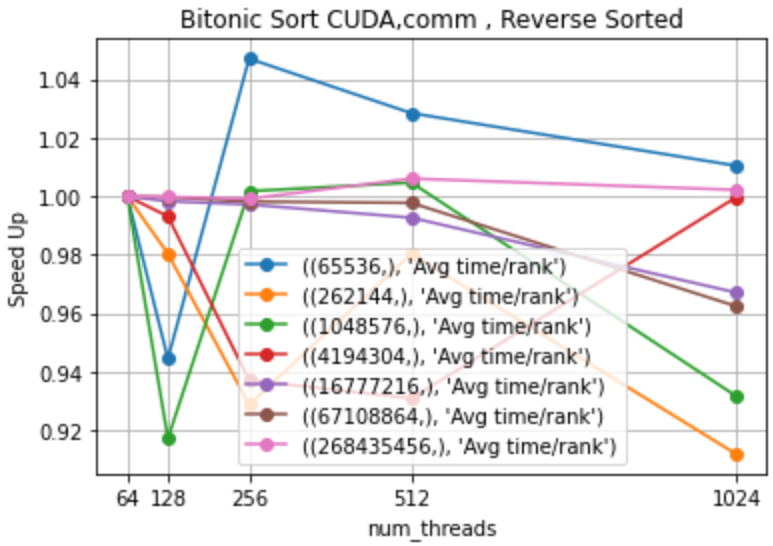
For the Bitonic Cuda implementation for strong scaling, the graphs follow a discernible decreasing trend, however some sorting algorithms for the input sizes do not follow this trend the majority of the sorts do. Some sorting algorithms such as comm for 2 ^28 for the perturbed sorting algorithms, as number of threads is increasing so does the time, this may have to due with the huge input size and the increased communication overhead that takes place in the Cudamemcpy, when a large numbers of data is being transferred from one array to another. There are some random large spikes especially in comp_large 2 ^16 and 2^18 for randomly sorted algorithms, this can be due to the low communication that is occurring between the threads, this can be shown in the comm graph for 2^16 and 2^18. As you can see when the comp_large graph spikes the communication graph is lowering. As the time increases in the communication graph, the computation graph is decreasing(with the same number of threads the time is lower). The different sort types can cause a workload imbalance among the threads, resulting in the differing runtimes caused by the communication overhead.

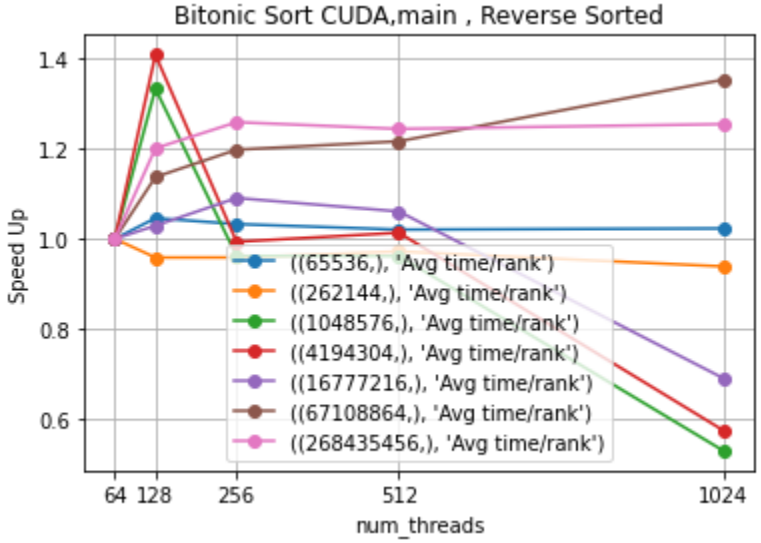
Speed Up

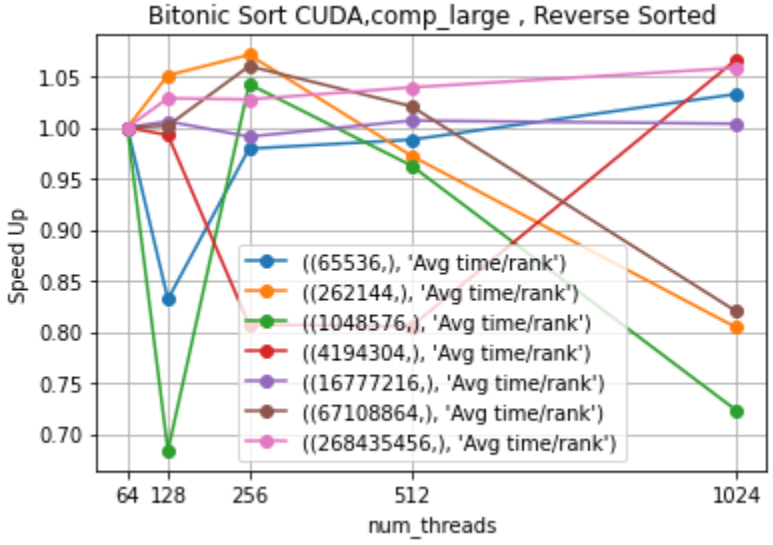

Bitonic Sort CUDA, comm, Sorted
Speed Up
num_threads
1.10
1.05
1.00
0.95
0.90
0.85
64
128
256
512
1024
((65536,), 'Avg time/rank')
((262144,), 'Avg time/rank')
((1048576,), 'Avg time/rank')
((4194304,), 'Avg time/rank')
((16777216,), 'Avg time/rank')
((67108864,), 'Avg time/rank')
((268435456,), 'Avg time/rank')

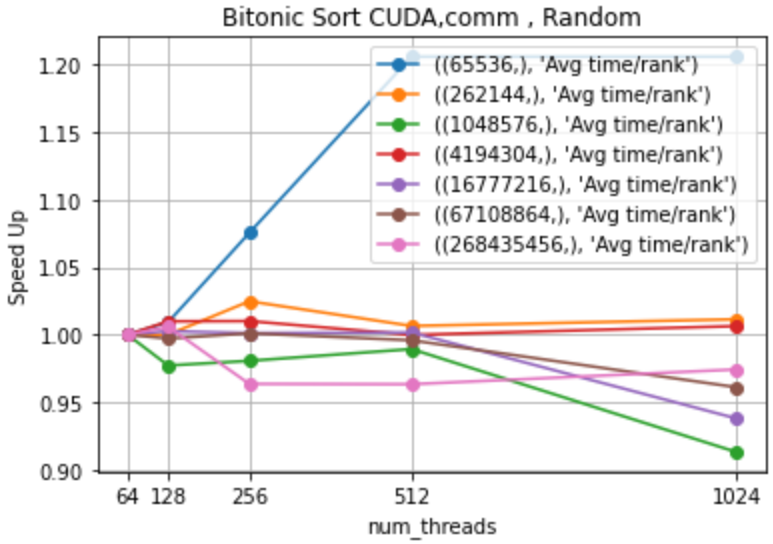

Bitonic Sort CUDA,comp_large , Sorted
Speed Up
num_threads
1.10
1.05
1.00
0.95
0.90
0.85
0.80
64
128
256
512
1024
((65536,), 'Avg time/rank')
((262144,), 'Avg time/rank')
((1048576,), 'Avg time/rank')
((4194304,), 'Avg time/rank')
((16777216,), 'Avg time/rank')
((67108864,), 'Avg time/rank')
((268435456,), 'Avg time/rank')

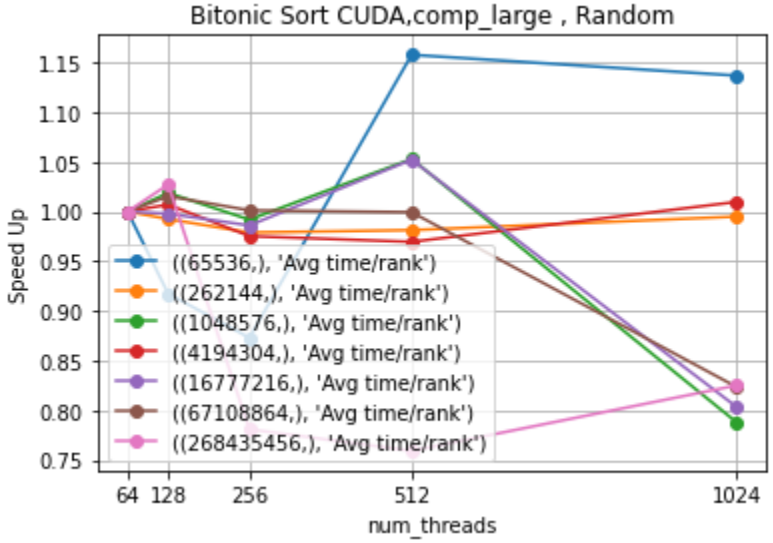

Bitonic Sort CUDA, main, Sorted
Speed Up
num_threads
120
100
80
60
40
20
0
64
128
256
512
1024
((65536,), 'Avg time/rank')
((262144,), 'Avg time/rank')
((1048576,), 'Avg time/rank')
((4194304,), 'Avg time/rank')
((16777216,), 'Avg time/rank')
((67108864,), 'Avg time/rank')
((268435456,), 'Avg time/rank')

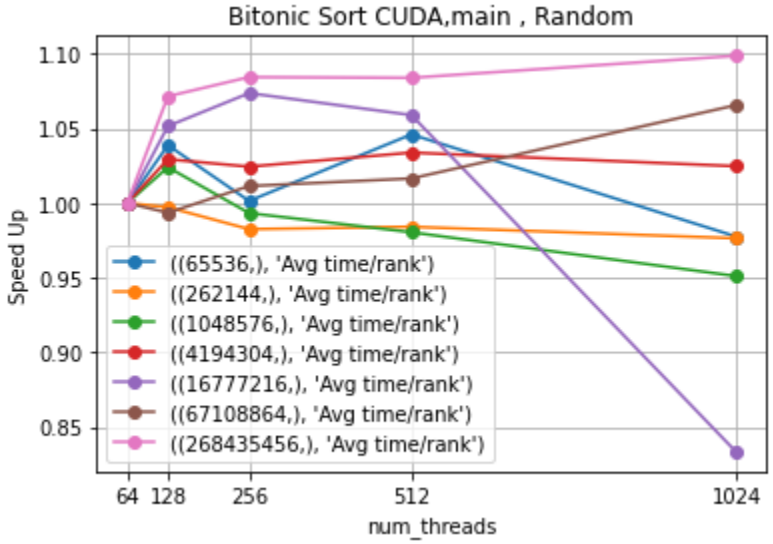

Bitonic Sort CUDA,comm , Reverse Sorted
Speed Up
num_threads
1.04
1.02
1.00
0.98
0.96
0.94
0.92
64
128
256
512
1024
((65536,), 'Avg time/rank')
((262144,), 'Avg time/rank')
((1048576,), 'Avg time/rank')
((4194304,), 'Avg time/rank')
((16777216,), 'Avg time/rank')
((67108864,), 'Avg time/rank')
((268435456,), 'Avg time/rank')

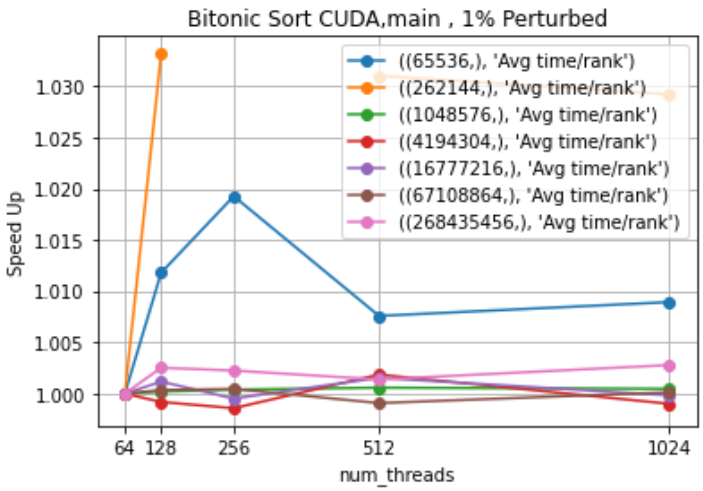

Bitonic Sort CUDA,main , Reverse Sorted
Speed Up
num_threads
1.4
1.2
1.0
0.8
0.6
64
128
256
512
1024
((65536,), 'Avg time/rank')
((262144,), 'Avg time/rank')
((1048576,), 'Avg time/rank')
((4194304,), 'Avg time/rank')
((16777216,), 'Avg time/rank')
((67108864,), 'Avg time/rank')
((268435456,), 'Avg time/rank')

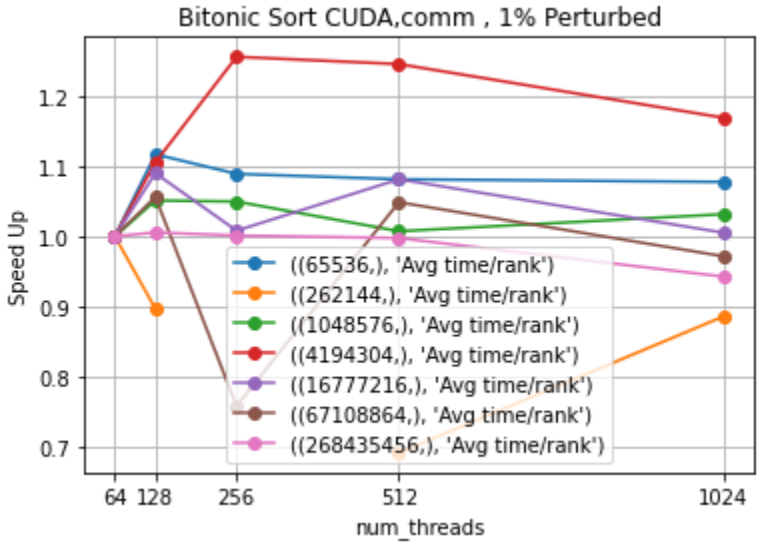

Bitonic Sort CUDA,comp_large , Reverse Sorted
Speed Up
num_threads
1.05
1.00
0.95
0.90
0.85
0.80
0.75
0.70
64
128
256
512
1024
((65536,), 'Avg time/rank')
((262144,), 'Avg time/rank')
((1048576,), 'Avg time/rank')
((4194304,), 'Avg time/rank')
((16777216,), 'Avg time/rank')
((67108864,), 'Avg time/rank')
((268435456,), 'Avg time/rank')

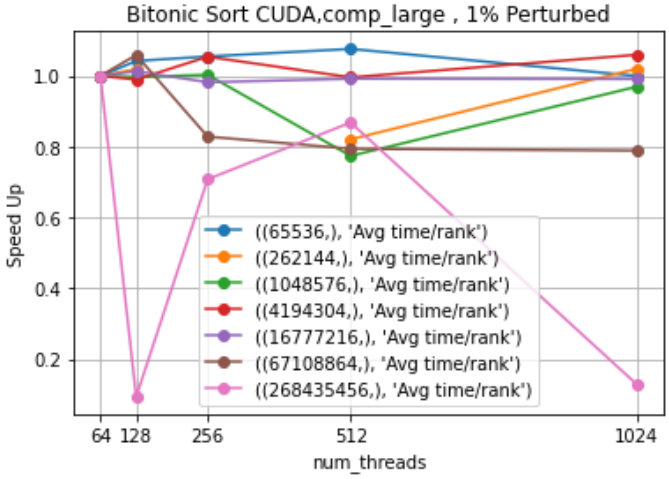

Bitonic Sort CUDA,comm , Random
Speed Up
num_threads
((65536,), 'Avg time/rank')
((262144,), 'Avg time/rank')
((1048576,), 'Avg time/rank')
((4194304,), 'Avg time/rank')
((16777216,), 'Avg time/rank')
((67108864,), 'Avg time/rank')
((268435456,), 'Avg time/rank')


Bitonic Sort CUDA,comp_large , Random
Speed Up
num_threads
((65536,), 'Avg time/rank')
((262144,), 'Avg time/rank')
((1048576,), 'Avg time/rank')
((4194304,), 'Avg time/rank')
((16777216,), 'Avg time/rank')
((67108864,), 'Avg time/rank')
((268435456,), 'Avg time/rank')


Bitonic Sort CUDA,main , Random
Speed Up
num_threads
((65536,), 'Avg time/rank')
((262144,), 'Avg time/rank')
((1048576,), 'Avg time/rank')
((4194304,), 'Avg time/rank')
((16777216,), 'Avg time/rank')
((67108864,), 'Avg time/rank')
((268435456,), 'Avg time/rank')


Bitonic Sort CUDA,main , 1% Perturbed
Speed Up
num_threads
((65536,), 'Avg time/rank')
((262144,), 'Avg time/rank')
((1048576,), 'Avg time/rank')
((4194304,), 'Avg time/rank')
((16777216,), 'Avg time/rank')
((67108864,), 'Avg time/rank')
((268435456,), 'Avg time/rank')


Bitonic Sort CUDA,comm , 1% Perturbed
Speed Up
num_threads
((65536,), 'Avg time/rank')
((262144,), 'Avg time/rank')
((1048576,), 'Avg time/rank')
((4194304,), 'Avg time/rank')
((16777216,), 'Avg time/rank')
((67108864,), 'Avg time/rank')
((268435456,), 'Avg time/rank')

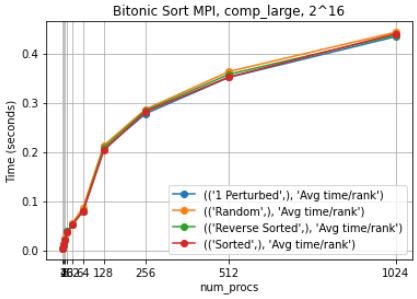

Bitonic Sort CUDA,comp_large , 1% Perturbed
Speed Up
num_threads
((65536,), 'Avg time/rank')
((262144,), 'Avg time/rank')
((1048576,), 'Avg time/rank')
((4194304,), 'Avg time/rank')
((16777216,), 'Avg time/rank')
((67108864,), 'Avg time/rank')
((268435456,), 'Avg time/rank')

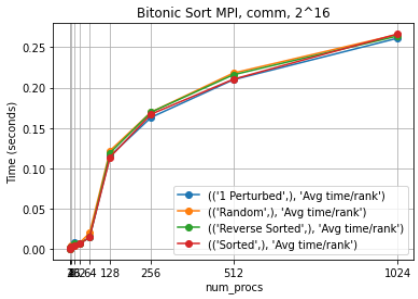
Analysis:

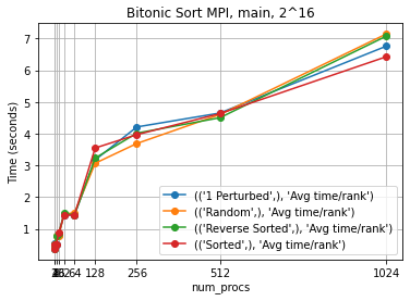
The speedup graphs for Bitonic sort Cuda are not ideal. The sorted graphs seem to increase and then go back to speedup of one when reaching the number of threads as 512. The Reverse sorted randomly spikes and then plateaus. The random graphs seem to be increasing and plateauing. And lastly the 1% perturbed are all over the place. This shows that the algorithm did not scale very well and adding more and more threads did not help improve the algorithm's performance.

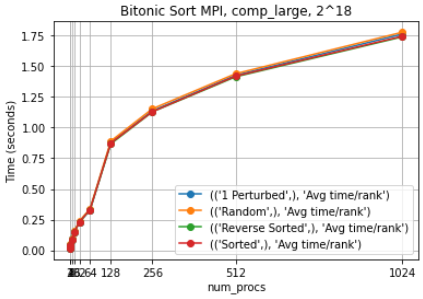
Bitonic Sort MPI

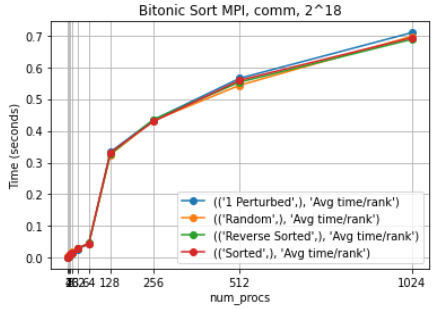
Strong Scaling

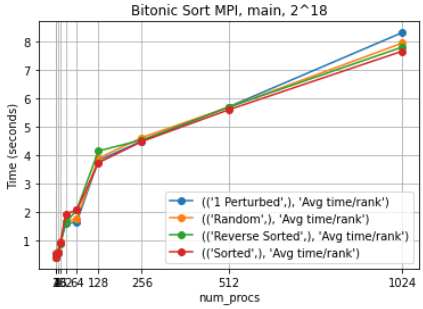

Bitonic Sort MPI, comp_large, 2^16
Time (seconds)
num_procs
('1 Perturbed',), 'Avg time/rank')
('Random',), 'Avg time/rank')
('Reverse Sorted',), 'Avg time/rank')
('Sorted',), 'Avg time/rank')

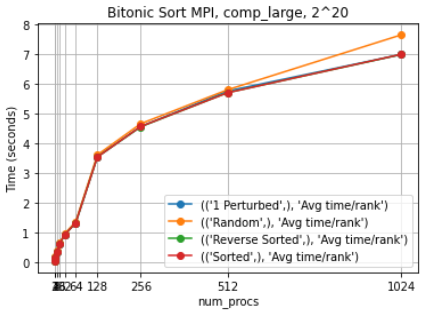

Bitonic Sort MPI, comm, 2^16
Time (seconds)
num_procs
('1 Perturbed',), 'Avg time/rank')
('Random',), 'Avg time/rank')
('Reverse Sorted',), 'Avg time/rank')
('Sorted',), 'Avg time/rank')

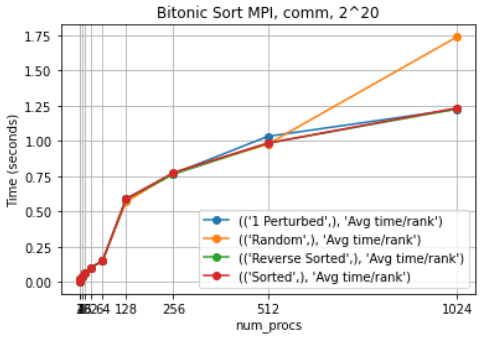

Bitonic Sort MPI, main, 2^16
Time (seconds)
num_procs
('1 Perturbed',), 'Avg time/rank')
('Random',), 'Avg time/rank')
('Reverse Sorted',), 'Avg time/rank')
('Sorted',), 'Avg time/rank')

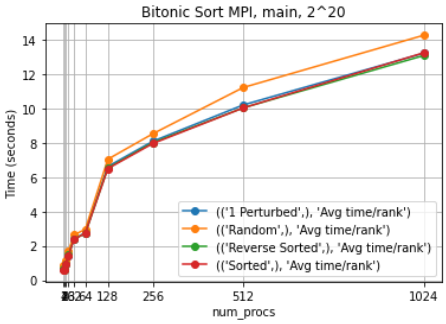

Bitonic Sort MPI, comp_large, 2^18
Time (seconds)
num_procs
('1 Perturbed',), 'Avg time/rank')
('Random',), 'Avg time/rank')
('Reverse Sorted',), 'Avg time/rank')
('Sorted',), 'Avg time/rank')

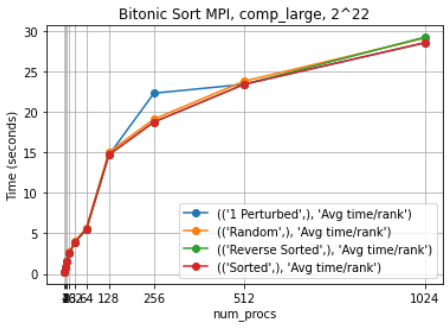

Bitonic Sort MPI, comm, 2^18
Time (seconds)
num_procs
('1 Perturbed',), 'Avg time/rank')
('Random',), 'Avg time/rank')
('Reverse Sorted',), 'Avg time/rank')
('Sorted',), 'Avg time/rank')

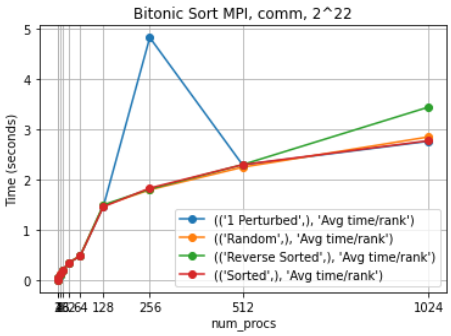

Bitonic Sort MPI, main, 2^18
Time (seconds)
num_procs
('1 Perturbed',), 'Avg time/rank')
('Random',), 'Avg time/rank')
('Reverse Sorted',), 'Avg time/rank')
('Sorted',), 'Avg time/rank')

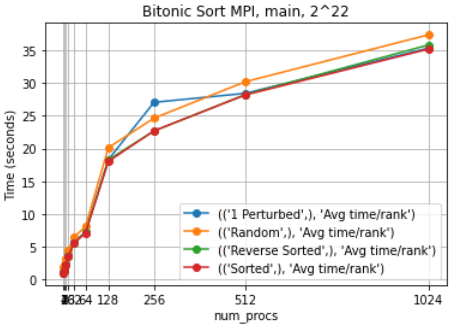

Bitonic Sort MPI, comp_large, 2^20
Time (seconds)
num_procs
('1 Perturbed',), 'Avg time/rank')
('Random',), 'Avg time/rank')
('Reverse Sorted',), 'Avg time/rank')
('Sorted',), 'Avg time/rank')

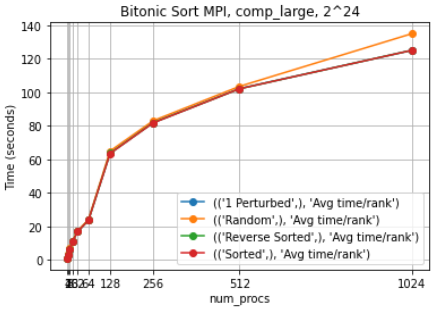

Bitonic Sort MPI, comm, 2^20
Time (seconds)
num_procs
('1 Perturbed',), 'Avg time/rank')
('Random',), 'Avg time/rank')
('Reverse Sorted',), 'Avg time/rank')
('Sorted',), 'Avg time/rank')

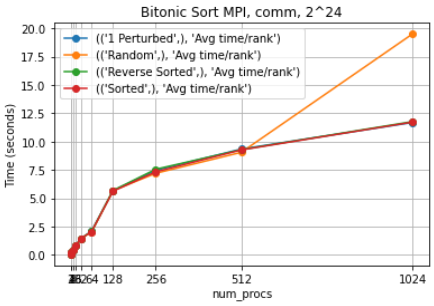

Bitonic Sort MPI, main, 2^20
Time (seconds)
num_procs
('1 Perturbed',), 'Avg time/rank')
('Random',), 'Avg time/rank')
('Reverse Sorted',), 'Avg time/rank')
('Sorted',), 'Avg time/rank')

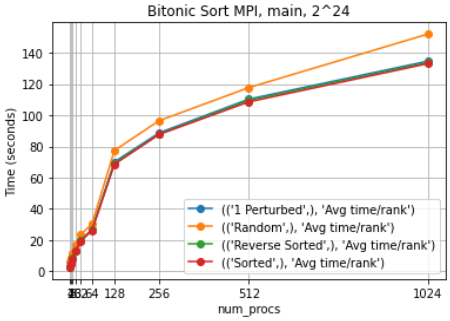

Bitonic Sort MPI, comp_large, 2^22
Time (seconds)
num_procs
('1 Perturbed',), 'Avg time/rank')
('Random',), 'Avg time/rank')
('Reverse Sorted',), 'Avg time/rank')
('Sorted',), 'Avg time/rank')

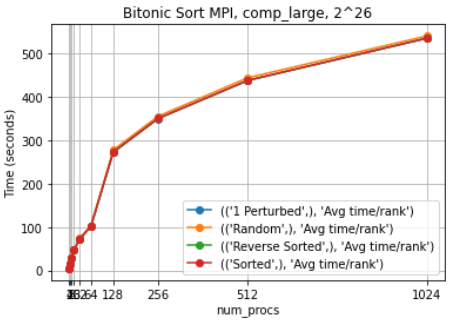

Bitonic Sort MPI, comm, 2^22
Time (seconds)
num_procs
('1 Perturbed',), 'Avg time/rank')
('Random',), 'Avg time/rank')
('Reverse Sorted',), 'Avg time/rank')
('Sorted',), 'Avg time/rank')

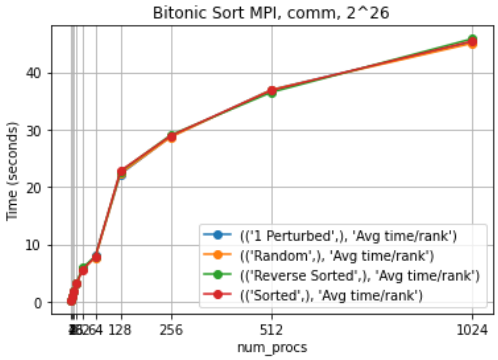

Bitonic Sort MPI, main, 2^22
Time (seconds)
num_procs
('1 Perturbed',), 'Avg time/rank')
('Random',), 'Avg time/rank')
('Reverse Sorted',), 'Avg time/rank')
('Sorted',), 'Avg time/rank')

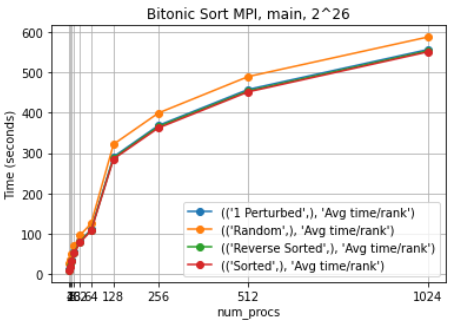

Bitonic Sort MPI, comp_large, 2^24
Time (seconds)
num_procs
(('1 Perturbed',), 'Avg time/rank')
(('Random',), 'Avg time/rank')
(('Reverse Sorted',), 'Avg time/rank')
(('Sorted',), 'Avg time/rank')

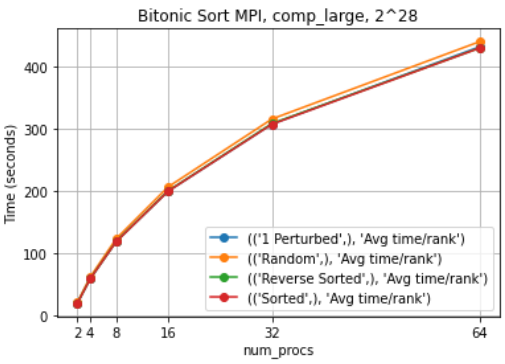

Bitonic Sort MPI, comm, 2^24
Time (seconds)
num_procs
(('1 Perturbed',), 'Avg time/rank')
(('Random',), 'Avg time/rank')
(('Reverse Sorted',), 'Avg time/rank')
(('Sorted',), 'Avg time/rank')

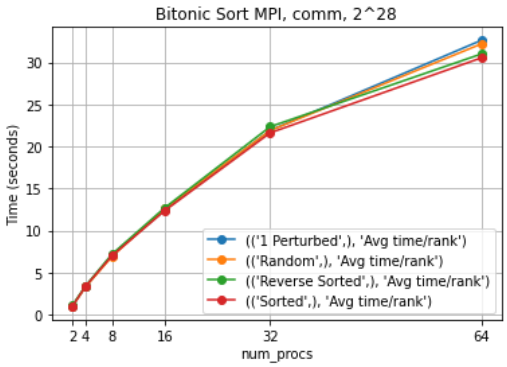

Bitonic Sort MPI, main, 2^24
Time (seconds)
num_procs
(('1 Perturbed',), 'Avg time/rank')
(('Random',), 'Avg time/rank')
(('Reverse Sorted',), 'Avg time/rank')
(('Sorted',), 'Avg time/rank')

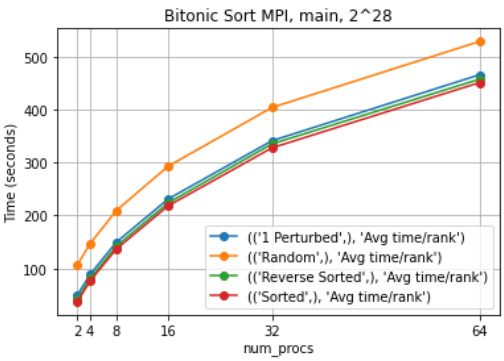

Bitonic Sort MPI, comp_large, 2^26
Time (seconds)
num_procs
(('1 Perturbed',), 'Avg time/rank')
(('Random',), 'Avg time/rank')
(('Reverse Sorted',), 'Avg time/rank')
(('Sorted',), 'Avg time/rank')


Bitonic Sort MPI, comm, 2^26
Time (seconds)
num_procs
(('1 Perturbed',), 'Avg time/rank')
(('Random',), 'Avg time/rank')
(('Reverse Sorted',), 'Avg time/rank')
(('Sorted',), 'Avg time/rank')


Bitonic Sort MPI, main, 2^26
Time (seconds)
num_procs
(('1 Perturbed',), 'Avg time/rank')
(('Random',), 'Avg time/rank')
(('Reverse Sorted',), 'Avg time/rank')
(('Sorted',), 'Avg time/rank')


Bitonic Sort MPI, comp_large, 2^28
Time (seconds)
num_procs
(('1 Perturbed',), 'Avg time/rank')
(('Random',), 'Avg time/rank')
(('Reverse Sorted',), 'Avg time/rank')
(('Sorted',), 'Avg time/rank')


Bitonic Sort MPI, comm, 2^28
Time (seconds)
num_procs
(('1 Perturbed',), 'Avg time/rank')
(('Random',), 'Avg time/rank')
(('Reverse Sorted',), 'Avg time/rank')
(('Sorted',), 'Avg time/rank')

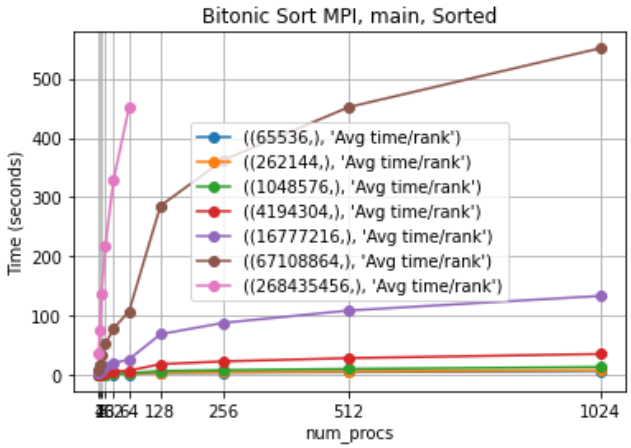

Bitonic Sort MPI, main, 2^28
Time (seconds)
num_procs
(('1 Perturbed',), 'Avg time/rank')
(('Random',), 'Avg time/rank')
(('Reverse Sorted',), 'Avg time/rank')
(('Sorted',), 'Avg time/rank')

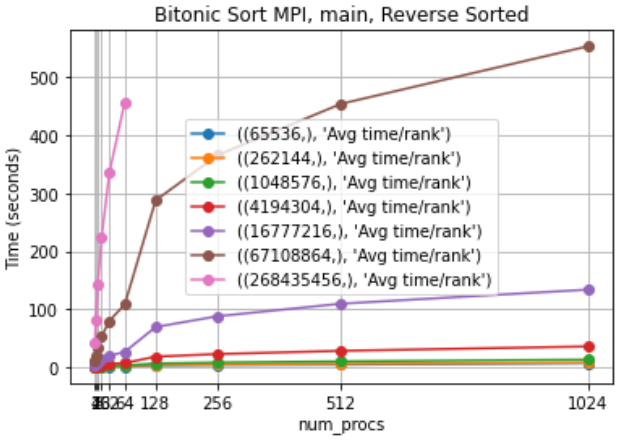
Analysis for strong scaling: The strong scaling graphs are not at all ideal for a bitonic sort algorithm because as the number of threads increases the time should decrease, because with an additional number of threads should make the time decrease. This can be because for a number of reasons, bitonic sort MPI is extremely difficult to scale and this is because of the large amount of communication overhead, there are a lot of data transfers from one array to another causing these graphs in my implementation.

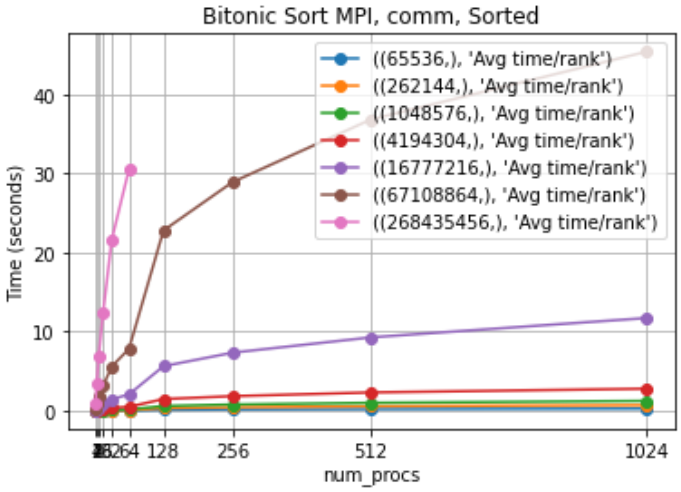
Analysis for weak scaling : The weak scaling graphs for Bitonic MPI are overall what they should look like except for the larger input sizes. This could be due to the algorithm being unable to scale well with the larger number of processors.

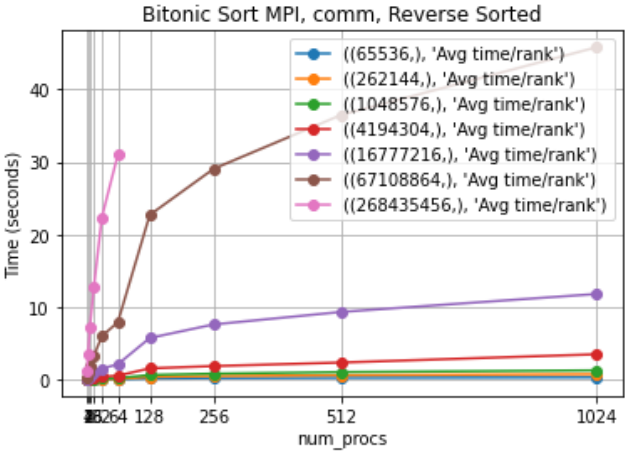
Weak Scaling

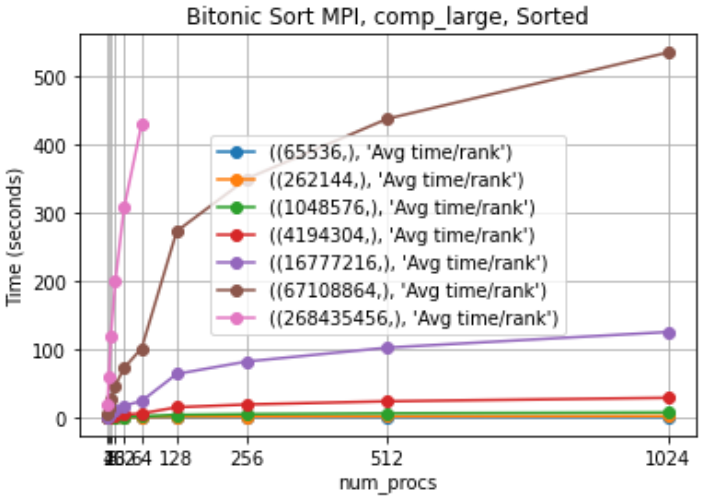
Weak Scaling

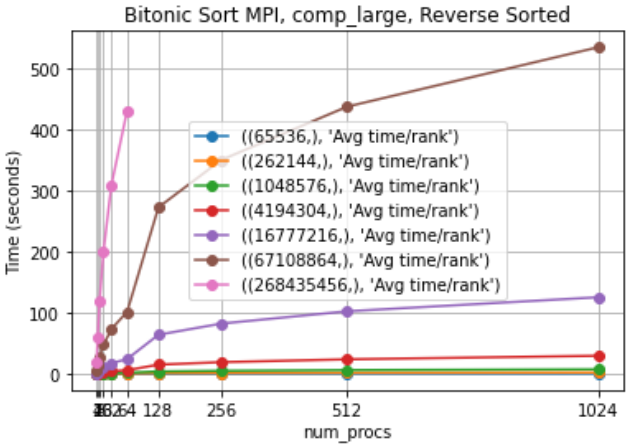


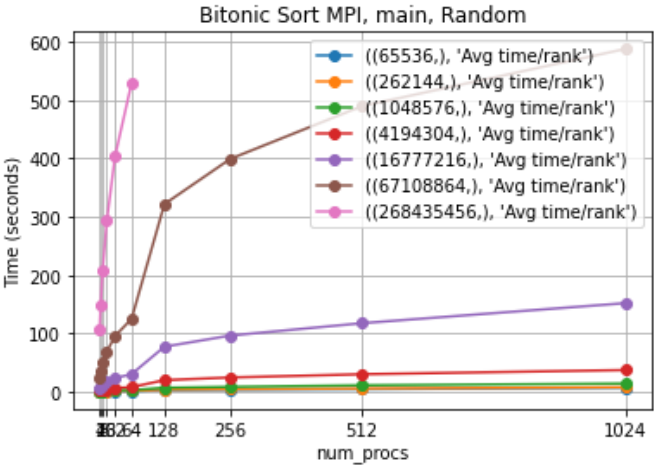


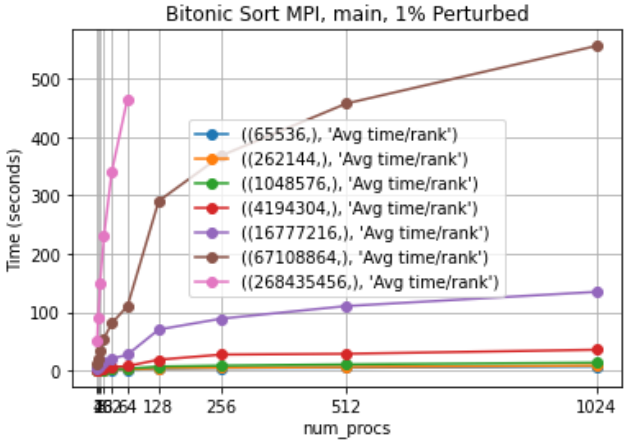


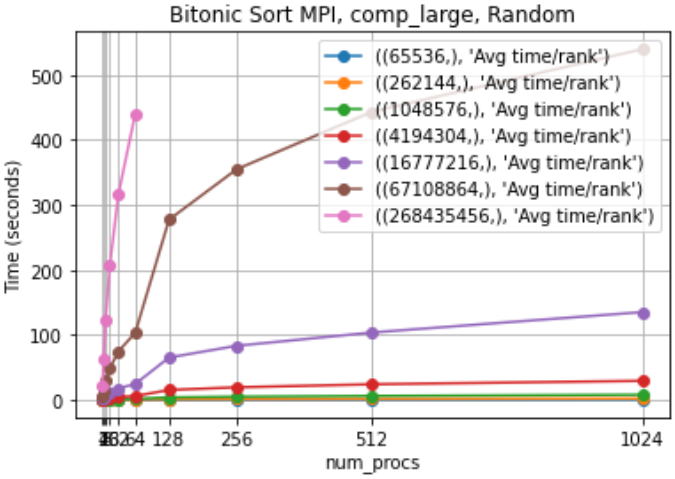


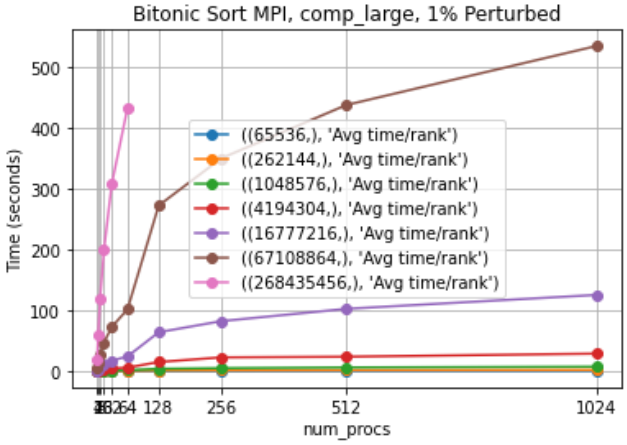


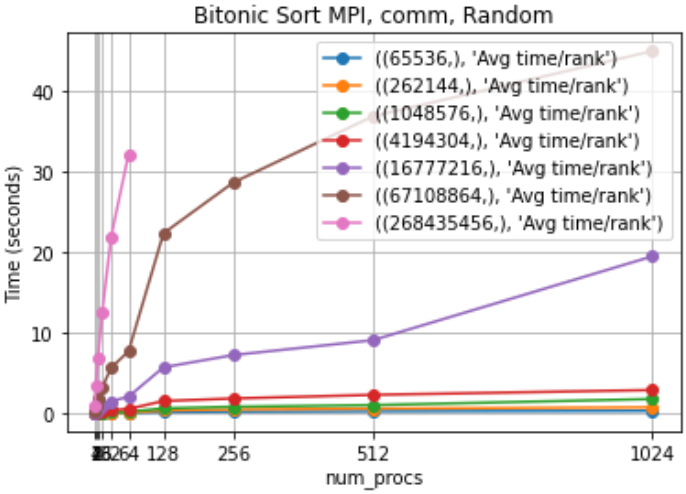


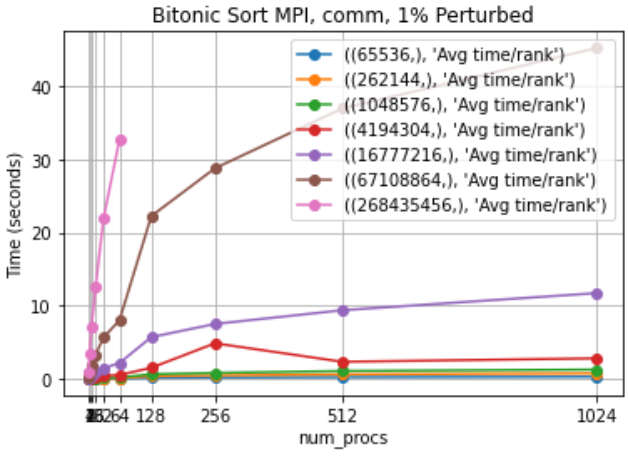

Bitonic Sort MPI, main, Random
Time (seconds)
num_procs
((65536,), 'Avg time/rank')
((262144,), 'Avg time/rank')
((1048576,), 'Avg time/rank')
((4194304,), 'Avg time/rank')
((16777216,), 'Avg time/rank')
((67108864,), 'Avg time/rank')
((268435456,), 'Avg time/rank')


Bitonic Sort MPI, main, 1% Perturbed
Time (seconds)
num_procs
((65536,), 'Avg time/rank')
((262144,), 'Avg time/rank')
((1048576,), 'Avg time/rank')
((4194304,), 'Avg time/rank')
((16777216,), 'Avg time/rank')
((67108864,), 'Avg time/rank')
((268435456,), 'Avg time/rank')

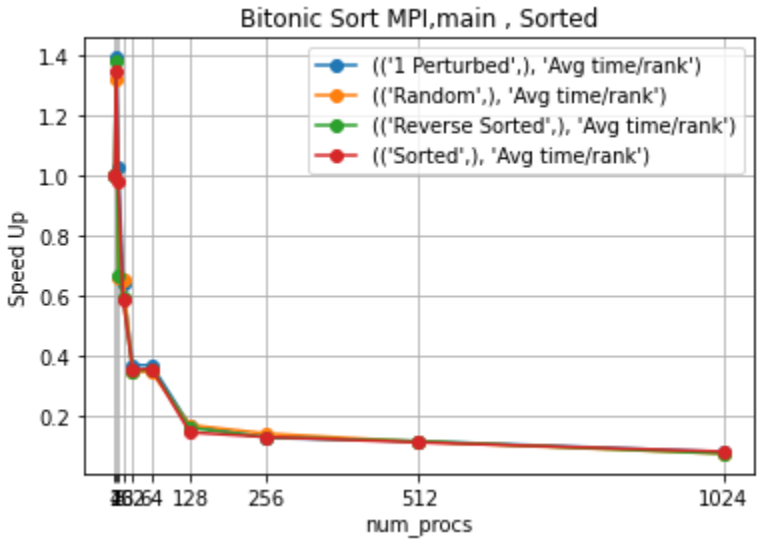

Bitonic Sort MPI, comp_large, Random
Time (seconds)
num_procs
((65536,), 'Avg time/rank')
((262144,), 'Avg time/rank')
((1048576,), 'Avg time/rank')
((4194304,), 'Avg time/rank')
((16777216,), 'Avg time/rank')
((67108864,), 'Avg time/rank')
((268435456,), 'Avg time/rank')

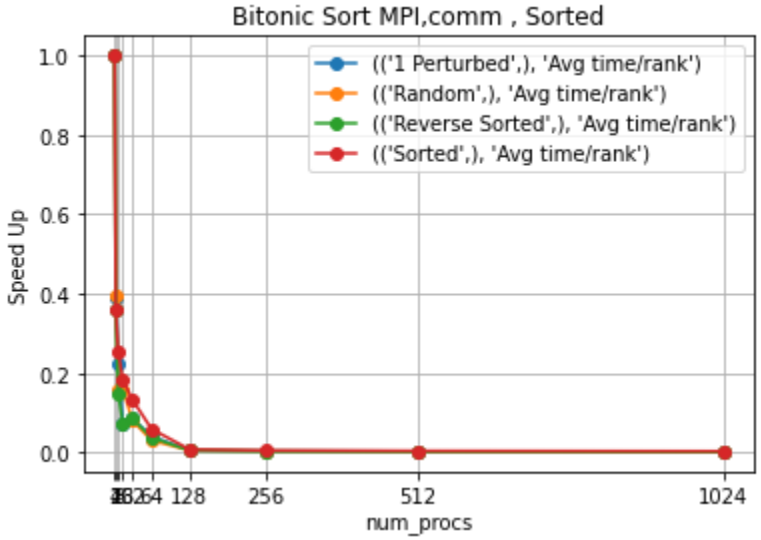

Bitonic Sort MPI, comp_large, 1% Perturbed
Time (seconds)
num_procs
((65536,), 'Avg time/rank')
((262144,), 'Avg time/rank')
((1048576,), 'Avg time/rank')
((4194304,), 'Avg time/rank')
((16777216,), 'Avg time/rank')
((67108864,), 'Avg time/rank')
((268435456,), 'Avg time/rank')

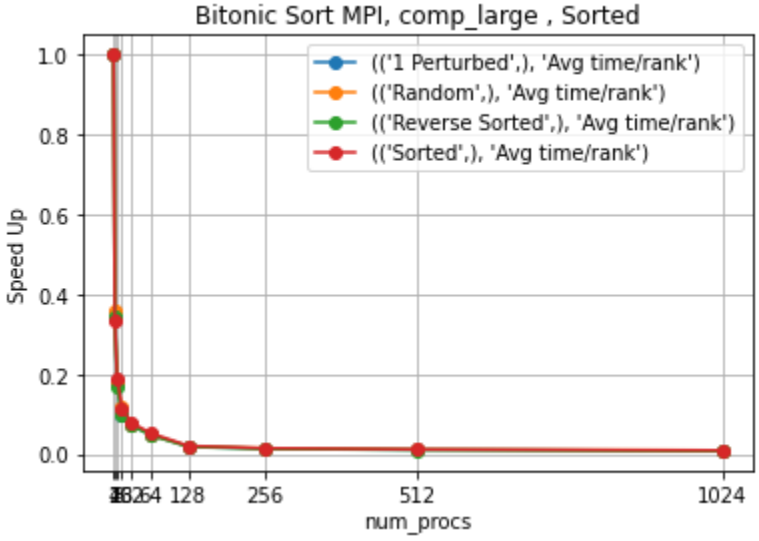

Bitonic Sort MPI, comm, Random
Time (seconds)
num_procs
((65536,), 'Avg time/rank')
((262144,), 'Avg time/rank')
((1048576,), 'Avg time/rank')
((4194304,), 'Avg time/rank')
((16777216,), 'Avg time/rank')
((67108864,), 'Avg time/rank')
((268435456,), 'Avg time/rank')

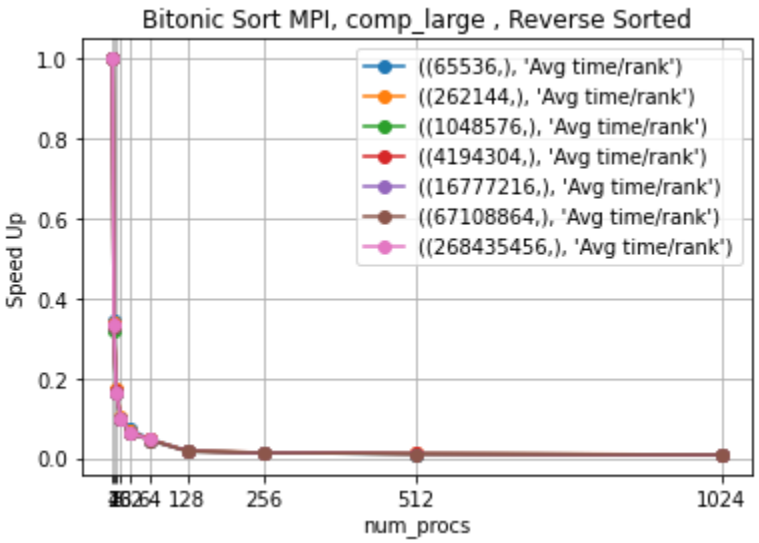

Bitonic Sort MPI, comm, 1% Perturbed
Time (seconds)
num_procs
((65536,), 'Avg time/rank')
((262144,), 'Avg time/rank')
((1048576,), 'Avg time/rank')
((4194304,), 'Avg time/rank')
((16777216,), 'Avg time/rank')
((67108864,), 'Avg time/rank')
((268435456,), 'Avg time/rank')

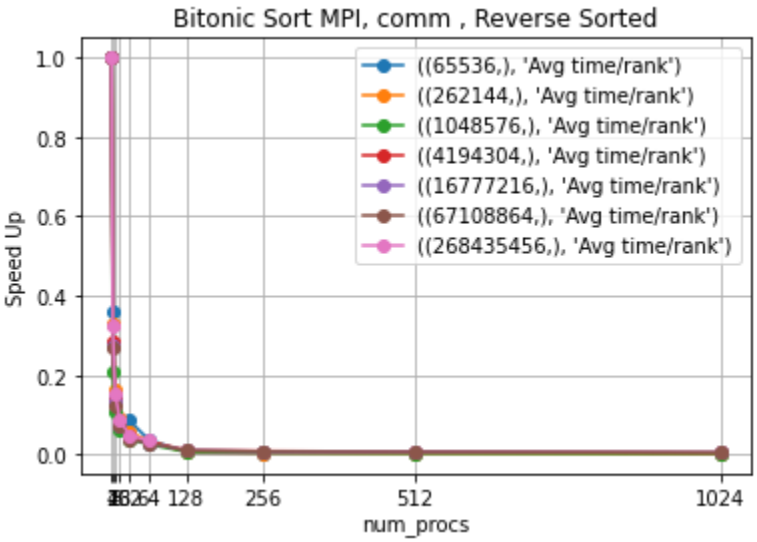
Speed Up

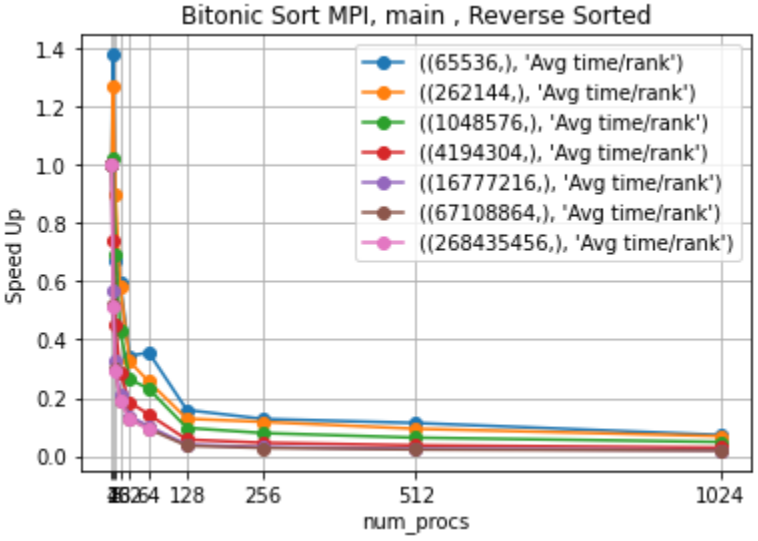

Bitonic Sort MPI,main , Sorted
((1 Perturbed',), 'Avg time/rank')
((Random',), 'Avg time/rank')
((Reverse Sorted',), 'Avg time/rank')
((Sorted',), 'Avg time/rank')
Speed Up
num_procs
1.4
1.2
1.0
0.8
0.6
0.4
0.2
128
256
512
1024

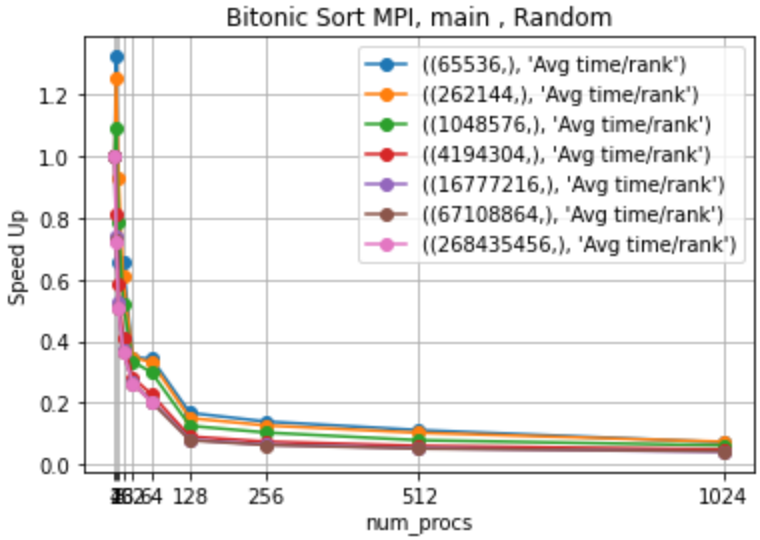

Bitonic Sort MPI,comm , Sorted
((1 Perturbed',), 'Avg time/rank')
((Random',), 'Avg time/rank')
((Reverse Sorted',), 'Avg time/rank')
((Sorted',), 'Avg time/rank')
Speed Up
num_procs
1.0
0.8
0.6
0.4
0.2
0.0
128
256
512
1024

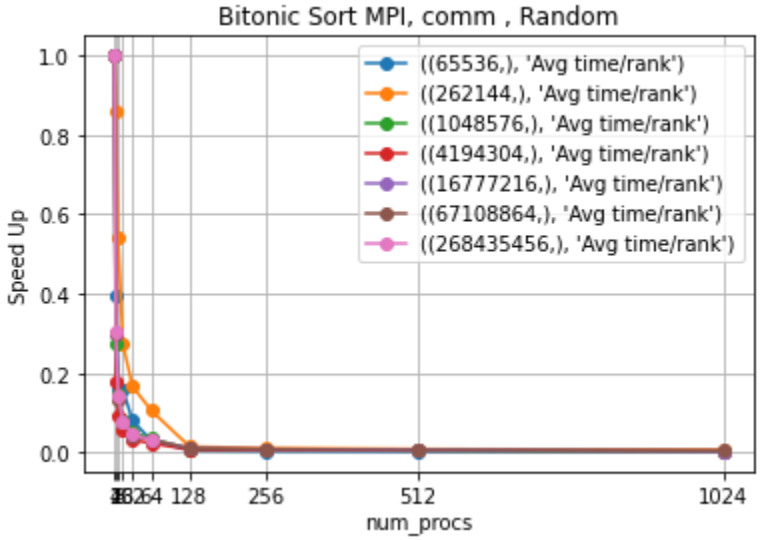

Bitonic Sort MPI, comp_large , Sorted
((1 Perturbed',), 'Avg time/rank')
((Random',), 'Avg time/rank')
((Reverse Sorted',), 'Avg time/rank')
((Sorted',), 'Avg time/rank')
Speed Up
num_procs
1.0
0.8
0.6
0.4
0.2
0.0
128
256
512
1024

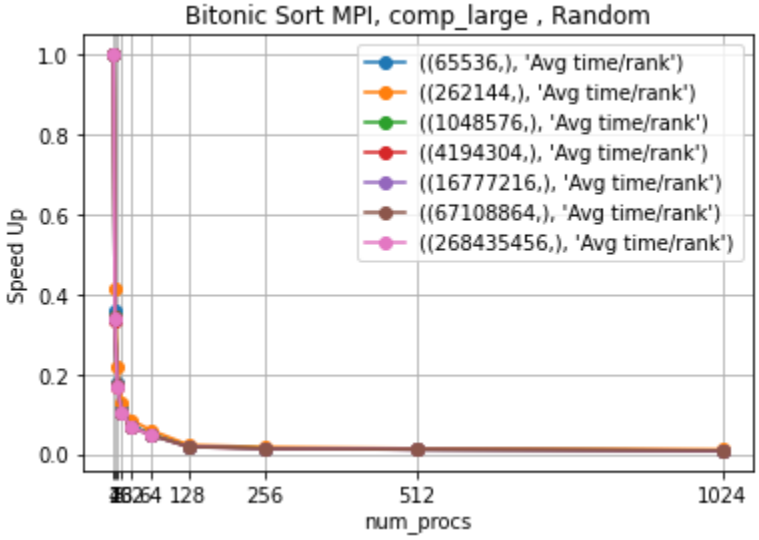

Bitonic Sort MPI, comp_large , Reverse Sorted
Speed Up
num_procs
((65536,), 'Avg time/rank')
((262144,), 'Avg time/rank')
((1048576,), 'Avg time/rank')
((4194304,), 'Avg time/rank')
((16777216,), 'Avg time/rank')
((67108864,), 'Avg time/rank')
((268435456,), 'Avg time/rank')

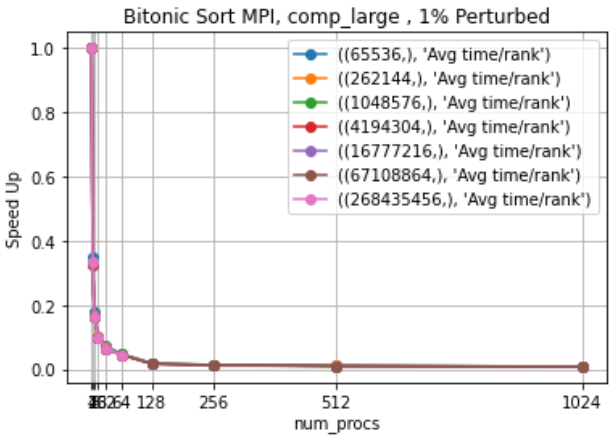

Bitonic Sort MPI, comm , Reverse Sorted
Speed Up
num_procs
((65536,), 'Avg time/rank')
((262144,), 'Avg time/rank')
((1048576,), 'Avg time/rank')
((4194304,), 'Avg time/rank')
((16777216,), 'Avg time/rank')
((67108864,), 'Avg time/rank')
((268435456,), 'Avg time/rank')

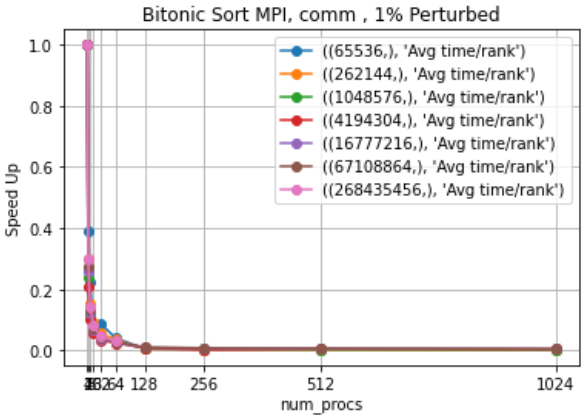

Bitonic Sort MPI, main , Reverse Sorted
Speed Up
num_procs
((65536,), 'Avg time/rank')
((262144,), 'Avg time/rank')
((1048576,), 'Avg time/rank')
((4194304,), 'Avg time/rank')
((16777216,), 'Avg time/rank')
((67108864,), 'Avg time/rank')
((268435456,), 'Avg time/rank')


Bitonic Sort MPI, main , Random
((65536,), 'Avg time/rank')
((262144,), 'Avg time/rank')
((1048576,), 'Avg time/rank')
((4194304,), 'Avg time/rank')
((16777216,), 'Avg time/rank')
((67108864,), 'Avg time/rank')
((268435456,), 'Avg time/rank')
Speed Up
num_procs

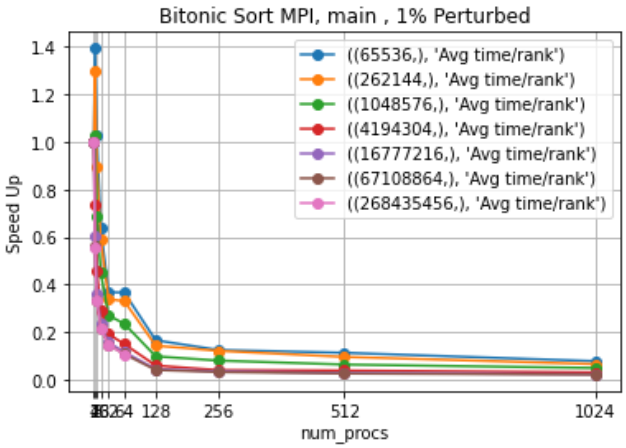

Bitonic Sort MPI, comm , Random
((65536,), 'Avg time/rank')
((262144,), 'Avg time/rank')
((1048576,), 'Avg time/rank')
((4194304,), 'Avg time/rank')
((16777216,), 'Avg time/rank')
((67108864,), 'Avg time/rank')
((268435456,), 'Avg time/rank')
Speed Up
num_procs


Bitonic Sort MPI, comp_large , Random
((65536,), 'Avg time/rank')
((262144,), 'Avg time/rank')
((1048576,), 'Avg time/rank')
((4194304,), 'Avg time/rank')
((16777216,), 'Avg time/rank')
((67108864,), 'Avg time/rank')
((268435456,), 'Avg time/rank')
Speed Up
num_procs





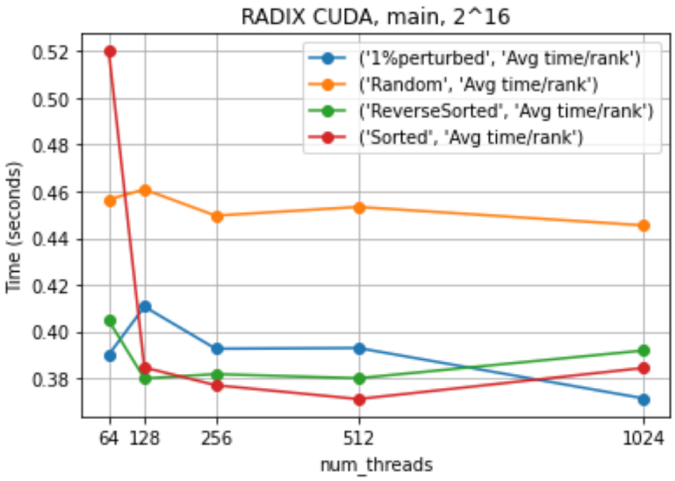
Analysis:

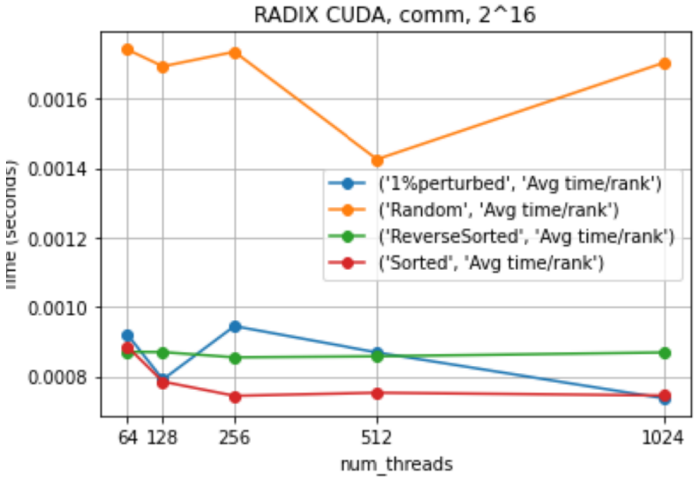


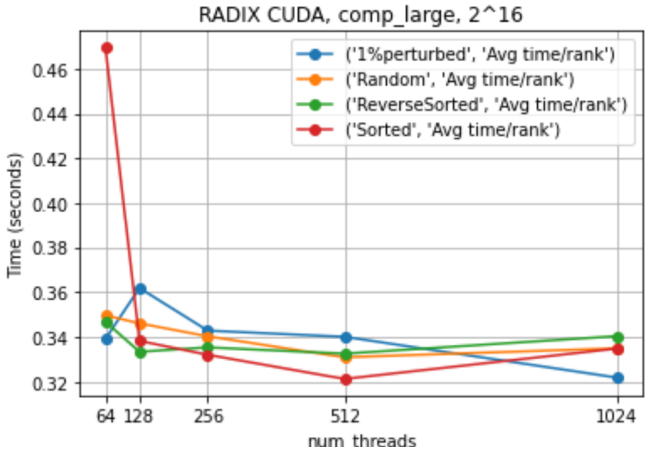
The speedup graphs for Bitonic sort MPI are not ideal. All the graphs for all the sorting algorithms seem to all decrease exponentially. This shows that the algorithm did not scale very well and adding more and more processes did not help improve the algorithm's performance.

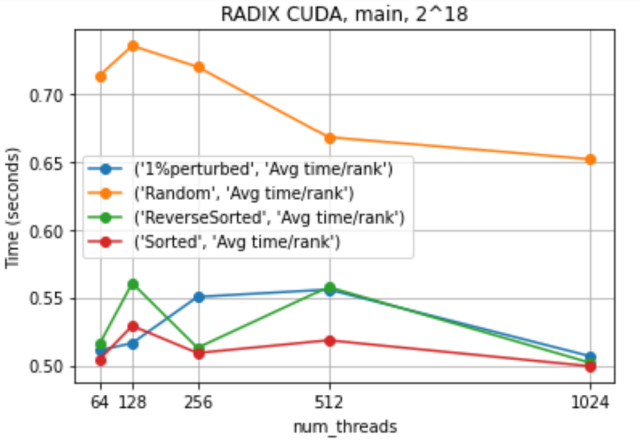
RADIX CUDA

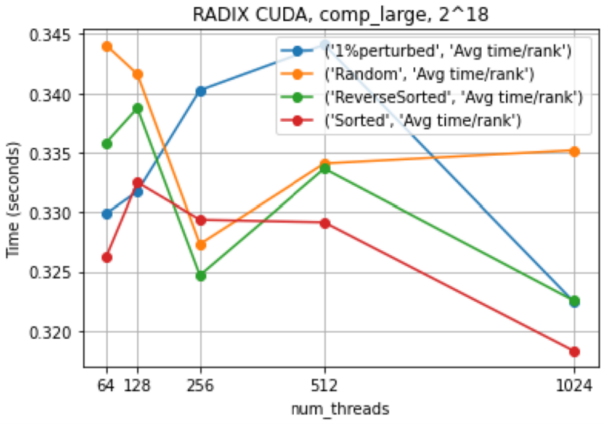
Strong scaling

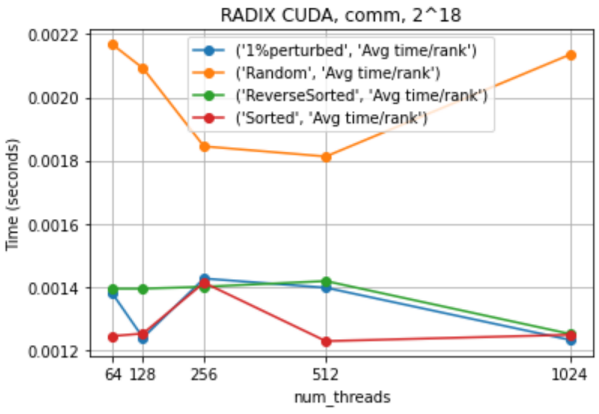


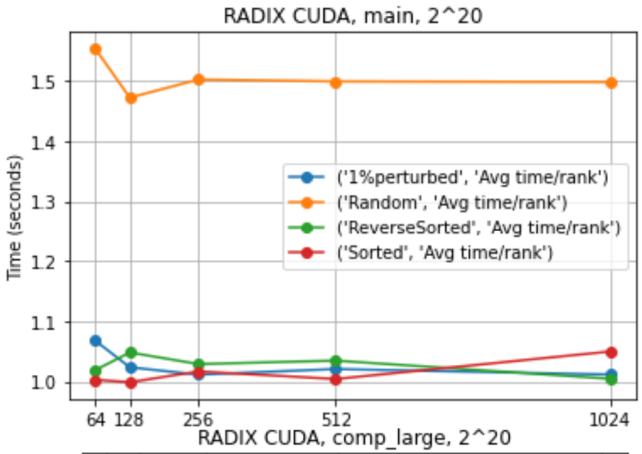


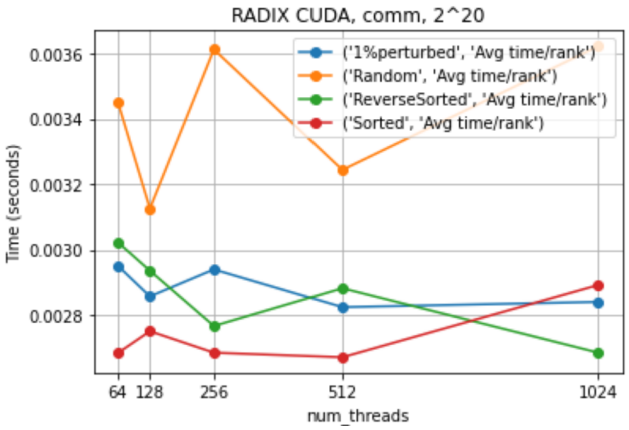


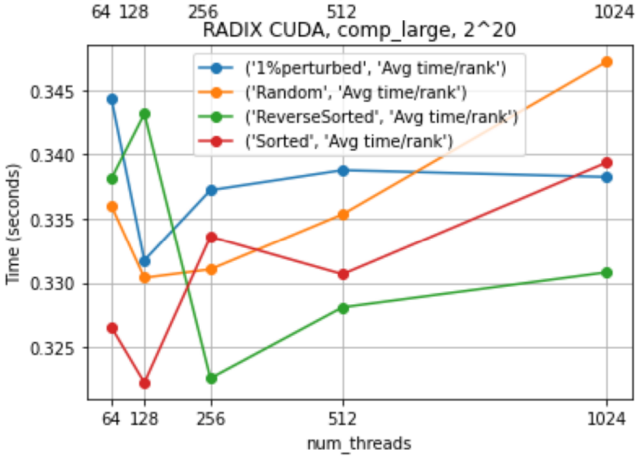


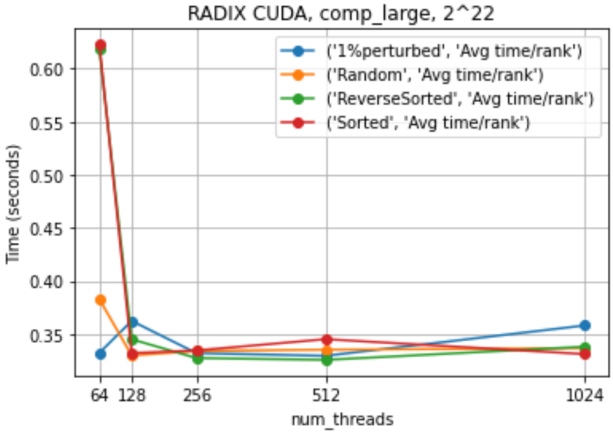


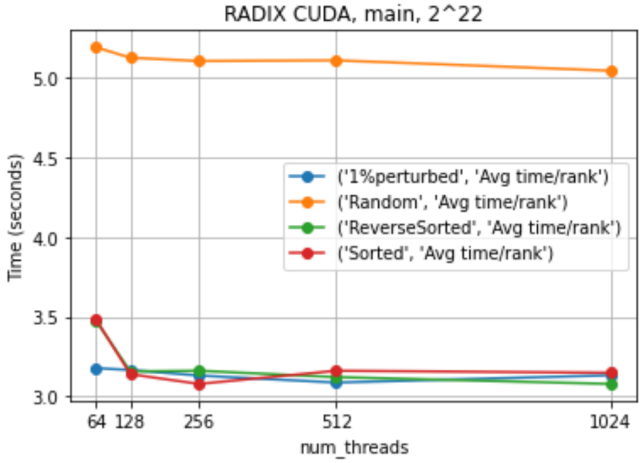


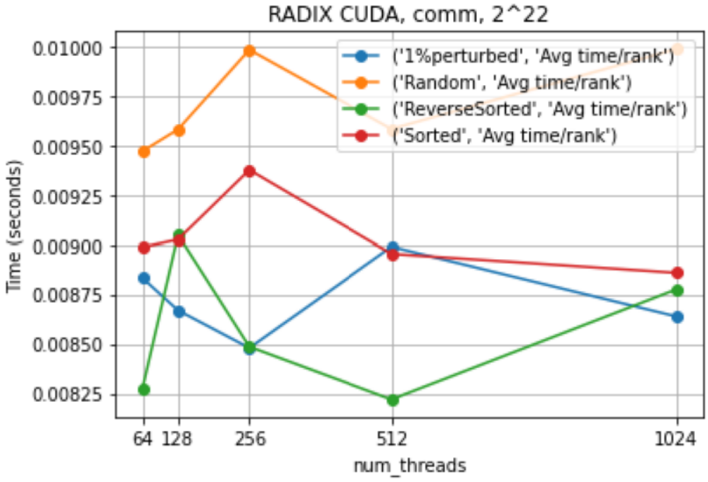

RADIX CUDA, main, 2^20
Time (seconds)
1.5
1.4
1.3
1.2
1.1
1.0
64 128 256 512 1024
('1%perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')

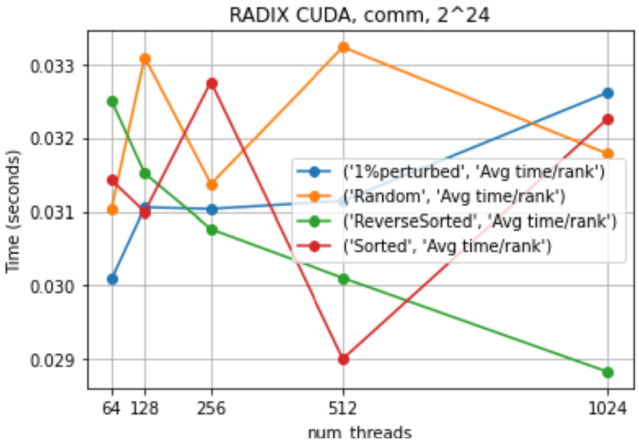

RADIX CUDA, comm, 2^20
Time (seconds)
0.0036
0.0034
0.0032
0.0030
0.0028
64 128 256 512 1024
num_threads
('1%perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')

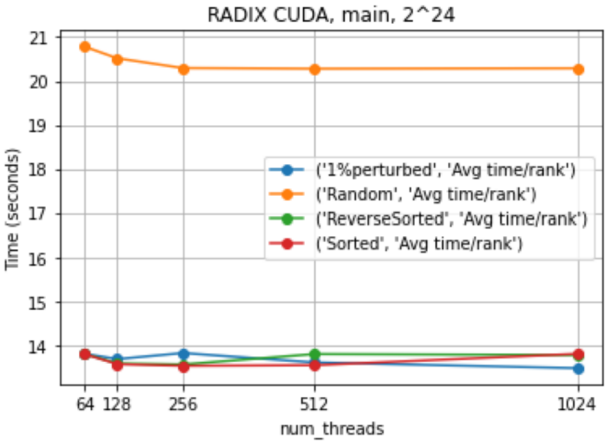

RADIX CUDA, comp_large, 2^20
Time (seconds)
0.345
0.340
0.335
0.330
0.325
64 128 256 512 1024
num_threads
('1%perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')

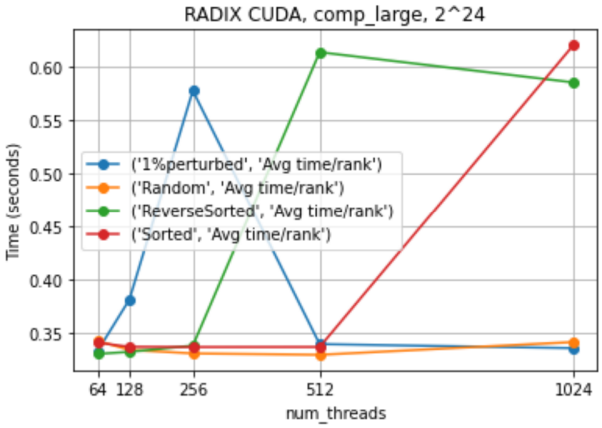

RADIX CUDA, comp_large, 2^22
Time (seconds)
0.60
0.55
0.50
0.45
0.40
0.35
64 128 256 512 1024
num_threads
('1%perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')

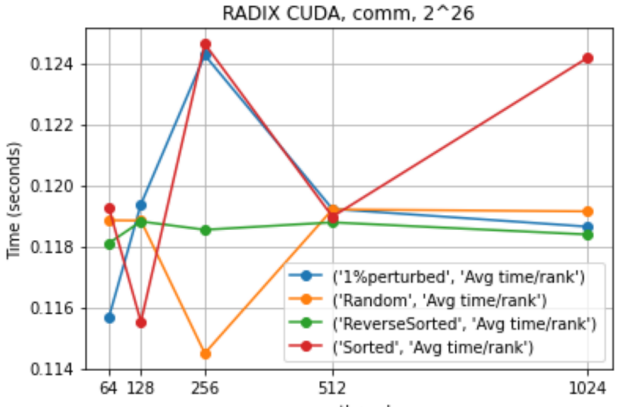

RADIX CUDA, main, 2^22
Time (seconds)
5.0
4.5
4.0
3.5
3.0
64 128 256 512 1024
num_threads
('1%perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')

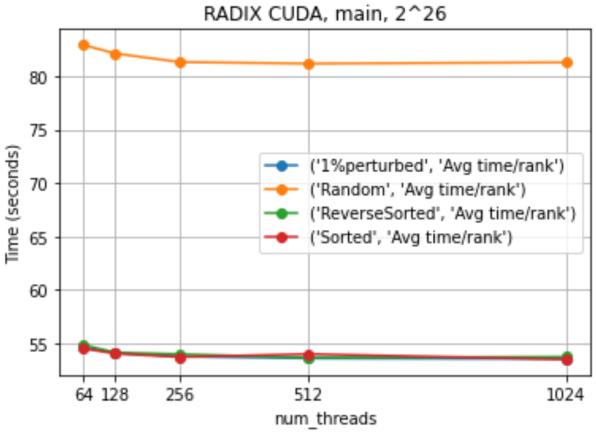

RADIX CUDA, comm, 2^22
Time (seconds)
0.01000
0.00975
0.00950
0.00925
0.00900
0.00875
0.00850
0.00825
64 128 256 512 1024
num_threads
('1%perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')

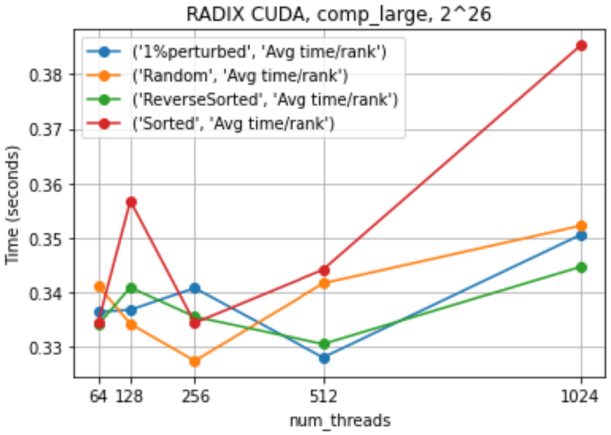

RADIX CUDA, comm, 2^24
Time (seconds)
0.033
0.032
0.031
0.030
0.029
64 128 256 512 1024
num_threads
('1%perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')

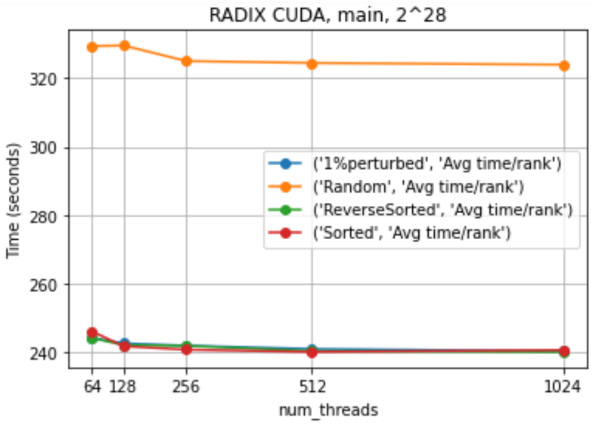

RADIX CUDA, main, 2^24
Time (seconds)
21
20
19
18
17
16
15
14
64 128 256 512 1024
num_threads
('1%perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')

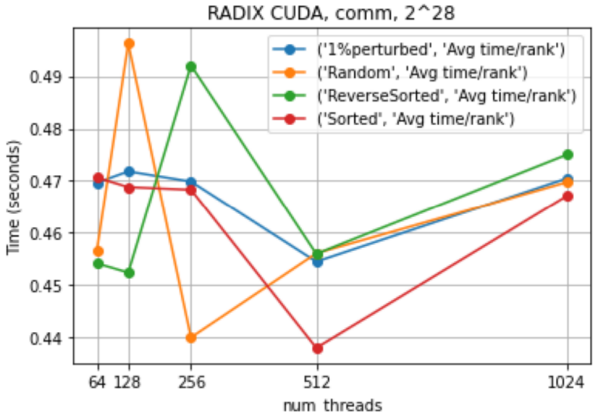

RADIX CUDA, comp_large, 2^24
Time (seconds)
num_threads
('1%perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')

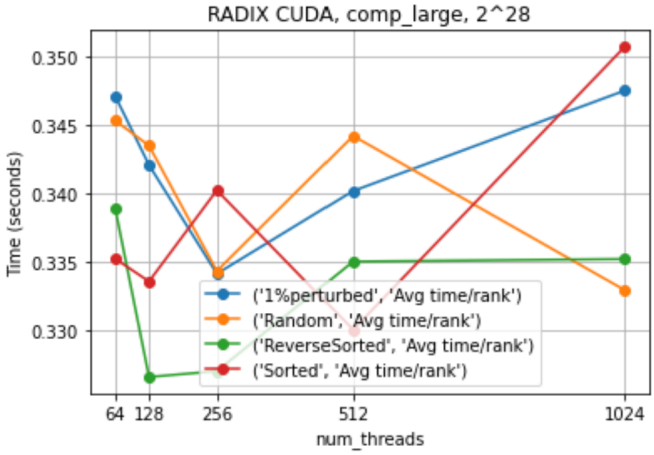

RADIX CUDA, comm, 2^26
Time (seconds)
('1%perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')


RADIX CUDA, main, 2^26
Time (seconds)
num_threads
('1%perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')


RADIX CUDA, comp_large, 2^26
Time (seconds)
num_threads
('1%perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')

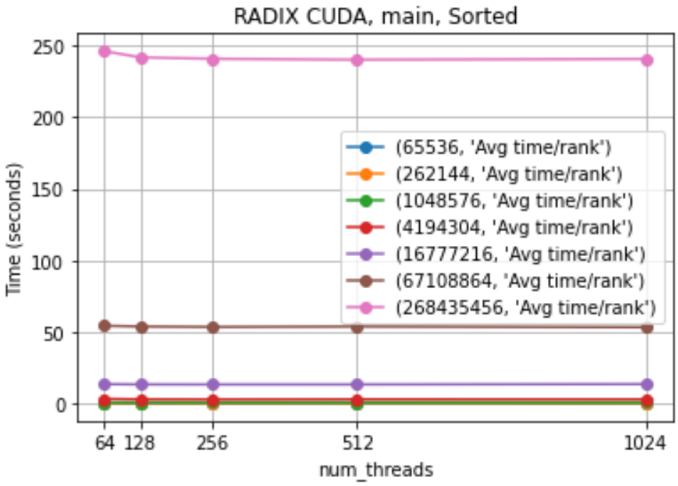

RADIX CUDA, main, 2^28
Time (seconds)
num_threads
('1%perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')

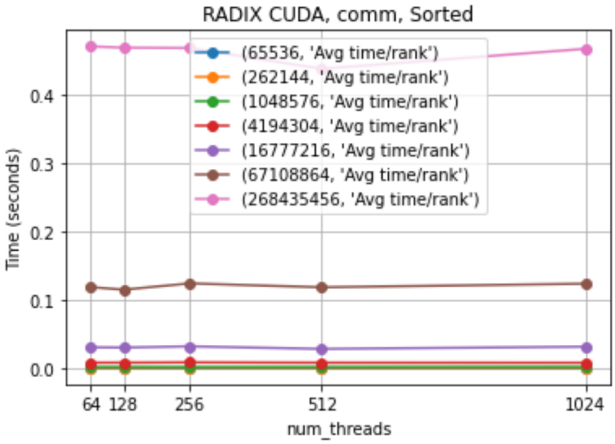

RADIX CUDA, comm, 2^28
Time (seconds)
num_threads
('1%perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')

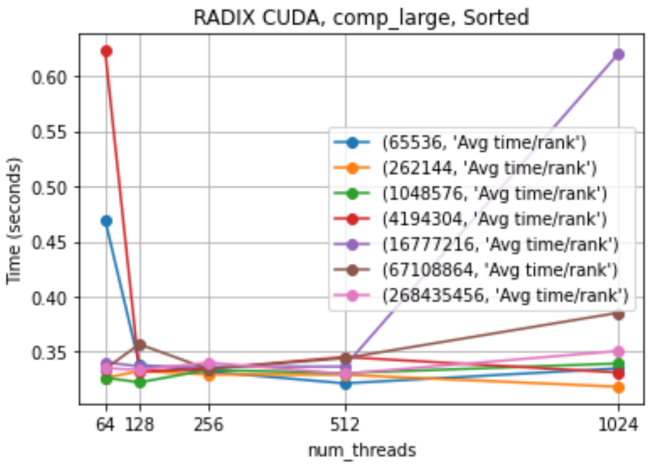

RADIX CUDA, comp_large, 2^28
Time (seconds)
num_threads
('1%perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')

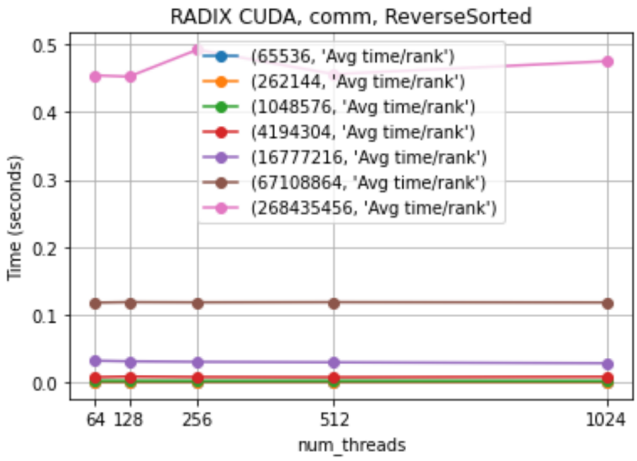
For the Radix Cuda strong scaling, you can observe that across most input sizes, the communication for the random array sorting took more time than the other three input types. This is especially clear in the main graphs, where the random input type took longer than all the others, though this is also likely due to the data initialization. Some of the communication graphs end up increasing their time along with the number of processors, this is most likely due to communication overhead. The strong scaling for computation appears to work as expected for smaller input sizes, but once it passes about 2^18, it gets more random looking as it does not scale well on large inputs.

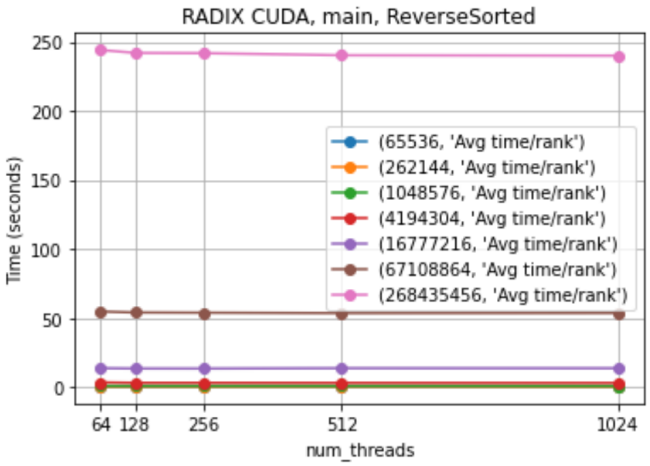
Weak Scaling

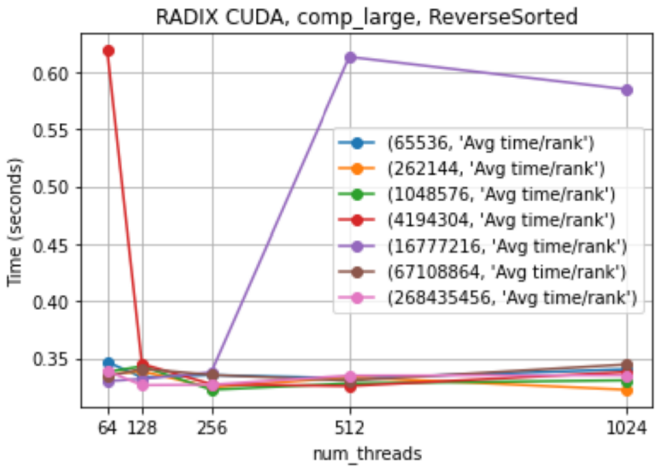


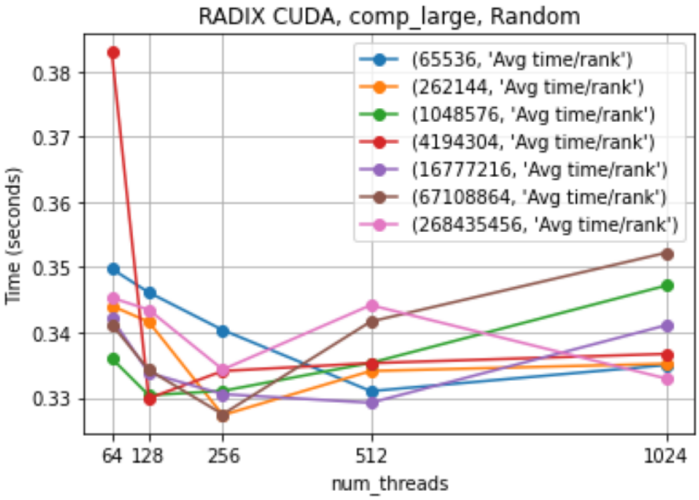


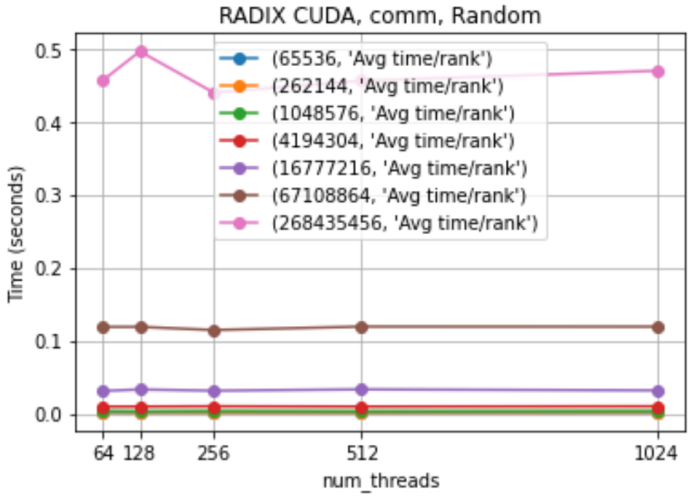


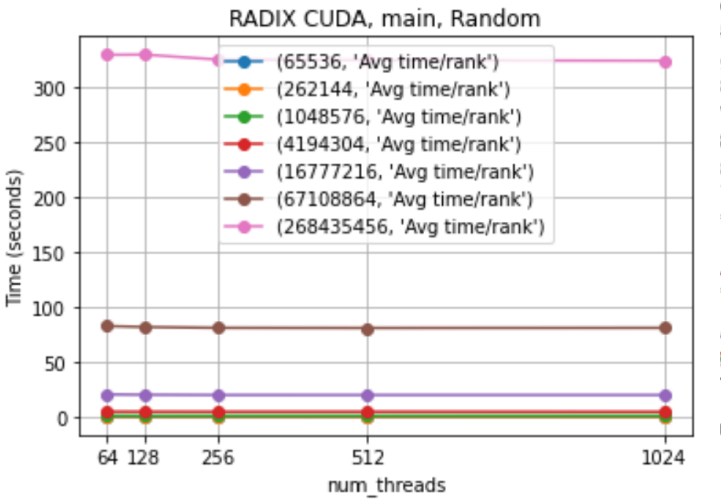


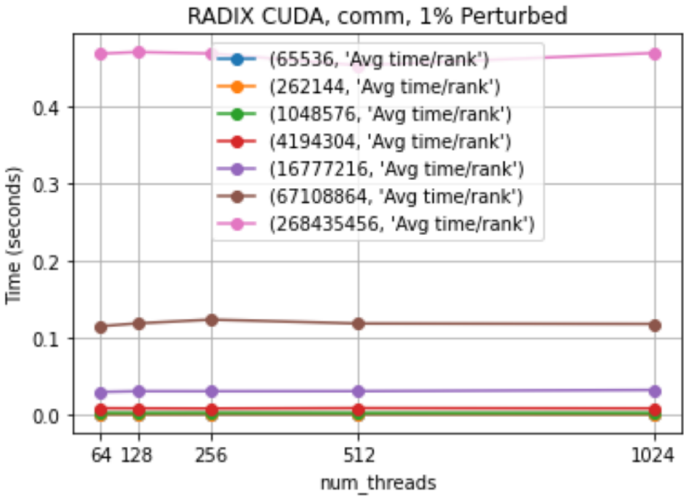

RADIX CUDA, main, ReverseSorted
Time (seconds)
num_threads
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')

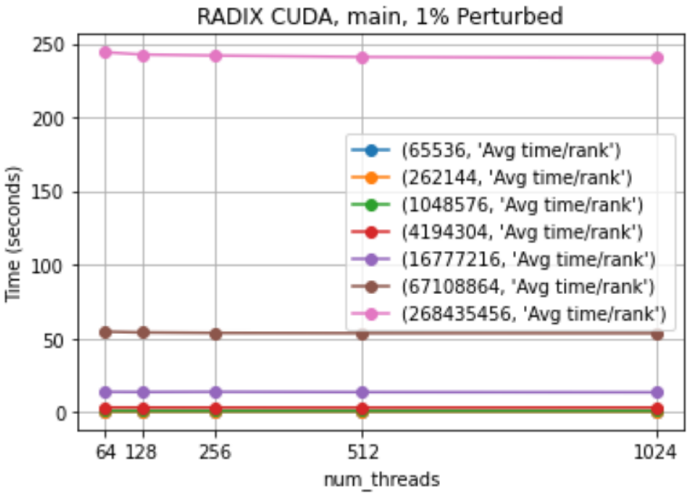

RADIX CUDA, comp_large, ReverseSorted
Time (seconds)
num_threads
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')

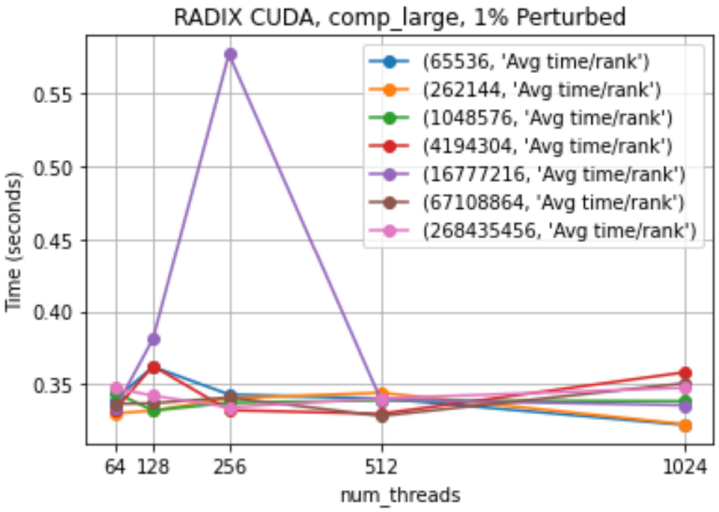

RADIX CUDA, comp_large, Random
Time (seconds)
num_threads
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')


RADIX CUDA, comm, Random
Time (seconds)
num_threads
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')


RADIX CUDA, main, Random
Time (seconds)
num_threads
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')

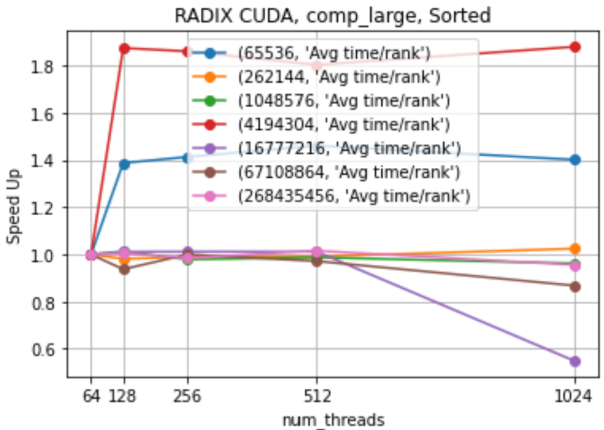

RADIX CUDA, comm, 1% Perturbed
Time (seconds)
num_threads
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')

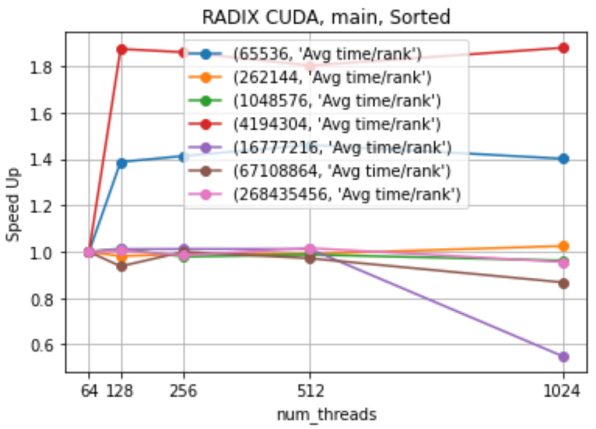


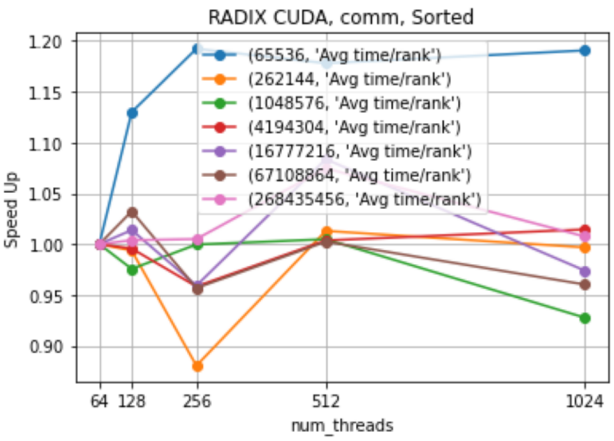


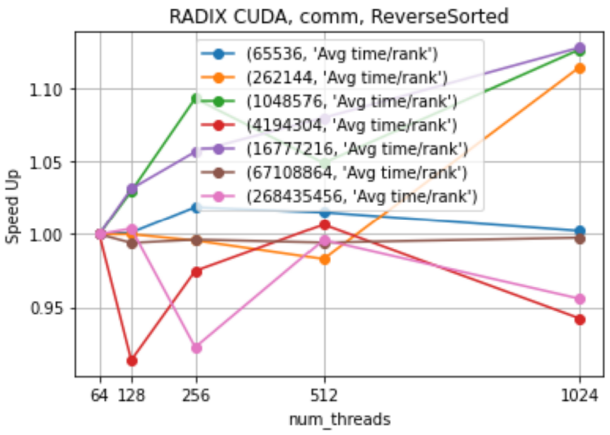
The weak scaling graphs for Radix Cuda are overall what they should look like. The communication and main graphs are fairly level or constant, which is the best case scenario for weak scaling. This may be because my algorithm does not need as much communication or synchronization between threads.This is because there is good scalability within my communication and overall algorithm.The computation graphs have slight spikes which is to be expected since they were a little all over the place in the strong scaling as well, however this does not drastically affect the main graphs, which are still constant.

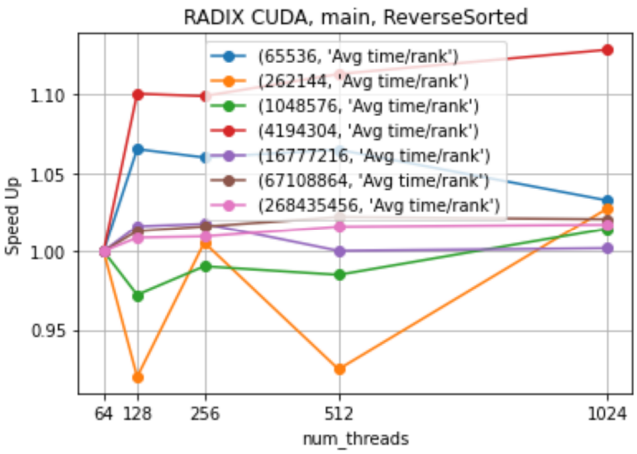
SPEEDUP

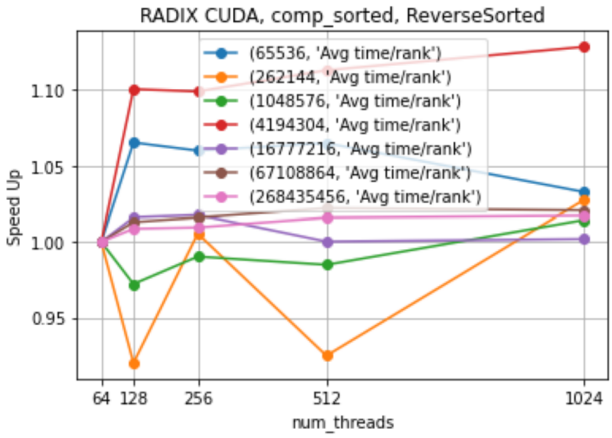


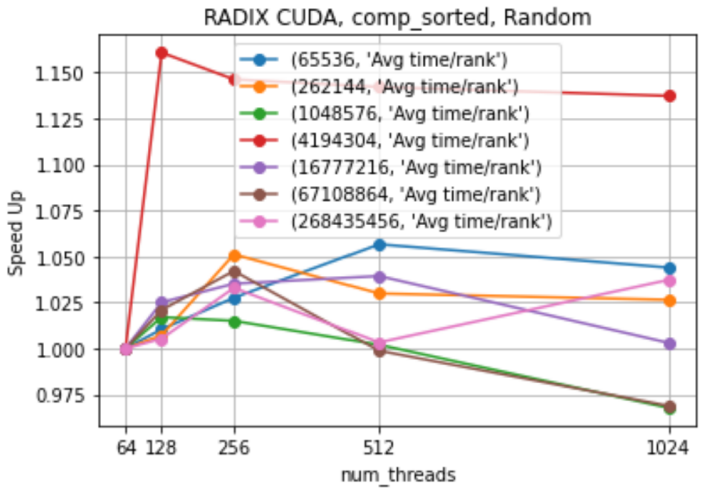


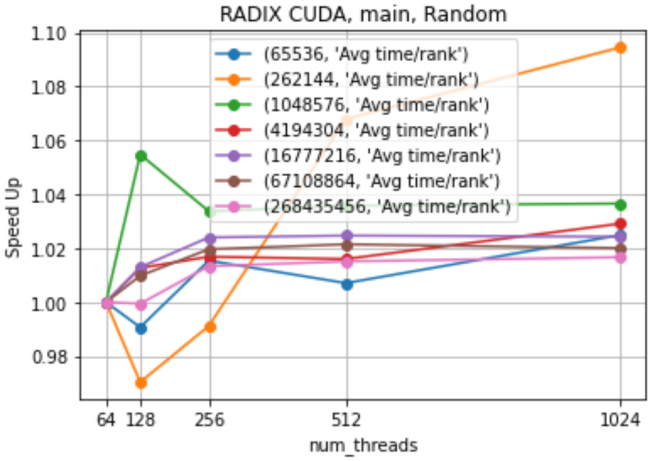


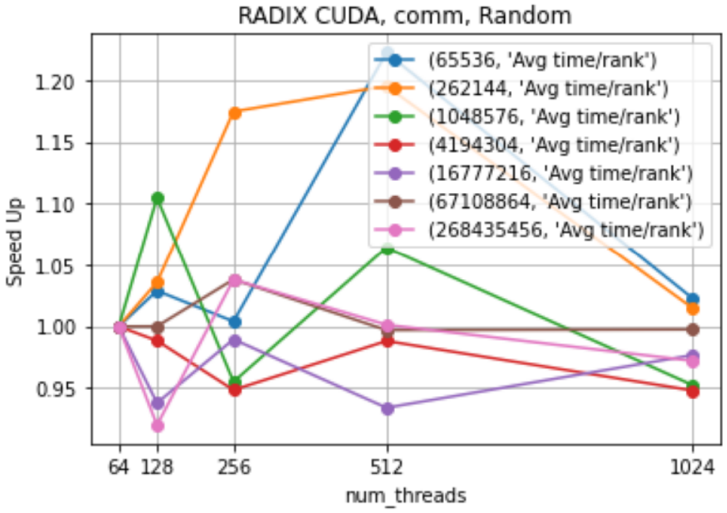


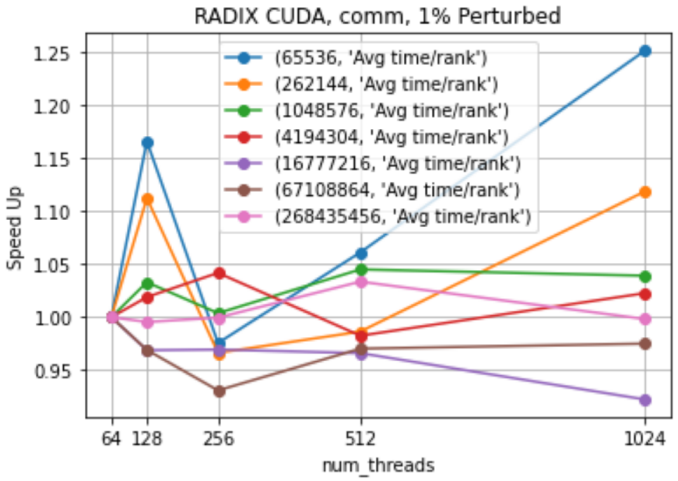

RADIX CUDA, main, ReverseSorted
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')
Speed Up
1.10
1.05
1.00
0.95
64 128 256 512 1024
num_threads

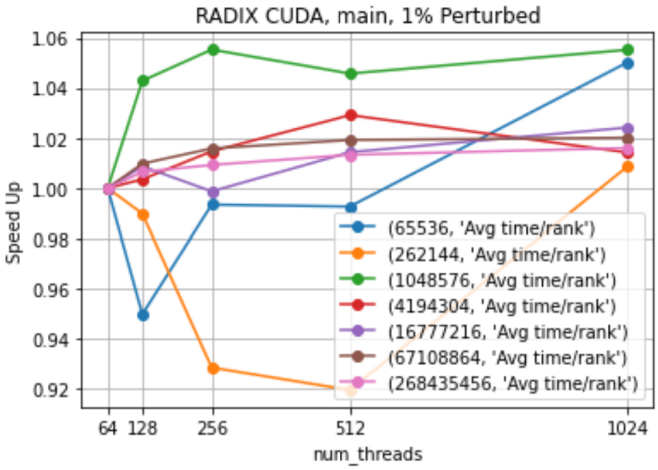

RADIX CUDA, comp_sorted, ReverseSorted
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')
Speed Up
1.10
1.05
1.00
0.95
64 128 256 512 1024
num_threads

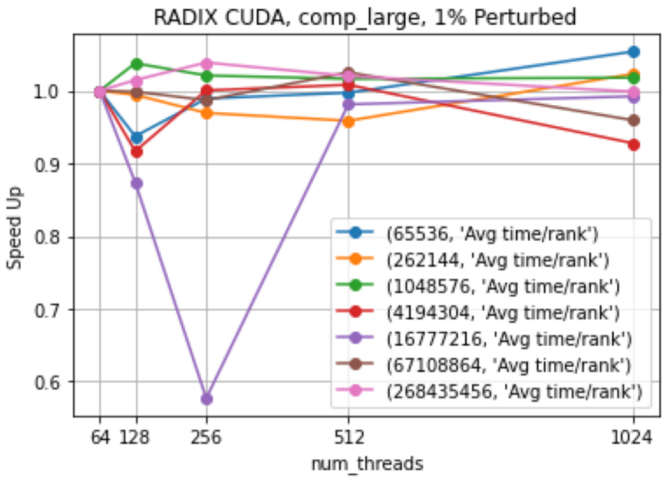

RADIX CUDA, comp_sorted, Random
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')
Speed Up
1.150
1.125
1.100
1.075
1.050
1.025
1.000
0.975
64 128 256 512 1024
num_threads


RADIX CUDA, main, Random
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')
Speed Up
1.10
1.08
1.06
1.04
1.02
1.00
0.98
64 128 256 512 1024
num_threads


RADIX CUDA, comm, Random
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')
Speed Up
1.20
1.15
1.10
1.05
1.00
0.95
64 128 256 512 1024
num_threads


RADIX CUDA, comm, 1% Perturbed
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')
Speed Up
1.25
1.20
1.15
1.10
1.05
1.00
0.95
64 128 256 512 1024
num_threads

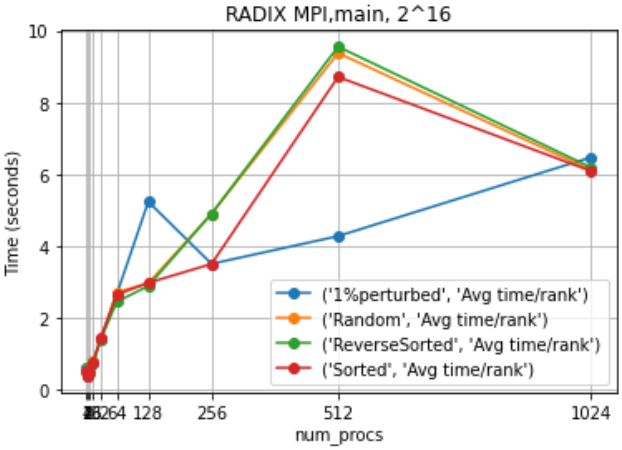


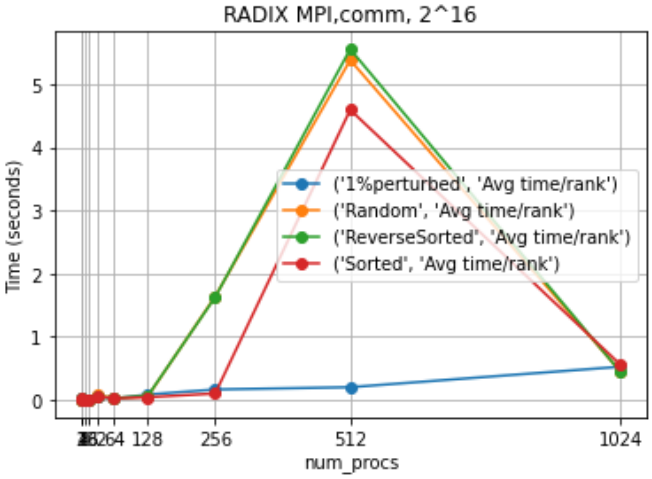


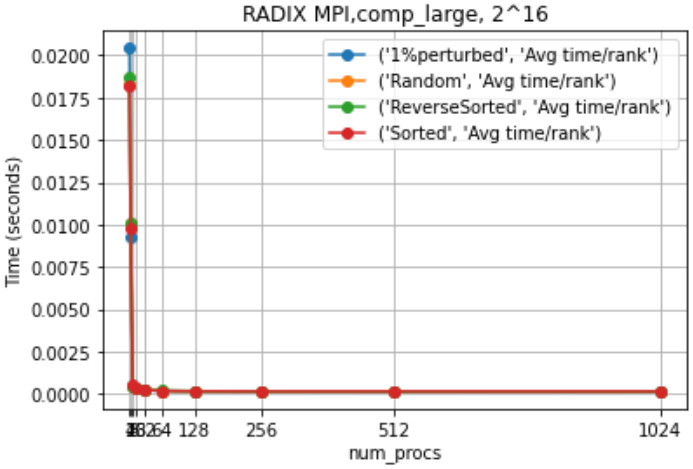
The speedup graphs for Radix Cuda are not ideal. The computation and main ones increase at first, but then seem to plateau after about 128 threads with some even decreasing. The communication ones on the other hand are more all over the place. This shows that the algorithm did not scale very well and adding more and more threads did not help improve the algorithm's performance.

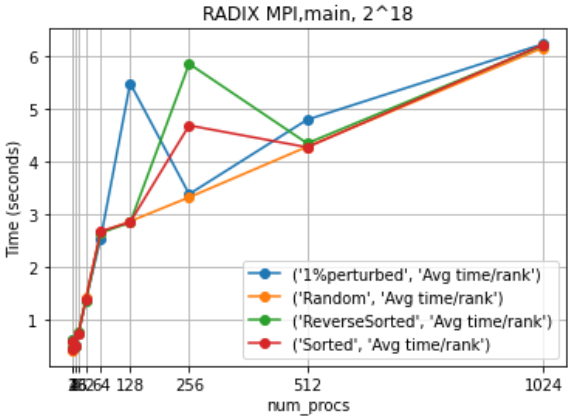
RADIX MPI

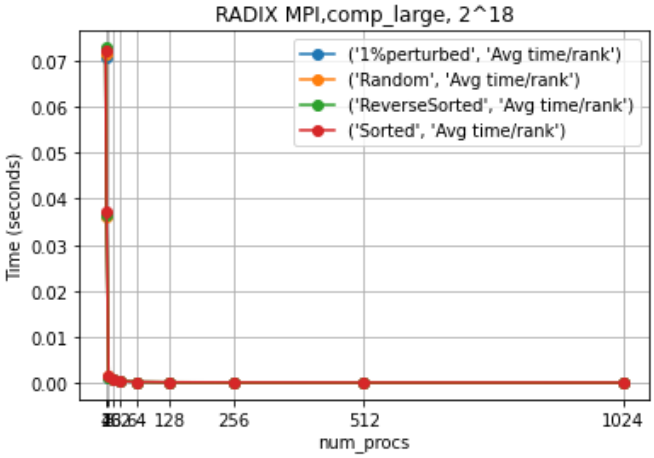
Strong Scaling

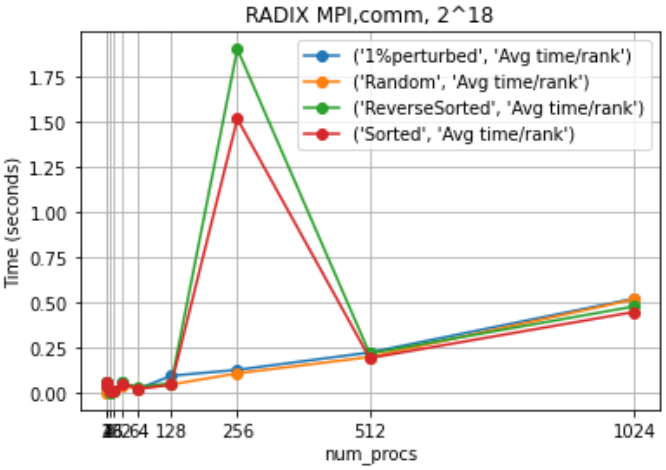


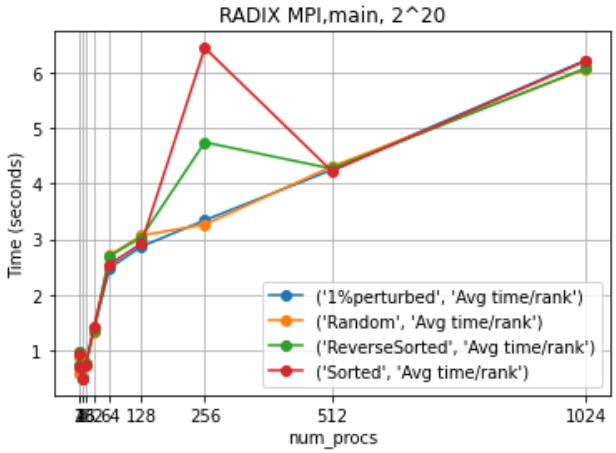


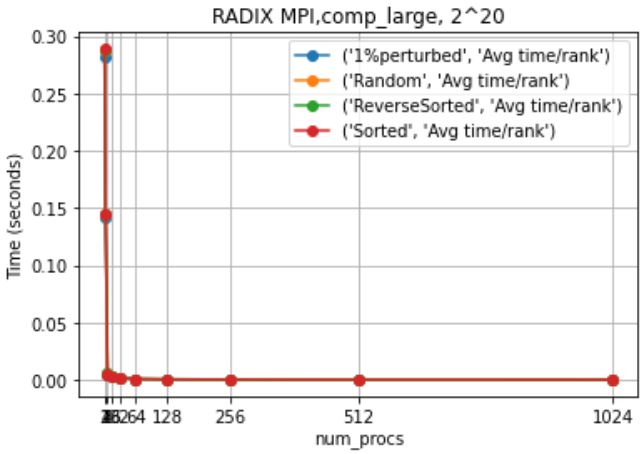


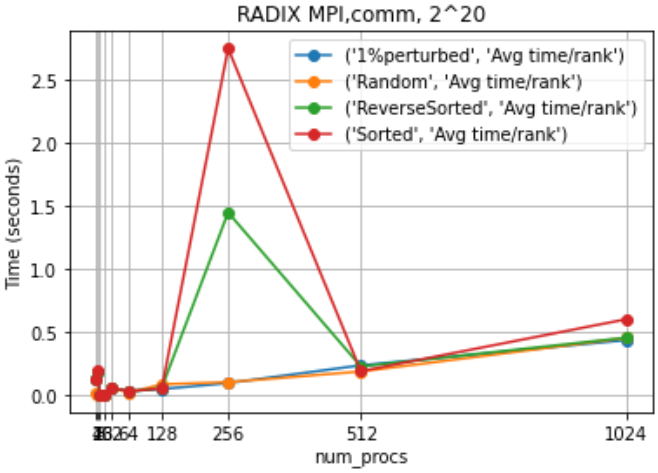


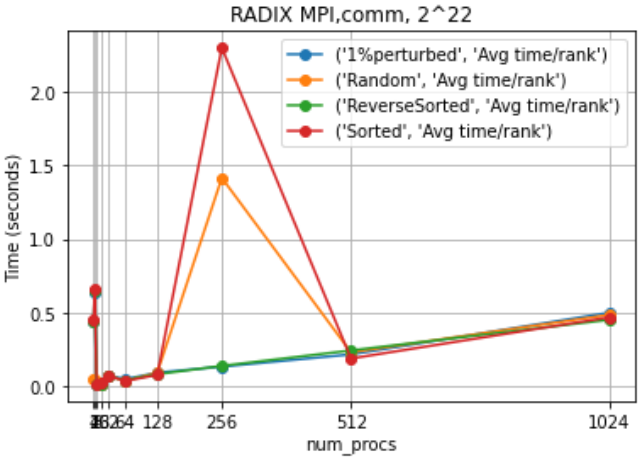


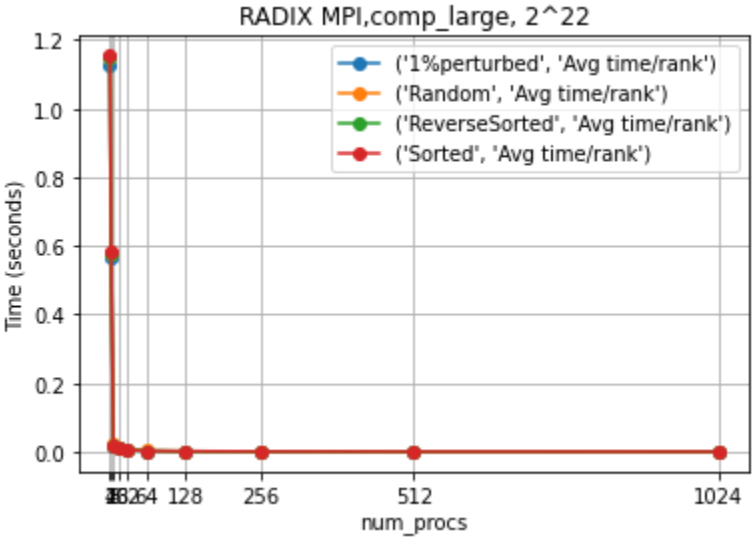


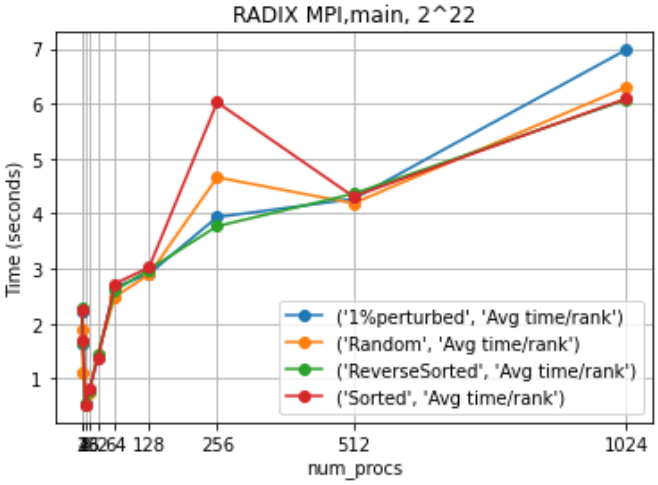

RADIX MPI,main, 2^20
Time (seconds)
num_procs
('1%perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')

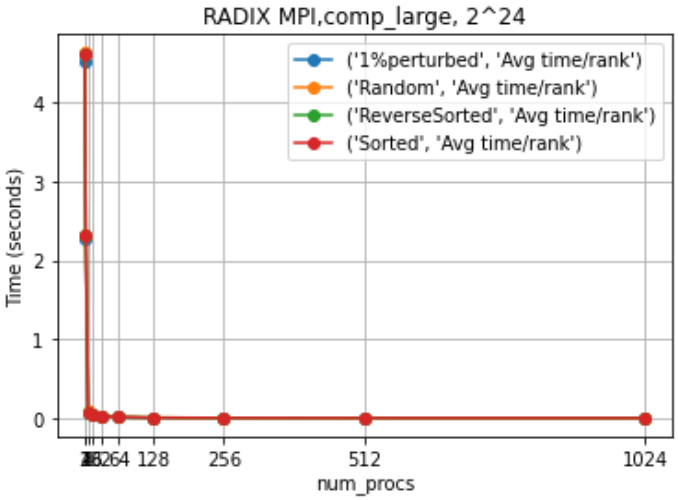

RADIX MPI,comp_large, 2^20
Time (seconds)
num_procs
('1%perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')

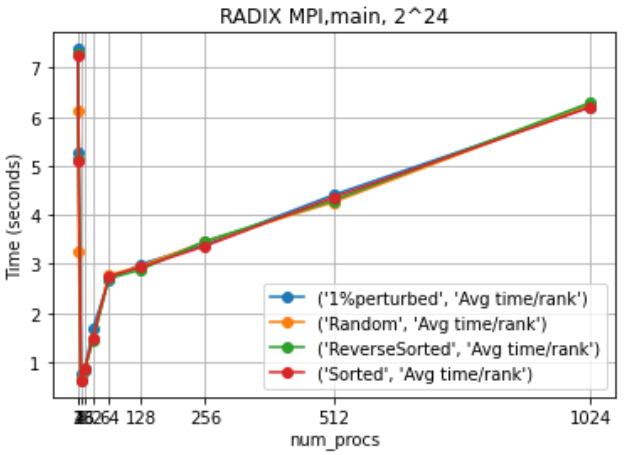

RADIX MPI,comm, 2^20
Time (seconds)
num_procs
('1%perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')

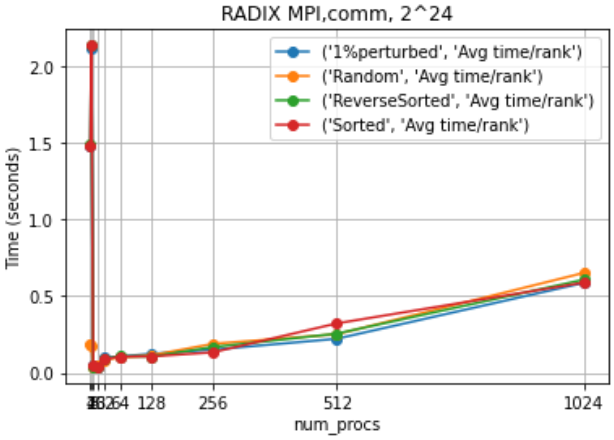

RADIX MPI,comm, 2^22
Time (seconds)
num_procs
('1%perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')

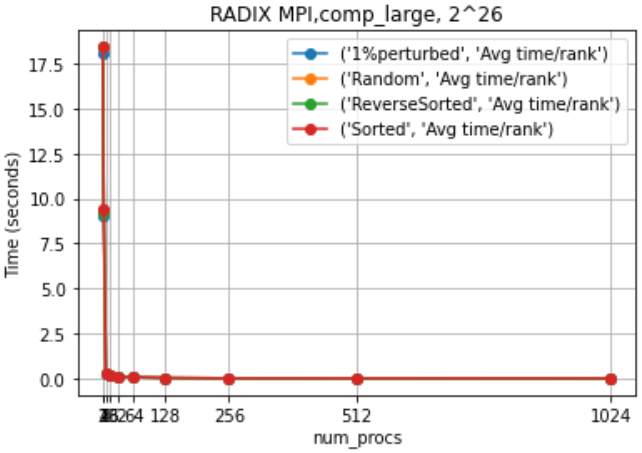

RADIX MPI,comp_large, 2^22
Time (seconds)
num_procs
('1%perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')

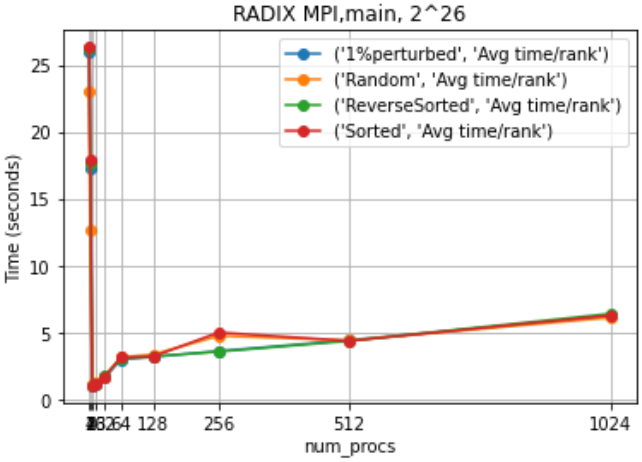

RADIX MPI,main, 2^22
Time (seconds)
num_procs
('1%perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')

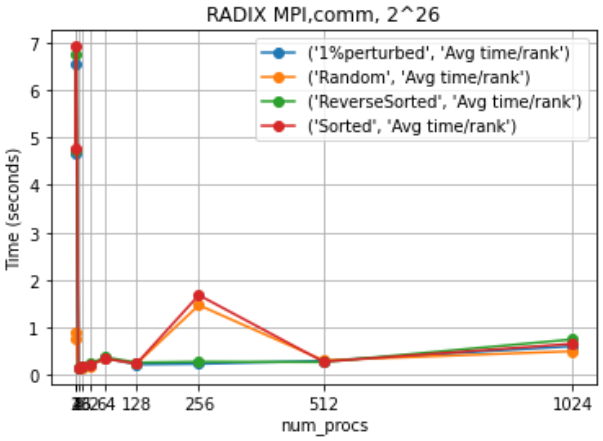

RADIX MPI,comp_large, 2^24
('1%perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')
Time (seconds)
num_procs

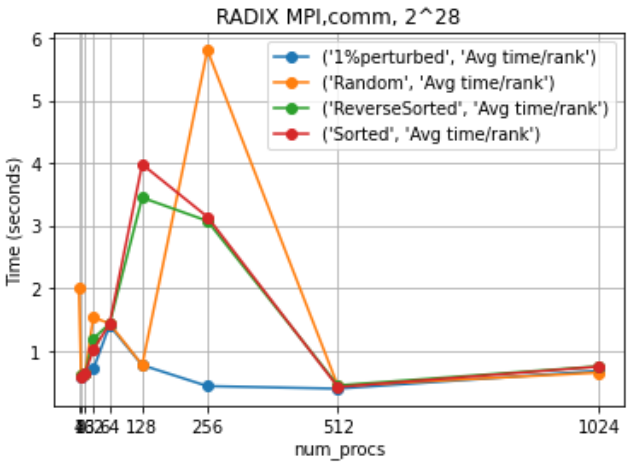

RADIX MPI,main, 2^24
('1%perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')
Time (seconds)
num_procs

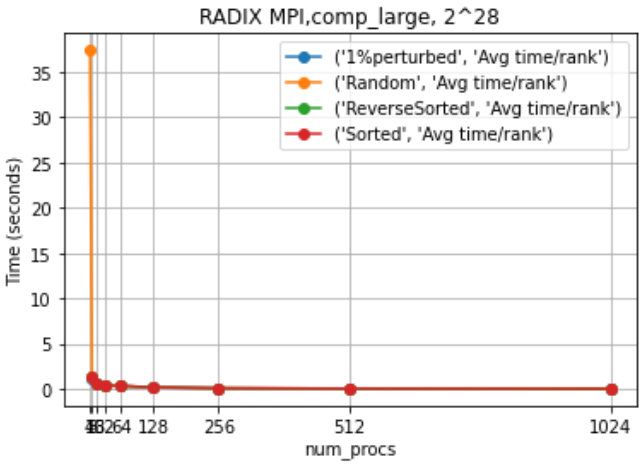

RADIX MPI,comm, 2^24
('1%perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')
Time (seconds)
num_procs

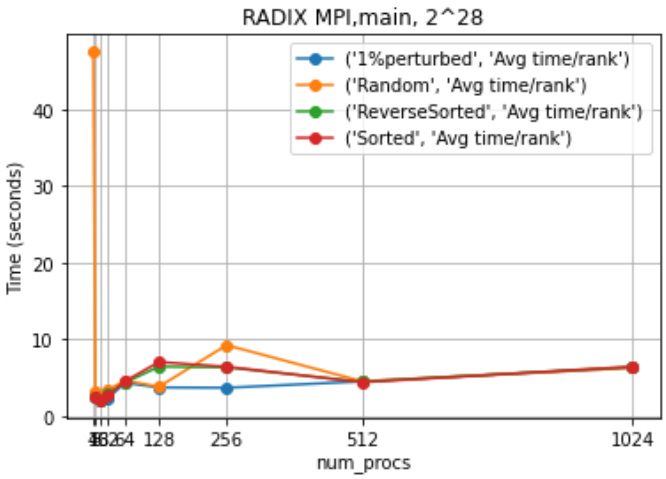

RADIX MPI,comp_large, 2^26
('1%perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')
Time (seconds)
num_procs


RADIX MPI,main, 2^26
('1%perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')
Time (seconds)
num_procs


RADIX MPI,comm, 2^26
('1%perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')
Time (seconds)
num_procs

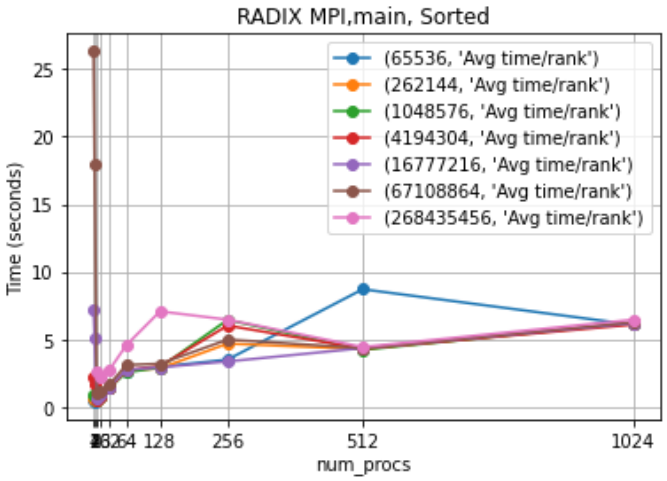


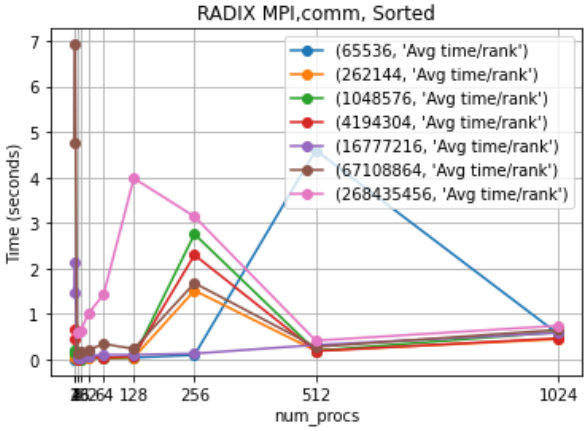


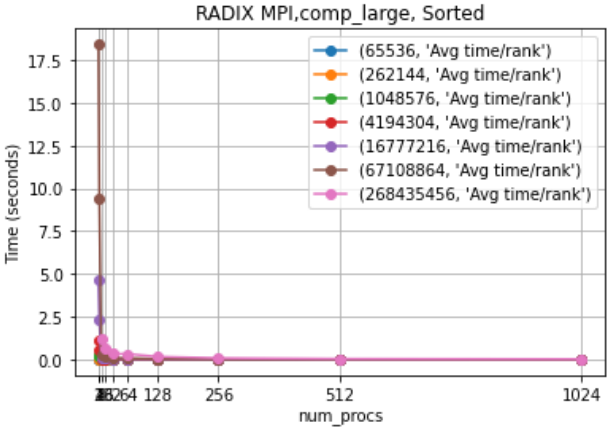


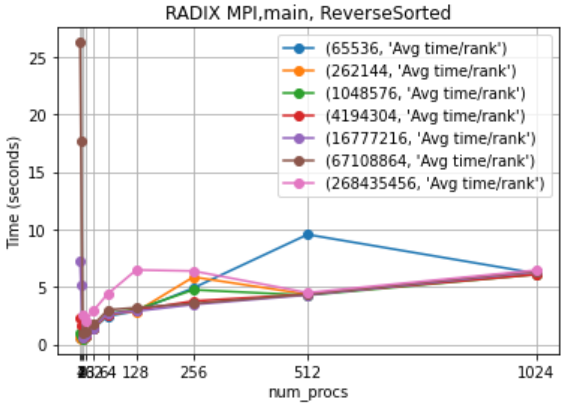
The strong scaling graphs for Radix MPI are pretty good for computation in that they decrease rapidly as the number of processors increases. This shows that my Radix MPI implementation does not have very much computation time and the time that it does take scales pretty well. The communication time is where the main bottleneck lies. The communication times randomly spike and usually increase overall, due to the communication overhead between all of the processes. This then results in the main trends being heavily representative of the communication times, since they have a larger effect than computation. The communication across my processors does not scale well with increased processors.

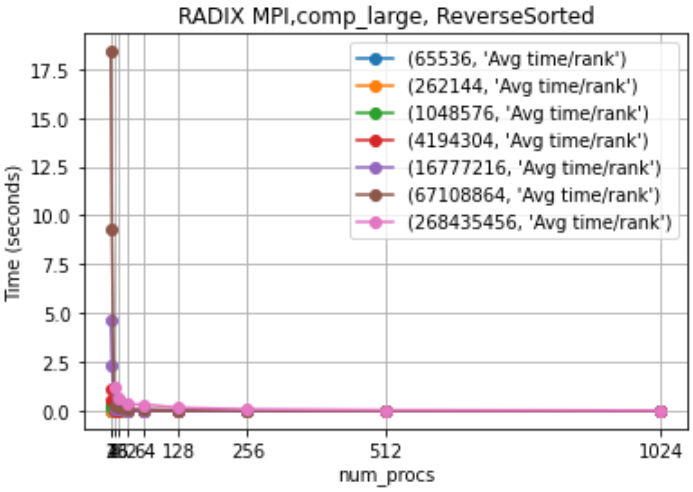
WEAK SCALING

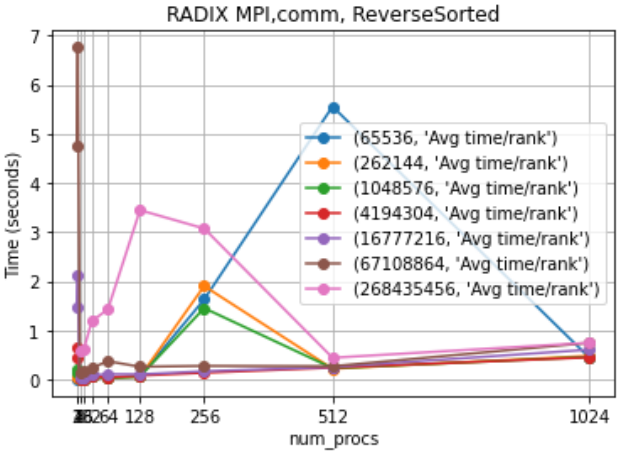


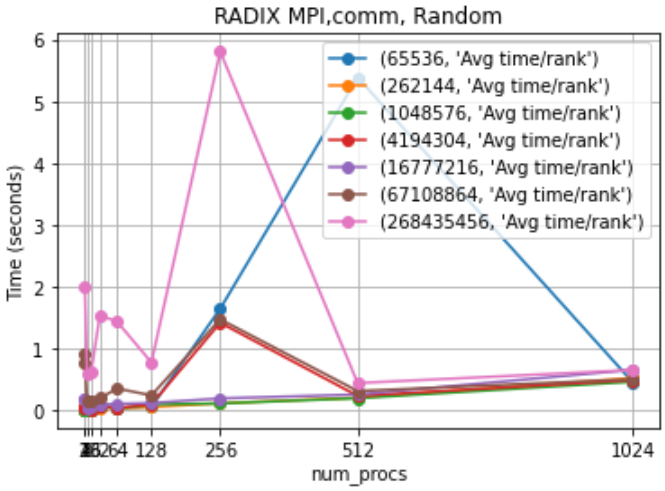


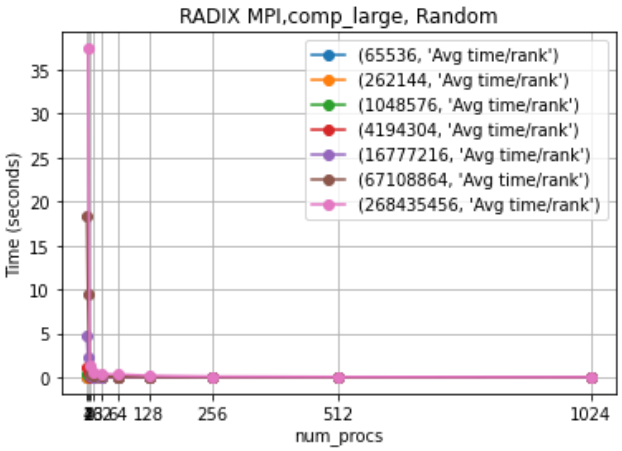


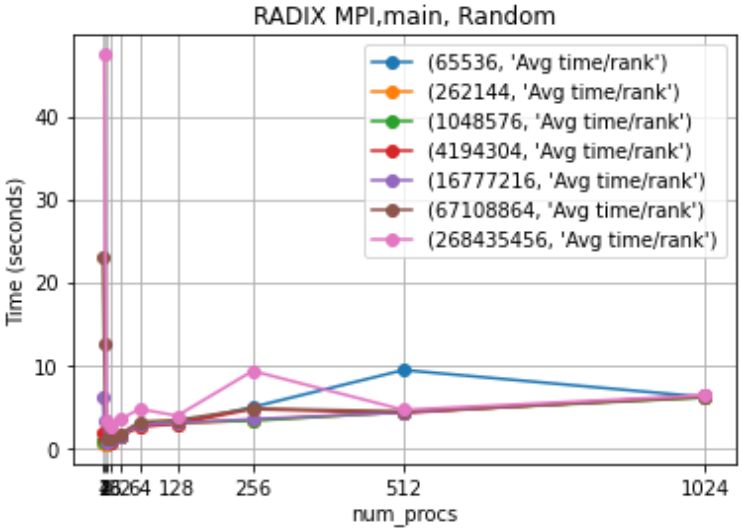


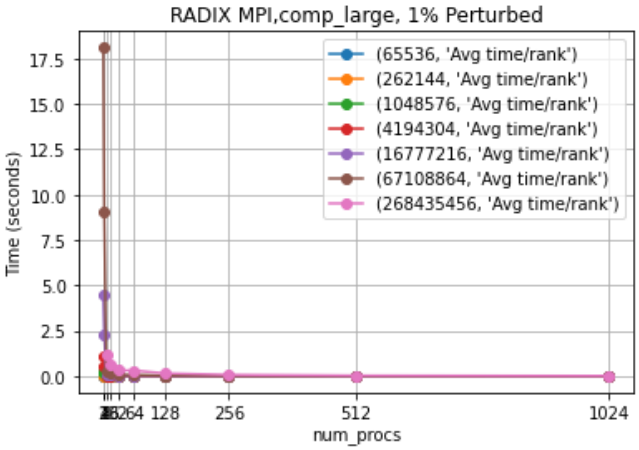


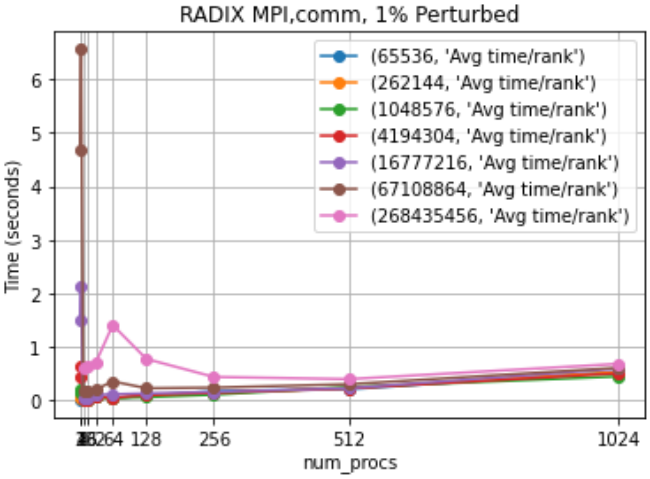


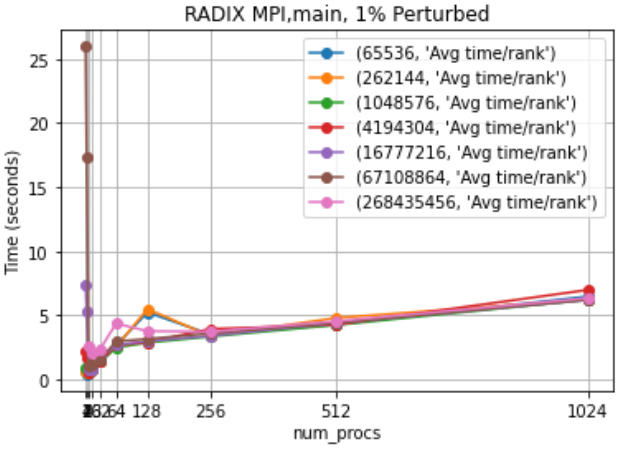

RADIX MPI,comm, Random
Time (seconds)
num_procs
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')


RADIX MPI,comp_large, Random
Time (seconds)
num_procs
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')


RADIX MPI,main, Random
Time (seconds)
num_procs
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')

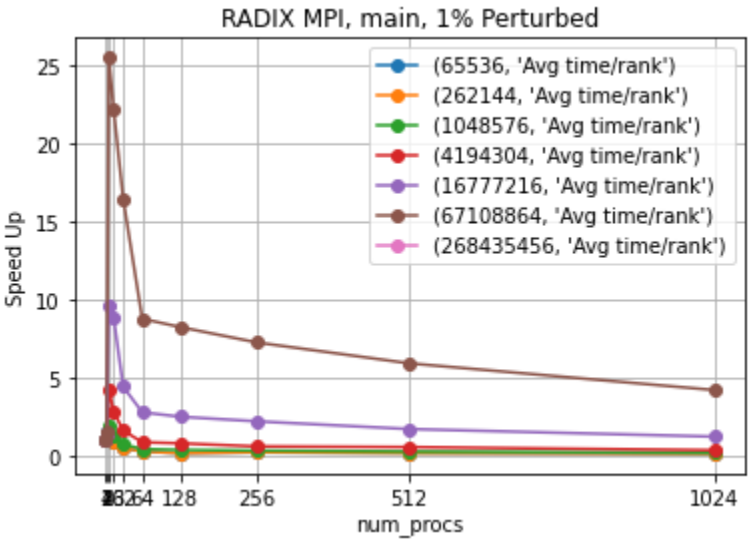

RADIX MPI,comp_large, 1% Perturbed
Time (seconds)
num_procs
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')

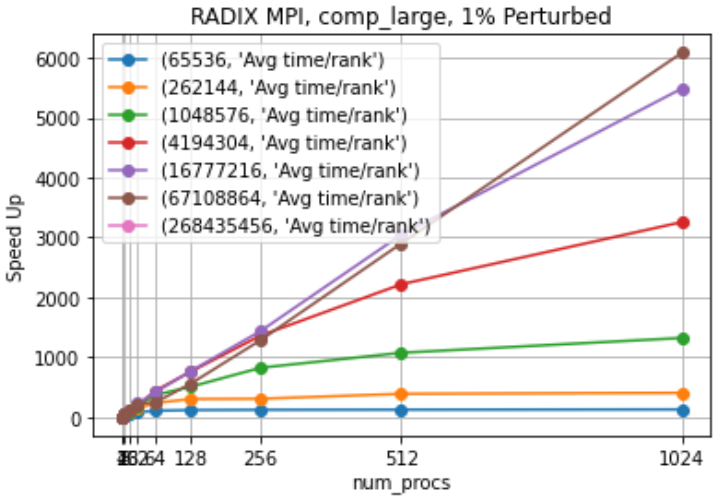

RADIX MPI,comm, 1% Perturbed
Time (seconds)
num_procs
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')

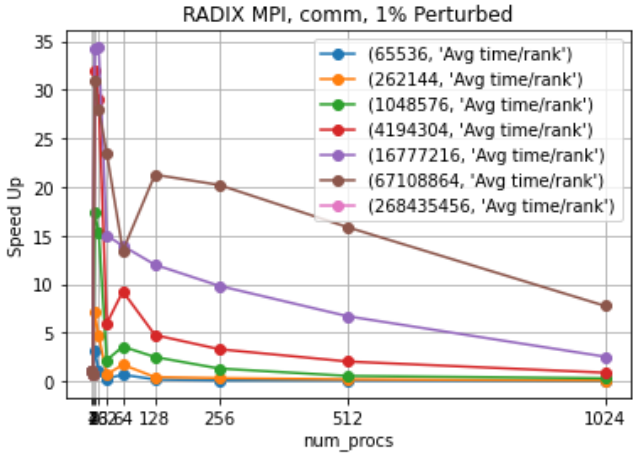

RADIX MPI,main, 1% Perturbed
Time (seconds)
num_procs
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')

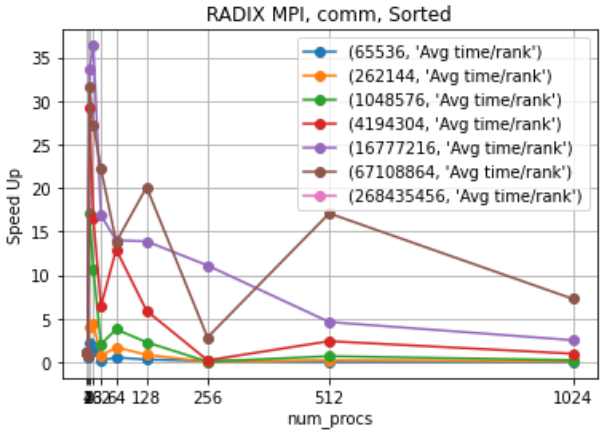
The weak scaling graphs for Radix MPI unfortunately have a similar trend to the strong scaling. The computation starts out very high but eventually evens out when you reach about 64 processors. The communication has spikes around 256 and 512 processors, but tends to even back out after that, so the communication overhead between processors is the worst at or around 256- 512 processors. Because of this, the trends for main again tend to follow the communication ones since that is what the main time tends to rely the most on.

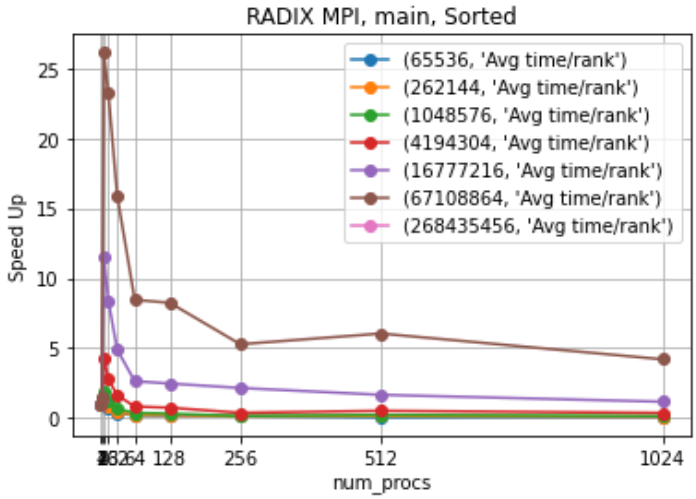
SPEEDUP

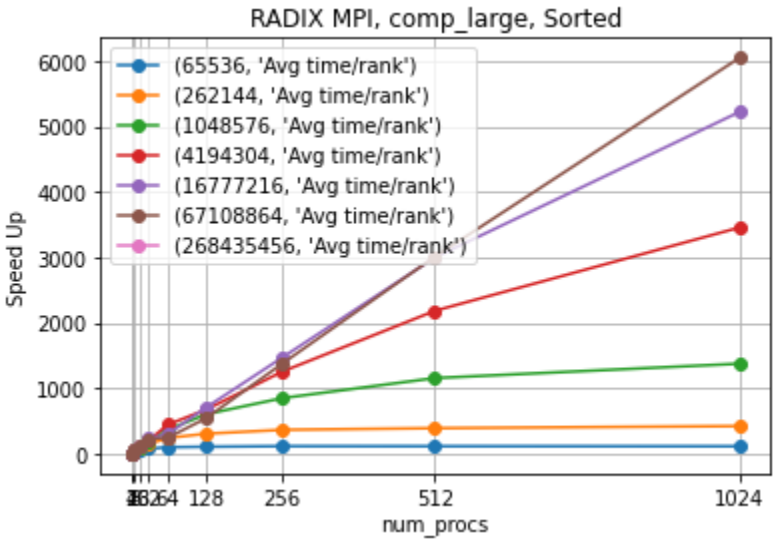

RADIX MPI, main, 1% Perturbed
Speed Up
num_procs
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')

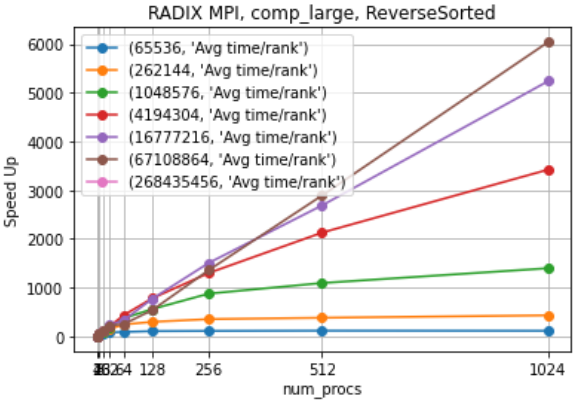

RADIX MPI, comp_large, 1% Perturbed
Speed Up
num_procs
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')

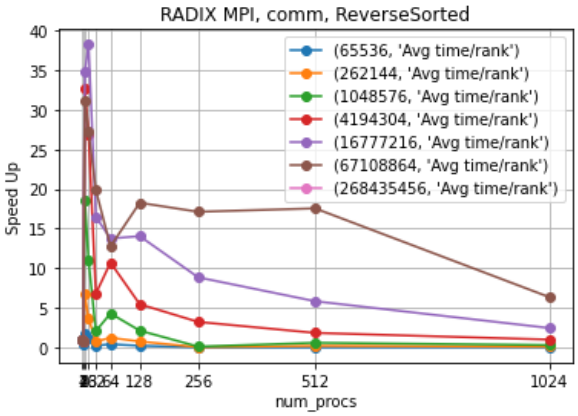

RADIX MPI, comm, 1% Perturbed
Speed Up
num_procs
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')

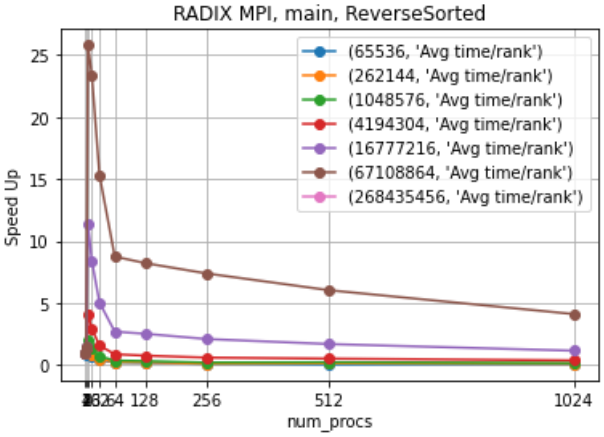

RADIX MPI, comm, Sorted
Speed Up
num_procs
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')

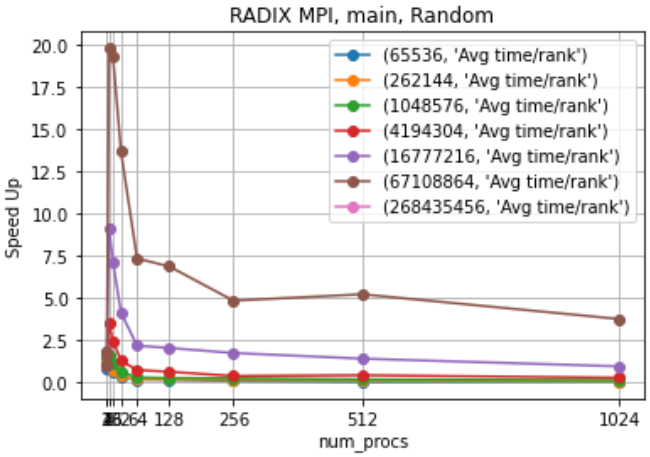

RADIX MPI, main, Sorted
Speed Up
num_procs
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')

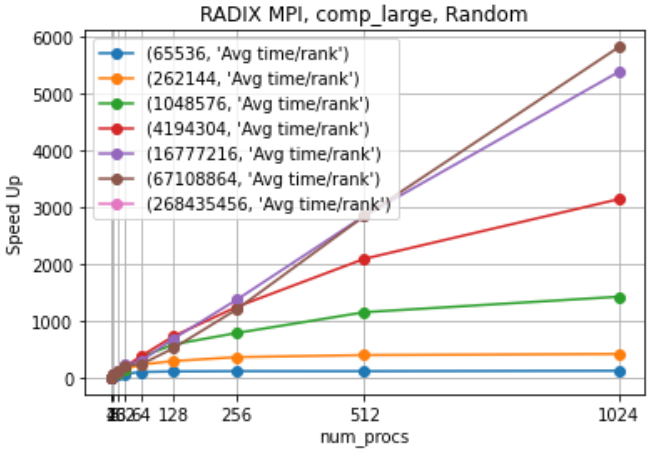

RADIX MPI, comp_large, Sorted
Speed Up
num_procs
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')

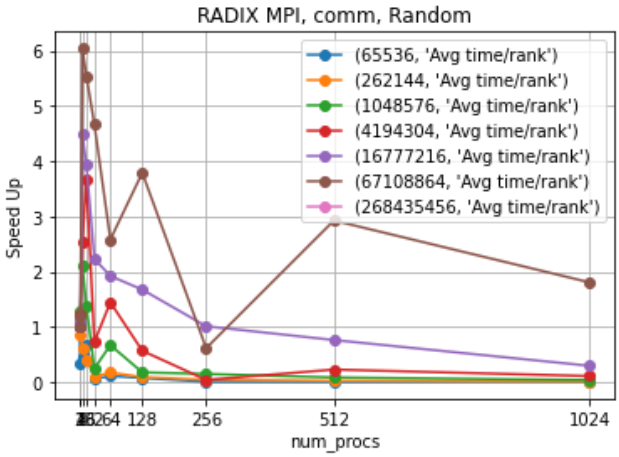

RADIX MPI, comp_large, ReverseSorted
Speed Up
num_procs
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')


RADIX MPI, comm, ReverseSorted
Speed Up
num_procs
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')


RADIX MPI, main, ReverseSorted
Speed Up
num_procs
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')


RADIX MPI, main, Random
Speed Up
num_procs
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')

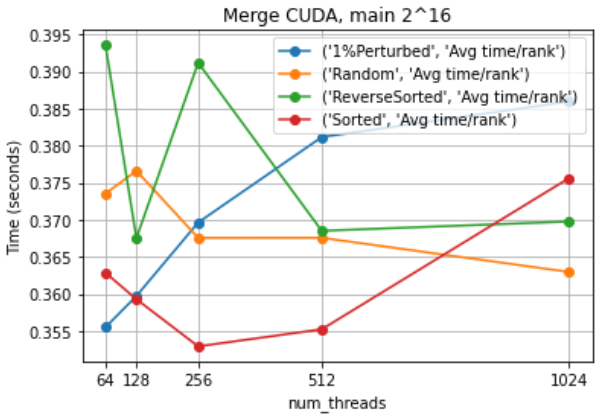

RADIX MPI, comp_large, Random
Speed Up
num_procs
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')

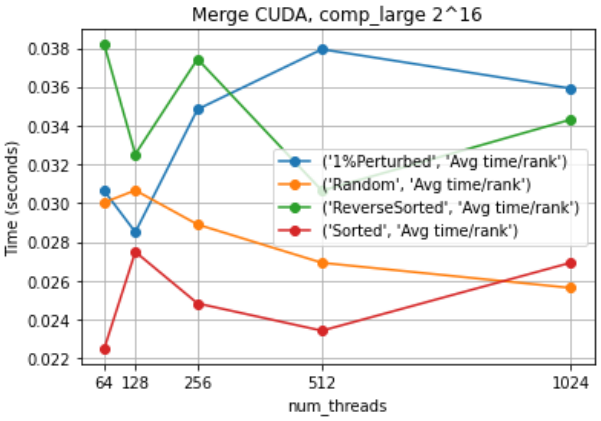

RADIX MPI, comm, Random
Speed Up
num_procs
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')

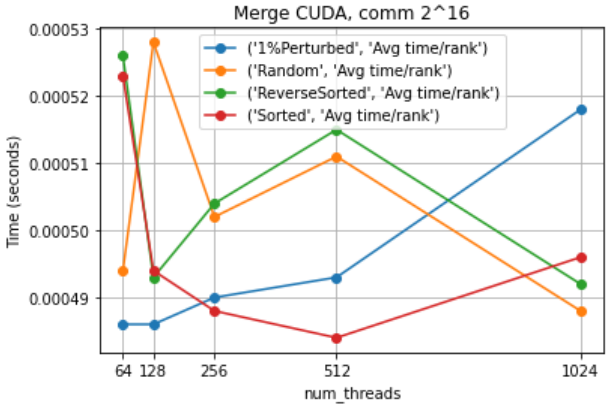
The speed up graphs for Radix MPI reveal similar trends to the strong and weak scaling. The computation graphs again reveal what you would expect in that the speedup increases as the number of processors increases since the more processors added, the less time the computation took. The communication and the resulting main graphs on the other hand show mostly the opposite of what we want in that as the number of processors increases, the speedup decreases due to significant communication overhead time. This means that my implementation of radix MPI did not, in fact, parallelize very well.

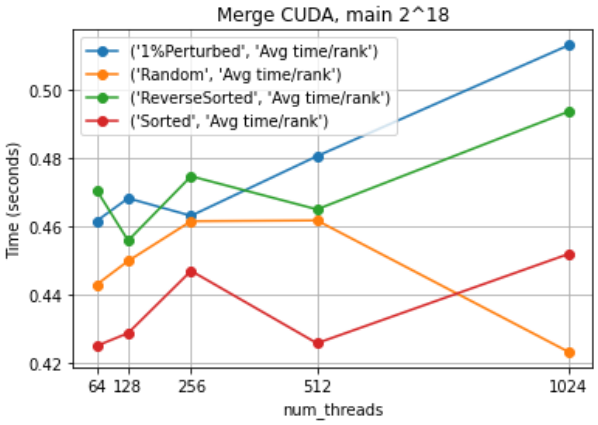
MERGE CUDA

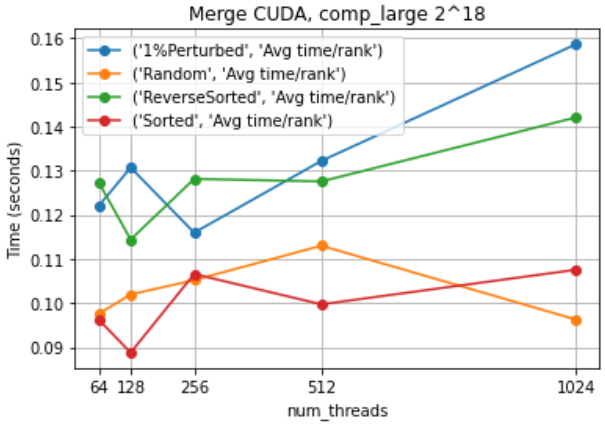
STRONG SCALING

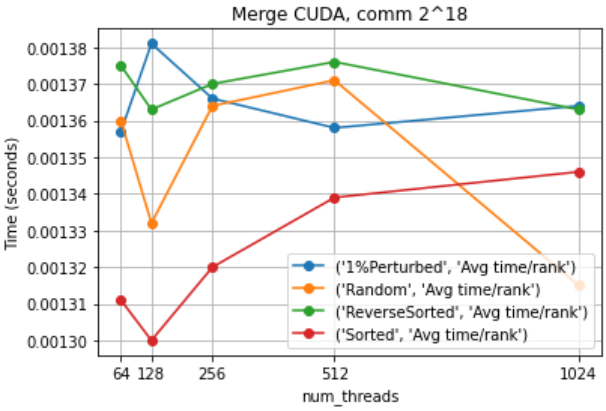


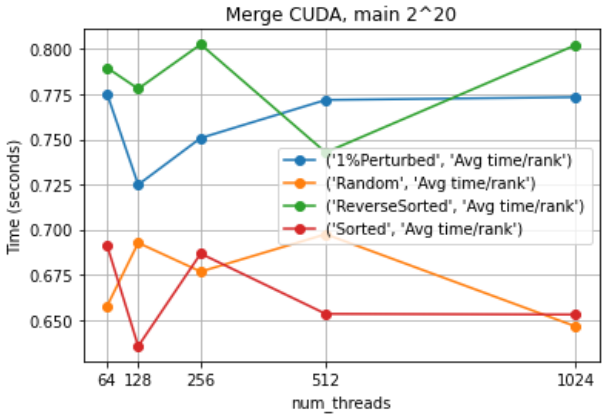


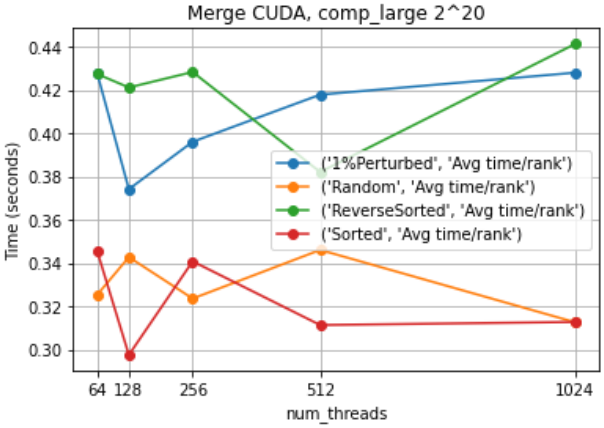


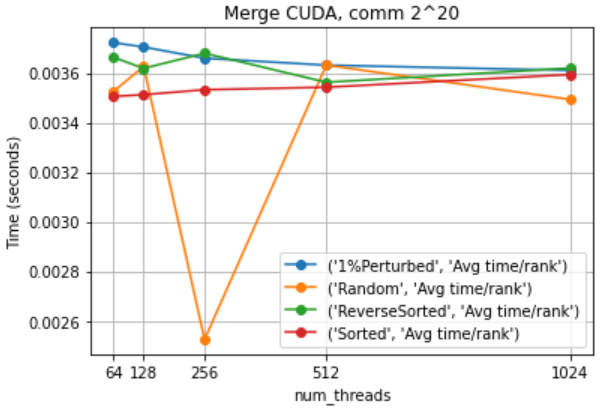


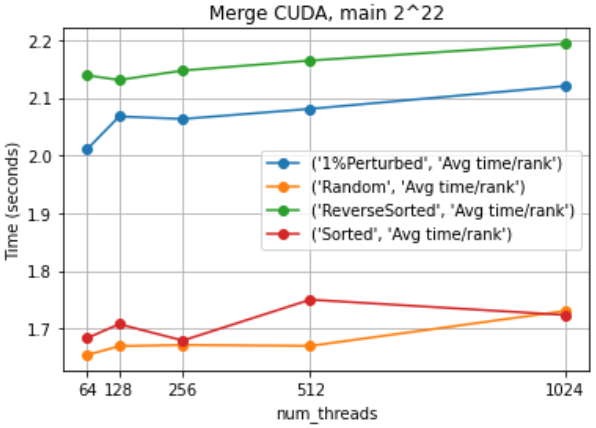


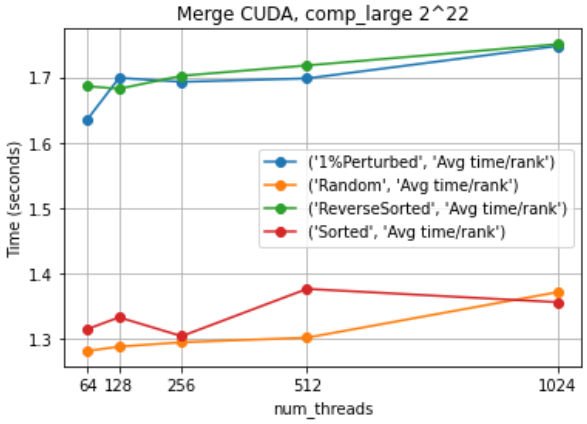


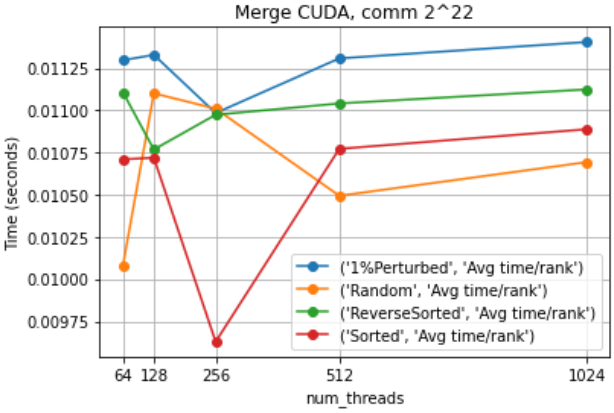

Merge CUDA, main 2^20
Time (seconds)
num_threads
('1%Perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')

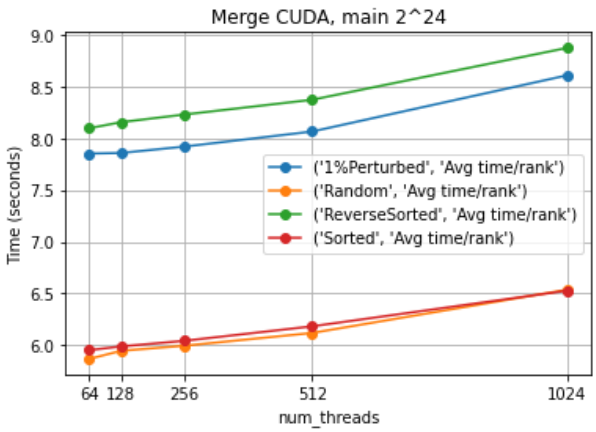

Merge CUDA, comp_large 2^20
Time (seconds)
num_threads
('1%Perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')

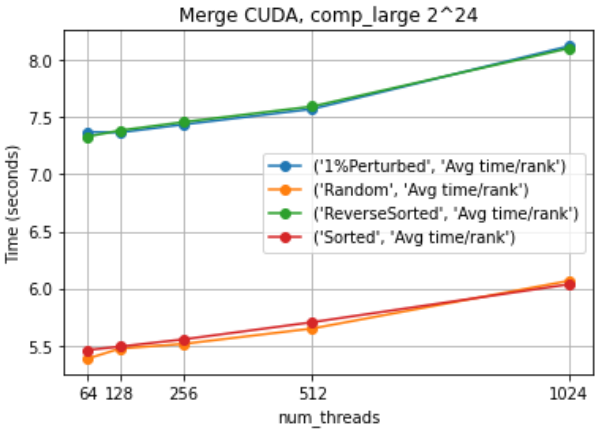

Merge CUDA, comm 2^20
Time (seconds)
num_threads
('1%Perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')

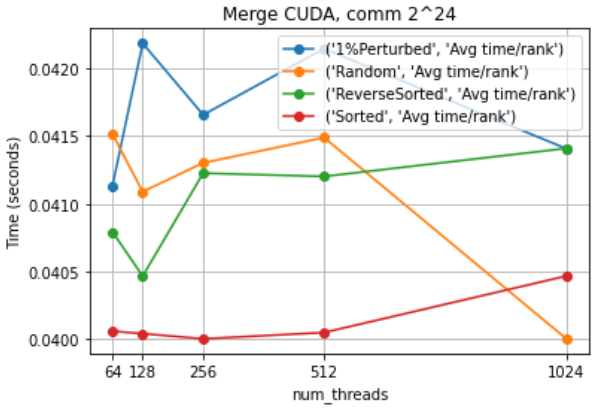

Merge CUDA, main 2^22
Time (seconds)
num_threads
('1%Perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')

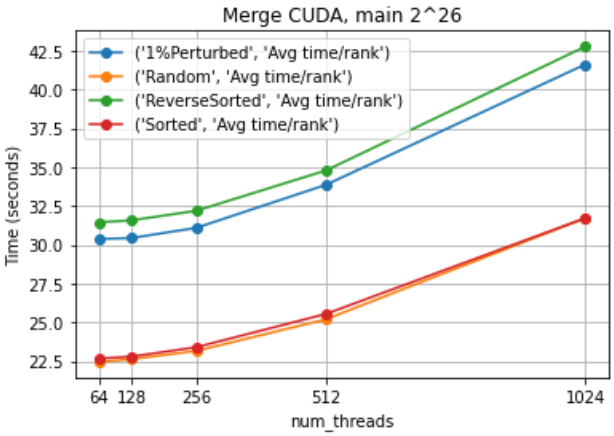

Merge CUDA, comp_large 2^22
Time (seconds)
num_threads
('1%Perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')

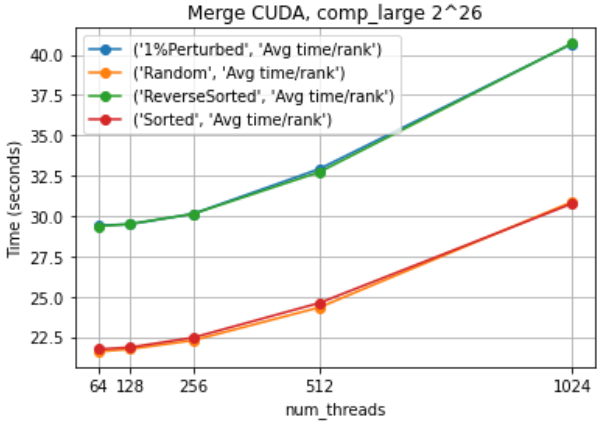

Merge CUDA, comm 2^22
Time (seconds)
num_threads
('1%Perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')

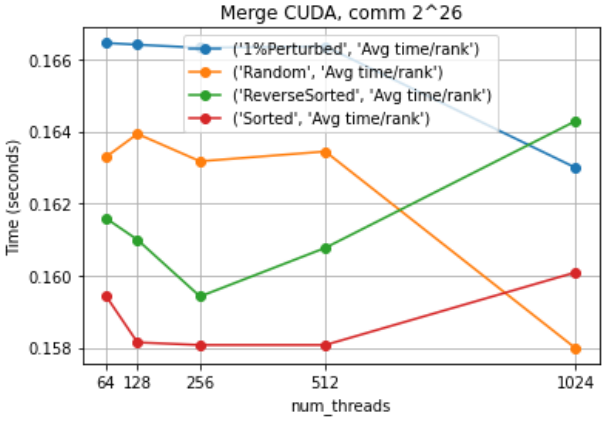

Merge CUDA, main 2^24
Time (seconds)
num_threads
('1%Perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')

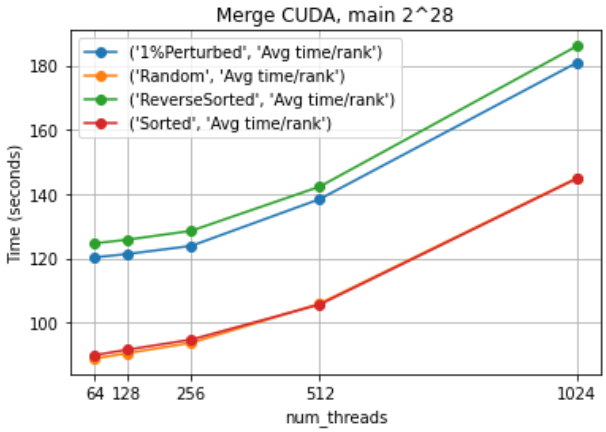

Merge CUDA, comp_large 2^24
Time (seconds)
num_threads
('1%Perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')

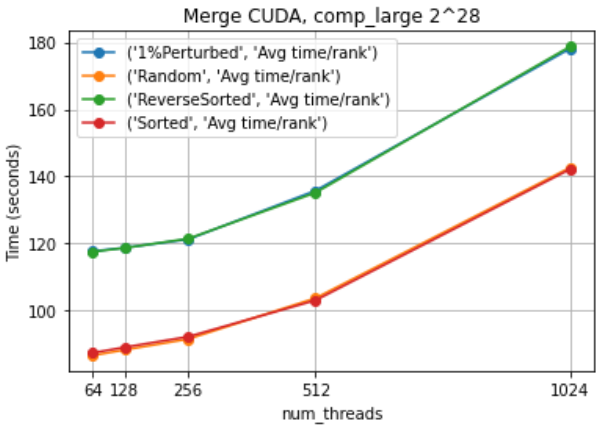

Merge CUDA, comm 2^24
Time (seconds)
num_threads
('1%Perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')

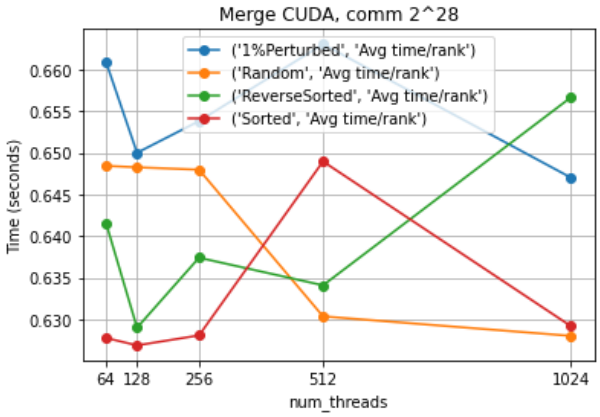

Merge CUDA, main 2^26
Time (seconds)
num_threads
('1%Perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')


Merge CUDA, comp_large 2^26
Time (seconds)
num_threads
('1%Perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')


Merge CUDA, comm 2^26
Time (seconds)
num_threads
('1%Perturbed', 'Avg time/rank')
('Random', 'Avg time/rank')
('ReverseSorted', 'Avg time/rank')
('Sorted', 'Avg time/rank')

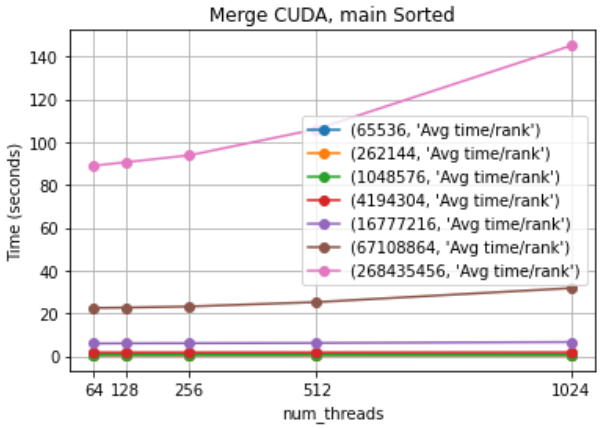


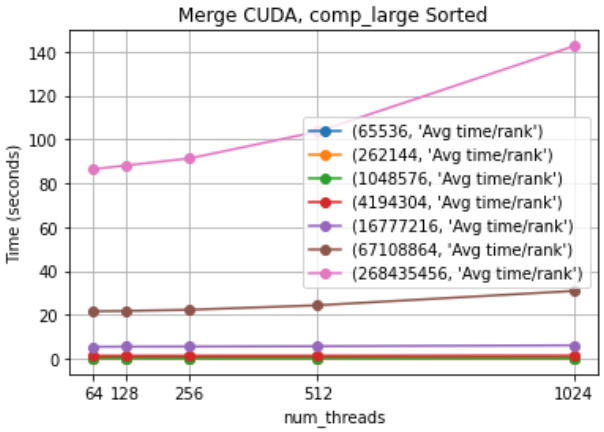


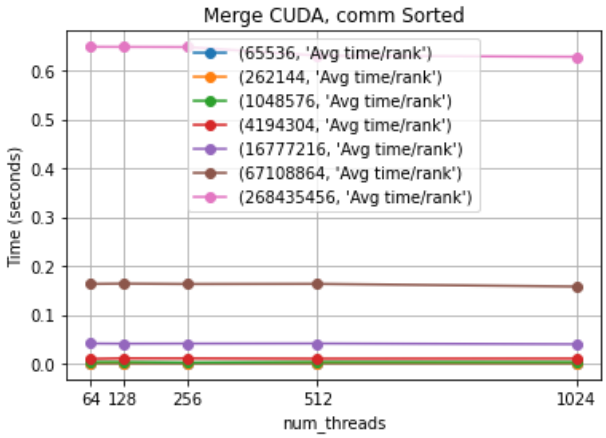


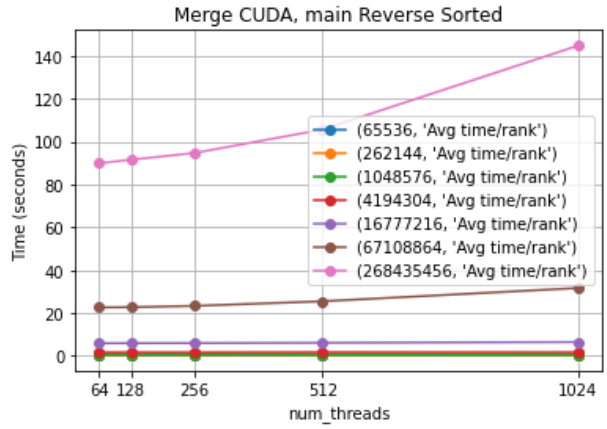
In the merge sort CUDA implementation for strong scaling, the comp_large graphs have a noticeable trend. For the smaller input sizes, the runtime as the number of threads increases is quite random with no discernible pattern. However, the input sizes 2^22 to 2^28 show a pattern of increased runtime as the number of threads increases. This is most likely due to my computation algorithm that did not parallelize well. I think my implementation of the merge sort should have been dynamic by allowing kernels to launch other kernels, which makes sense since merge sort is recursive. For communication, there seems to be a pattern where although there are some random spikes, the 1%perturbed and Sorted increases in runtime as the number of threads increase while the Random and Reverse Sorted decrease. For input sizes 2^24 and on, the Reverse Sorted and Sorted increase while 1%Perturbed and Random decrease. The different sort types can cause a workload imbalance among the threads, resulting in the differing runtimes caused by the communication overhead.

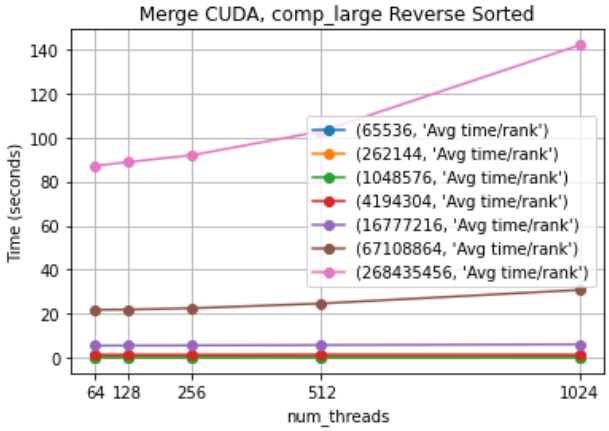
WEAK SCALING

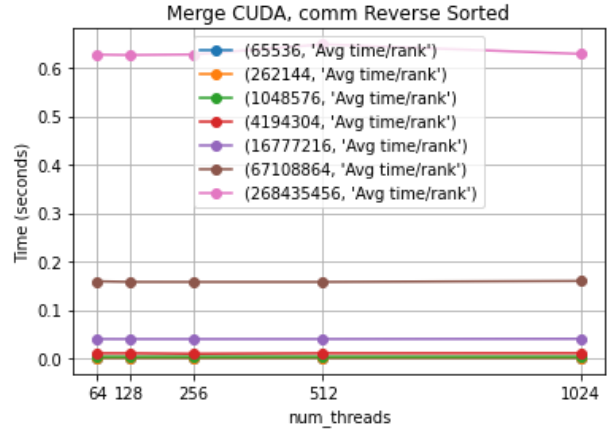


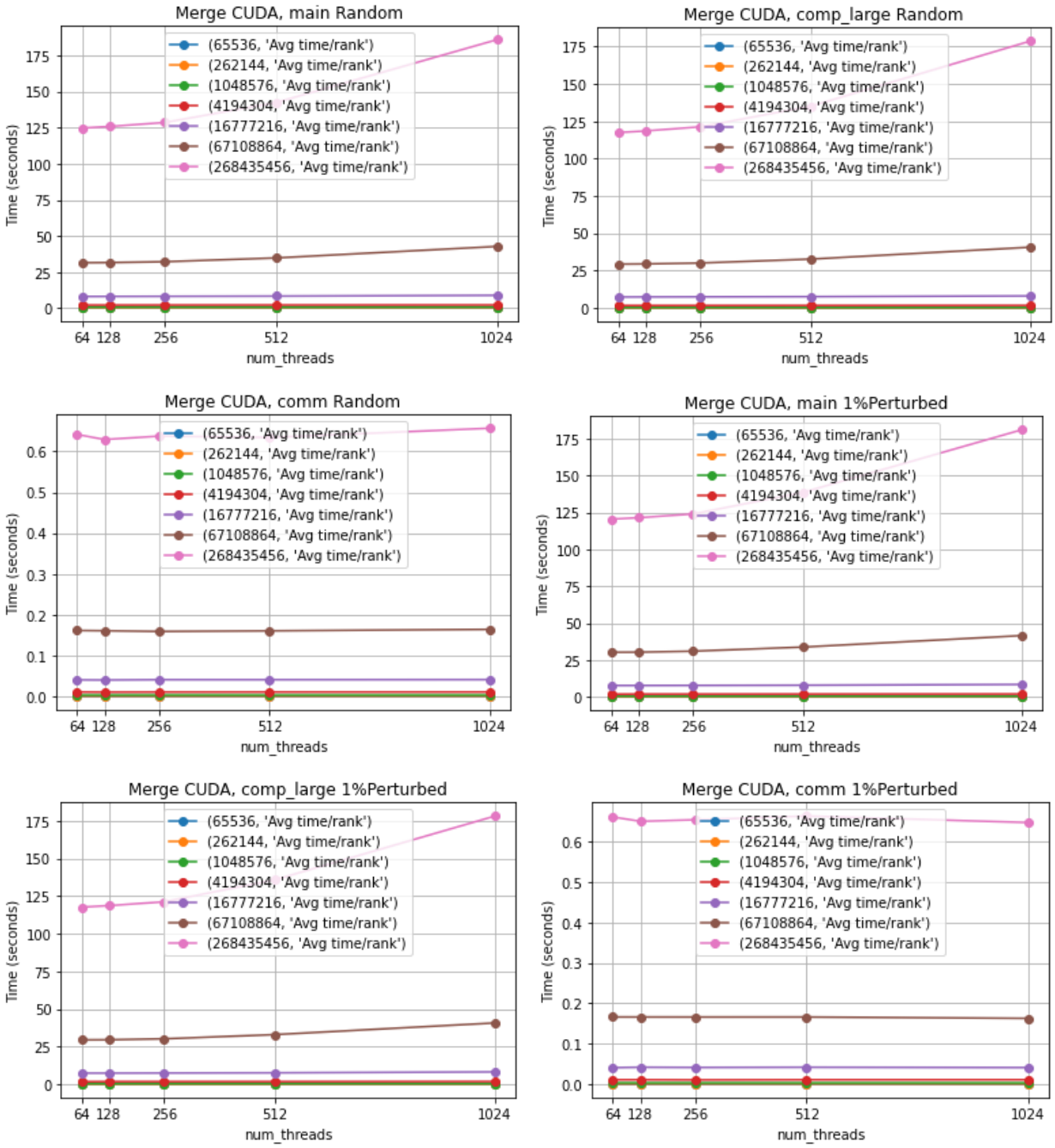






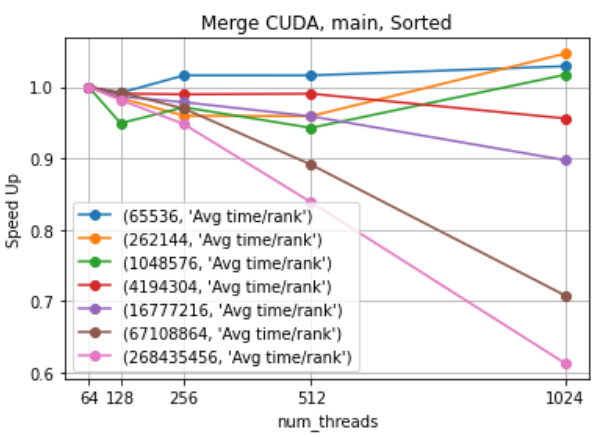


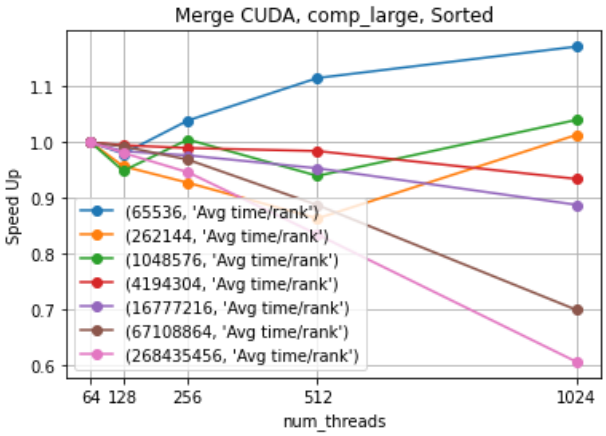


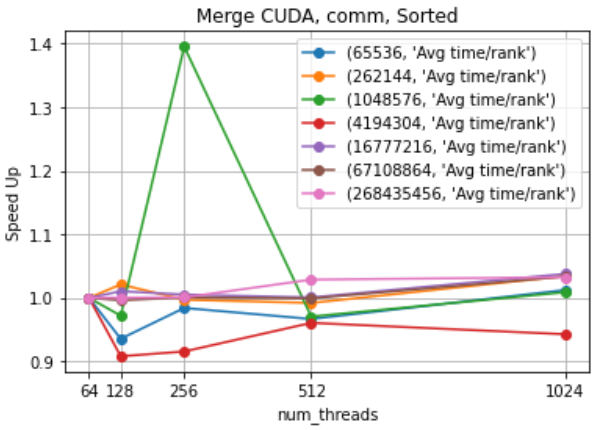


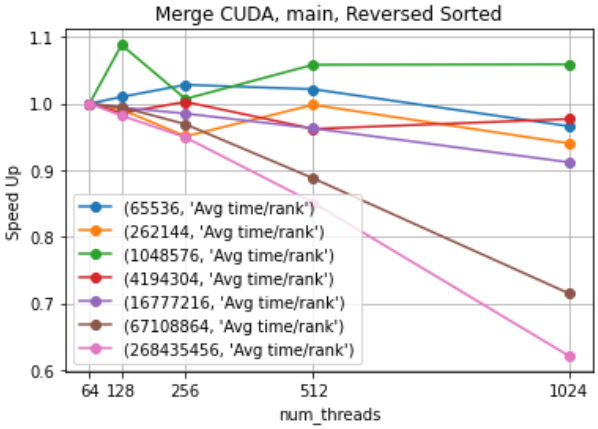
In the merge sort CUDA implementation for weak scaling, the computation large and communication graphs are similar in that they follow the ideal trend for weak scaling - constant run time as number of threads increase - except the computation large increases slightly. For computation large, the input sizes 2^16 to 2^20 are constant. The input sizes 2^22 to 2^28 slightly increase as the number of threads increase. This means there is good scalability within my computing algorithm. For communication, all of the graphs show that runtime is constant as the number of threads increases, which is the best case scenario for weak scaling. This is probably due to my implementation needing not as much communication/synchronization between threads.

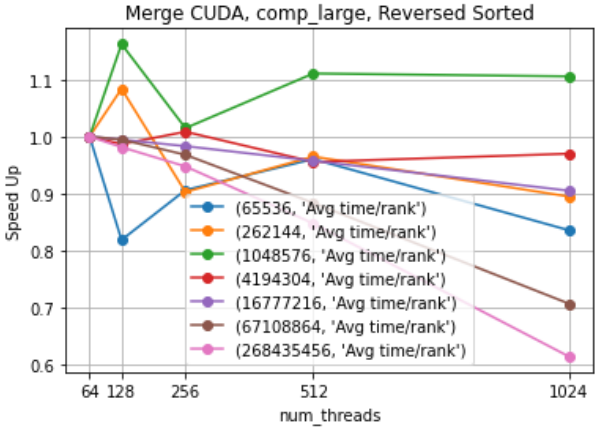
SPEEDUP

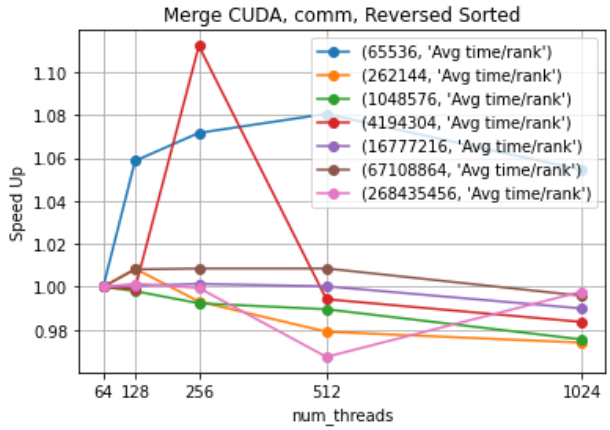

Merge CUDA, main, Sorted
Speed Up
num_threads
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')

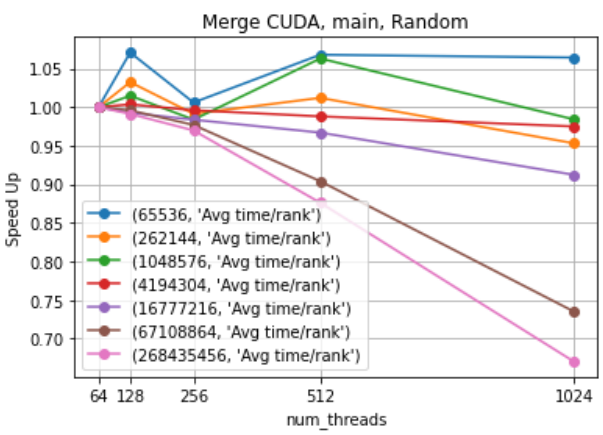

Merge CUDA, comp_large, Sorted
Speed Up
num_threads
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')

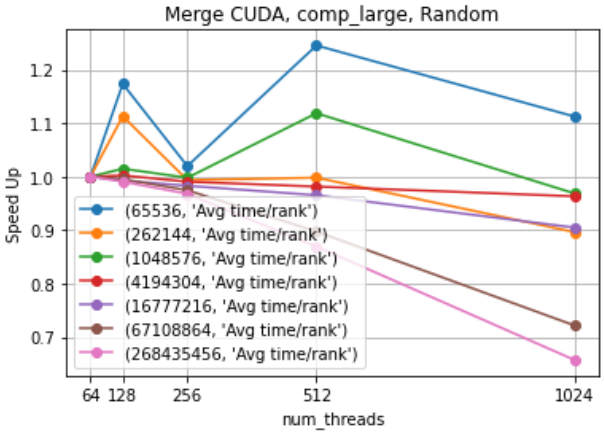

Merge CUDA, comm, Sorted
Speed Up
num_threads
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')

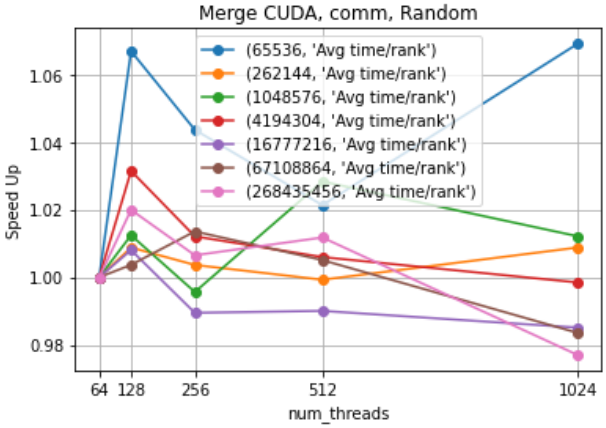

Merge CUDA, main, Reversed Sorted
Speed Up
num_threads
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')

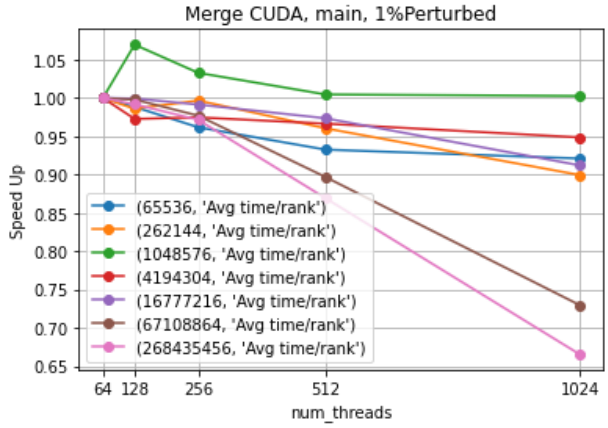

Merge CUDA, comp_large, Reversed Sorted
Speed Up
num_threads
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')

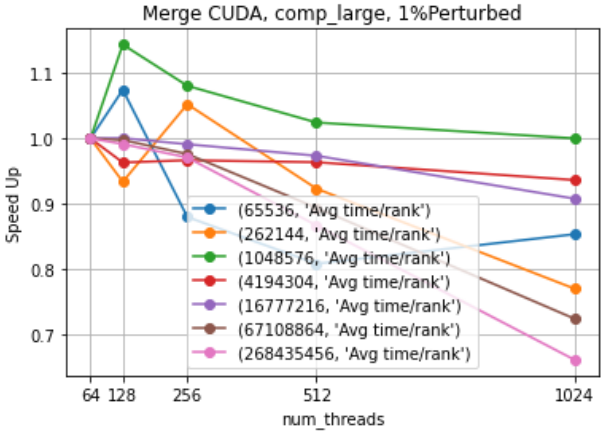

Merge CUDA, comm, Reversed Sorted
Speed Up
num_threads
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')

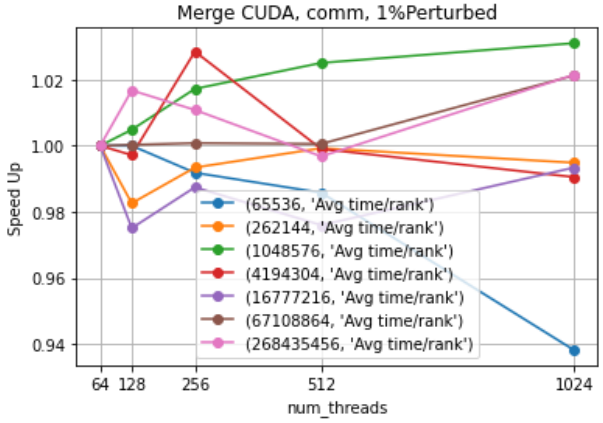

Merge CUDA, main, Random
Speed Up
num_threads
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')


Merge CUDA, comp_large, Random
Speed Up
num_threads
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')


Merge CUDA, comm, Random
Speed Up
num_threads
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')


Merge CUDA, main, 1%Perturbed
Speed Up
num_threads
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')

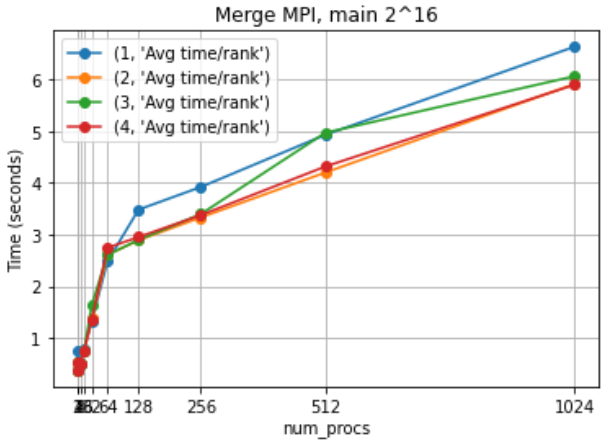

Merge CUDA, comp_large, 1%Perturbed
Speed Up
num_threads
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')

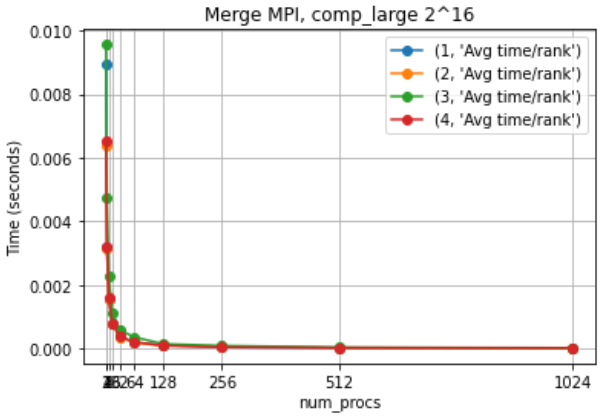

Merge CUDA, comm, 1%Perturbed
Speed Up
num_threads
(65536, 'Avg time/rank')
(262144, 'Avg time/rank')
(1048576, 'Avg time/rank')
(4194304, 'Avg time/rank')
(16777216, 'Avg time/rank')
(67108864, 'Avg time/rank')
(268435456, 'Avg time/rank')

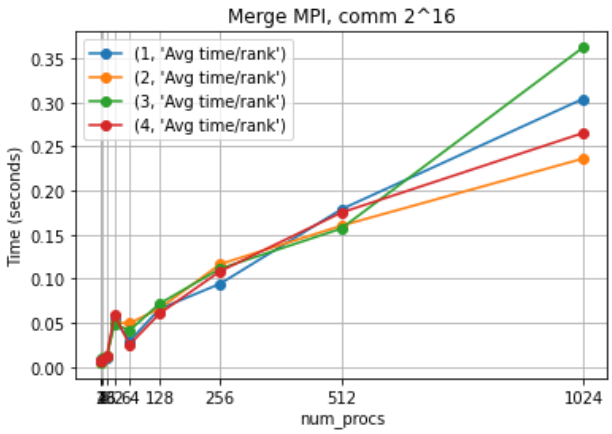
In the merge sort CUDA implementation for speedup, the computation large graphs show a general trend of decreasing in speedup as the number of threads increases. This could be due to diminishing returns from the thread management overhead. My implementation is not parallelized efficiently as I had trouble getting it to work in the first place for large input sizes. This is why my speedup is not good for computation. For the communication graphs, the Sorted array had no change in speedup, Reverse Sorted had a decreasing speedup, Random had a decreasing speedup except for input size 2^16 which increases, and 1%Perturbed had a mixture of increasing and decreasing speedups. Since the input is already sorted, there was probably less parallel processing occurring so speedup is constant. Since a reverse sorted array has a lot of dependencies, this could limit parallelism and decrease speedup. The random array had increased speedup for smaller input sizes and decreased speedup for larger ones, which is due to communication or synchronization overhead. I am unsure why the 1%Perturbed is having a speedup like that though.

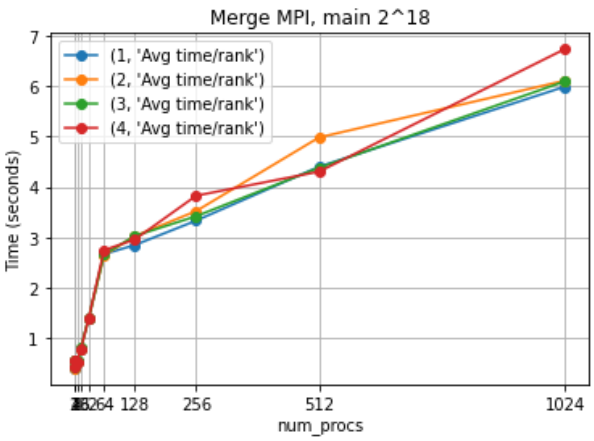
MERGE MPI

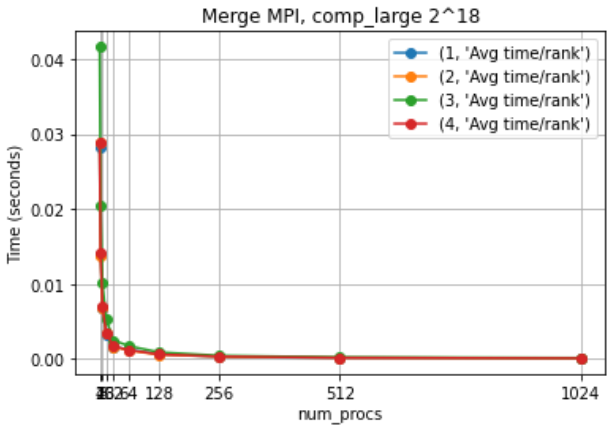
STRONG SCALING

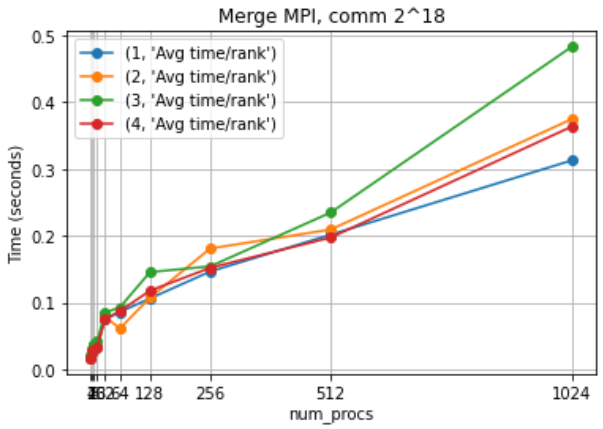


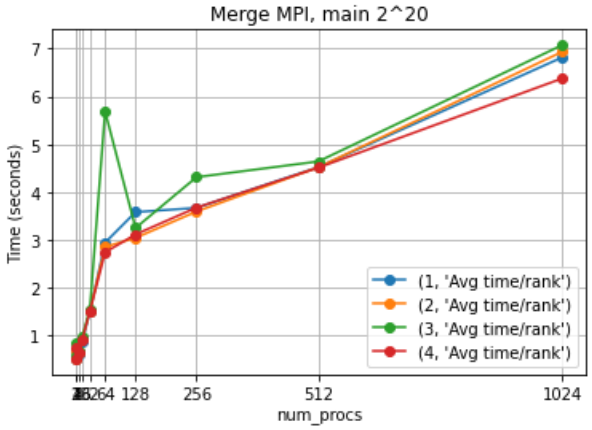


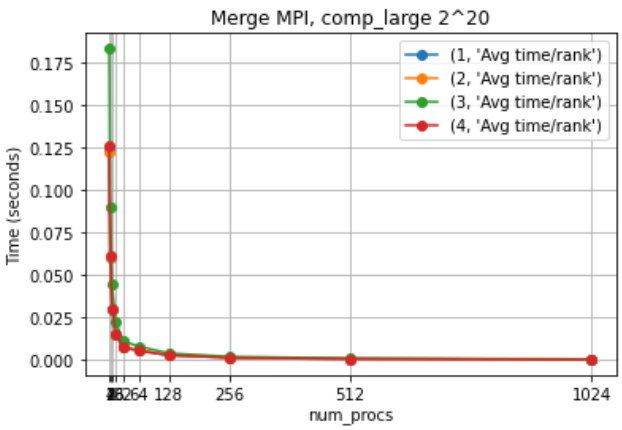


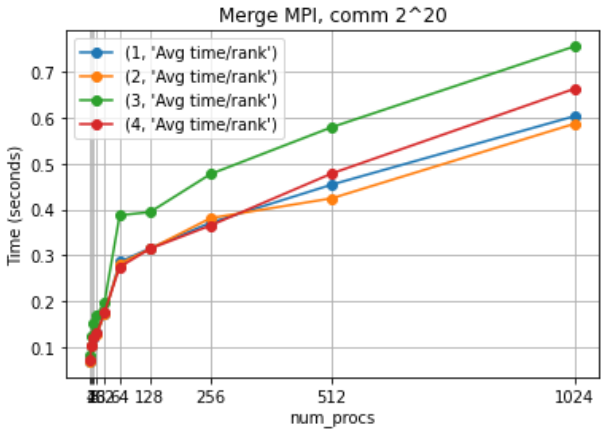


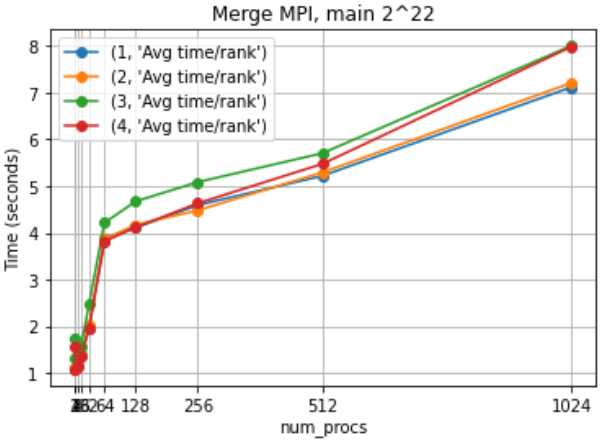

Merge MPI, comp_large 2^18
Time (seconds)
num_procs
(1, 'Avg time/rank')
(2, 'Avg time/rank')
(3, 'Avg time/rank')
(4, 'Avg time/rank')

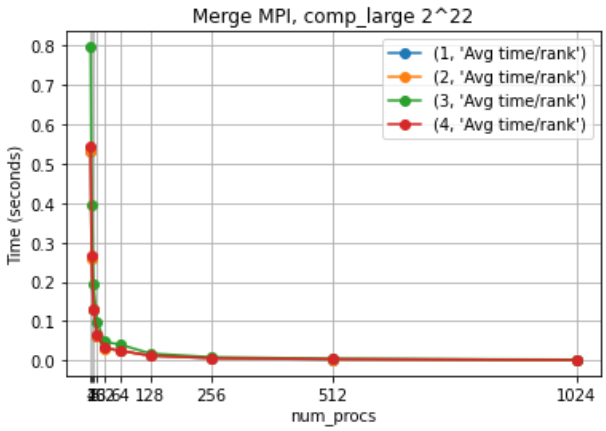

Merge MPI, comm 2^18
Time (seconds)
num_procs
(1, 'Avg time/rank')
(2, 'Avg time/rank')
(3, 'Avg time/rank')
(4, 'Avg time/rank')

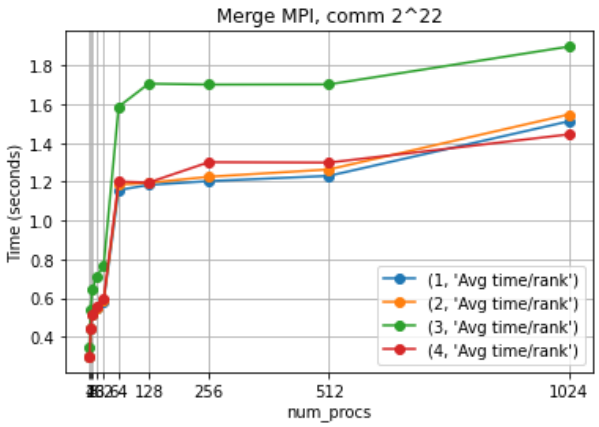

Merge MPI, main 2^20
Time (seconds)
num_procs
(1, 'Avg time/rank')
(2, 'Avg time/rank')
(3, 'Avg time/rank')
(4, 'Avg time/rank')

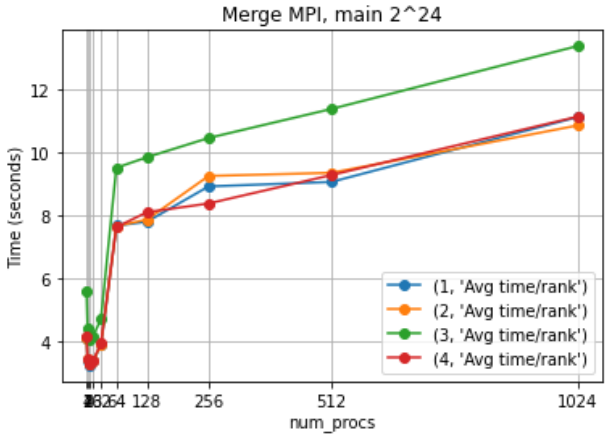

Merge MPI, comp_large 2^20
Time (seconds)
num_procs
(1, 'Avg time/rank')
(2, 'Avg time/rank')
(3, 'Avg time/rank')
(4, 'Avg time/rank')

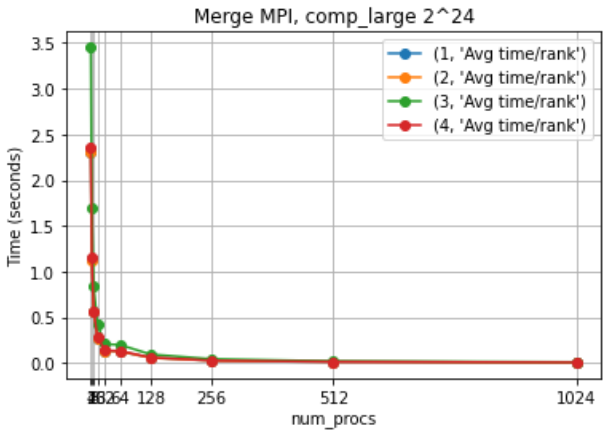

Merge MPI, comm 2^20
Time (seconds)
num_procs
(1, 'Avg time/rank')
(2, 'Avg time/rank')
(3, 'Avg time/rank')
(4, 'Avg time/rank')

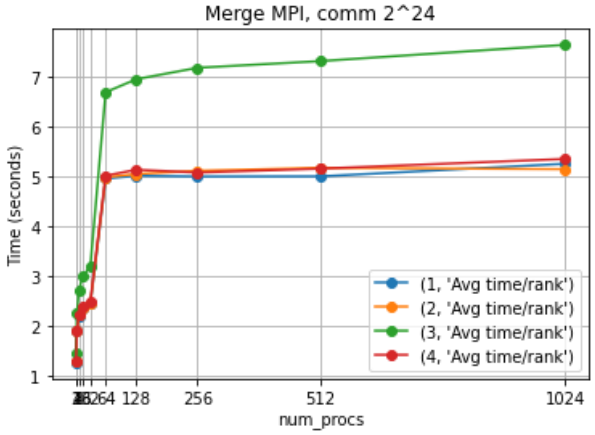

Merge MPI, main 2^22
Time (seconds)
num_procs
(1, 'Avg time/rank')
(2, 'Avg time/rank')
(3, 'Avg time/rank')
(4, 'Avg time/rank')

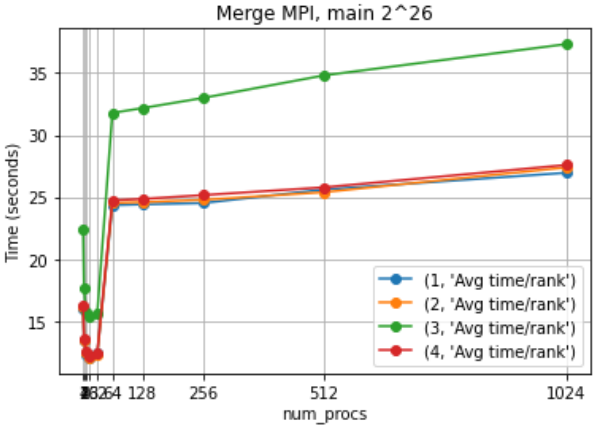

Merge MPI, comp_large 2^22
Time (seconds)
num_procs
(1, 'Avg time/rank')
(2, 'Avg time/rank')
(3, 'Avg time/rank')
(4, 'Avg time/rank')

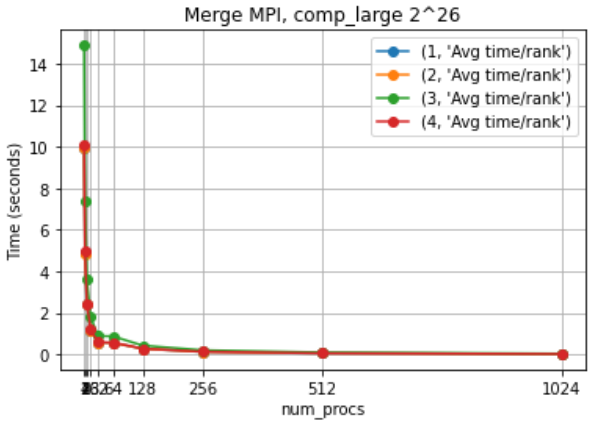

Merge MPI, comm 2^22
Time (seconds)
num_procs
(1, 'Avg time/rank')
(2, 'Avg time/rank')
(3, 'Avg time/rank')
(4, 'Avg time/rank')

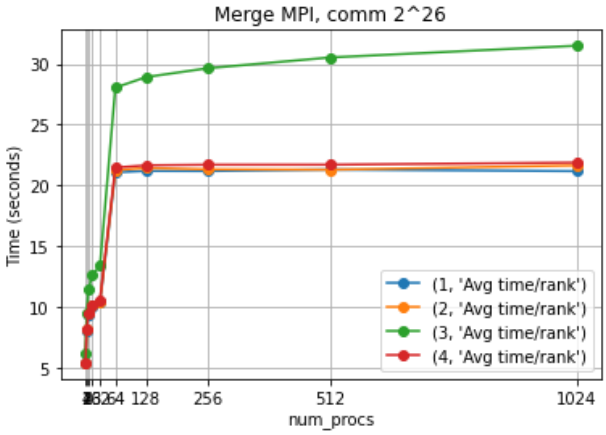

Merge MPI, main 2^24
Time (seconds)
num_procs
(1, 'Avg time/rank')
(2, 'Avg time/rank')
(3, 'Avg time/rank')
(4, 'Avg time/rank')

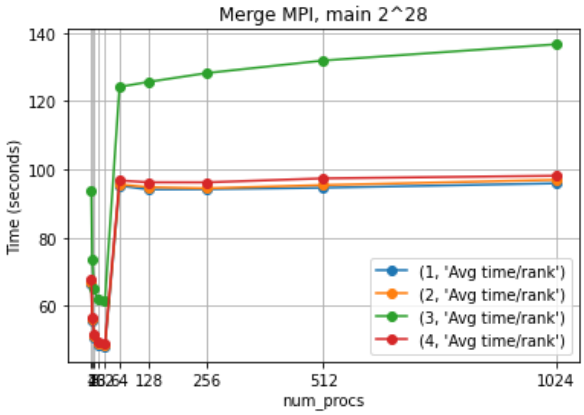

Merge MPI, comp_large 2^24
Time (seconds)
num_procs
(1, 'Avg time/rank')
(2, 'Avg time/rank')
(3, 'Avg time/rank')
(4, 'Avg time/rank')

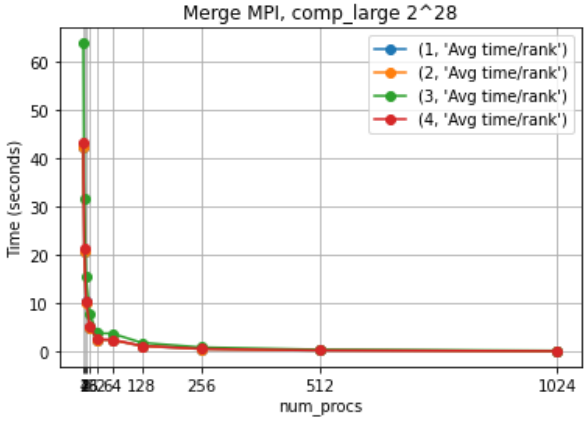

Merge MPI, comm 2^24
Time (seconds)
num_procs
(1, 'Avg time/rank')
(2, 'Avg time/rank')
(3, 'Avg time/rank')
(4, 'Avg time/rank')

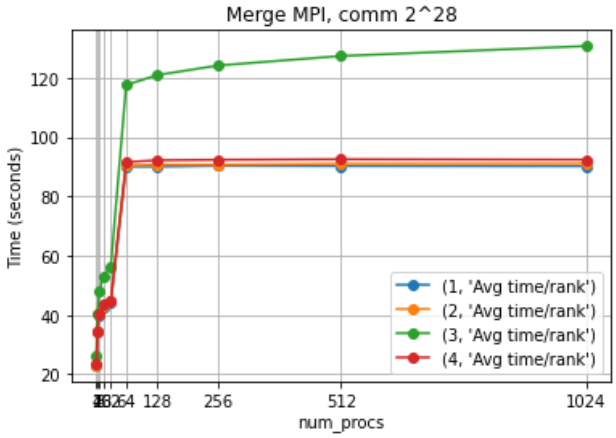

Merge MPI, main 2^26
Time (seconds)
num_procs
(1, 'Avg time/rank')
(2, 'Avg time/rank')
(3, 'Avg time/rank')
(4, 'Avg time/rank')





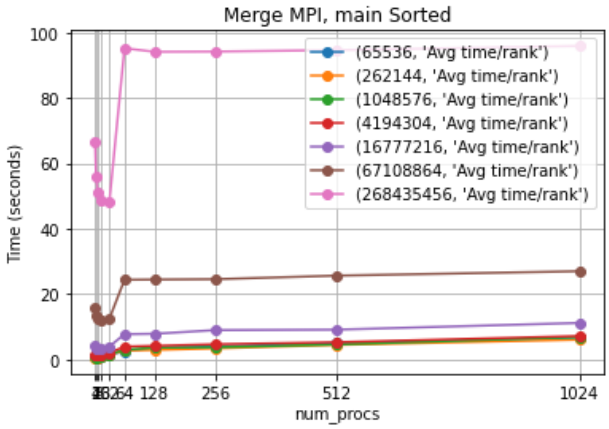


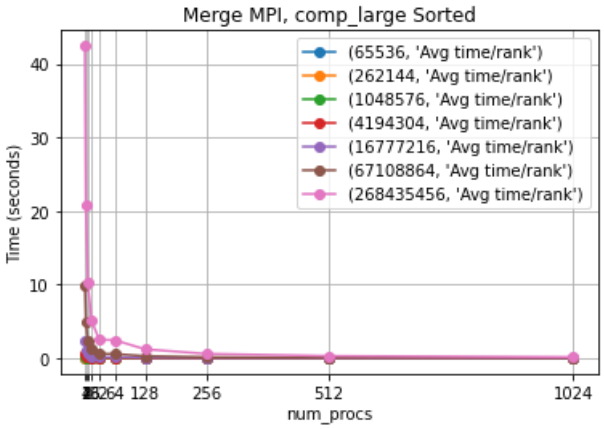


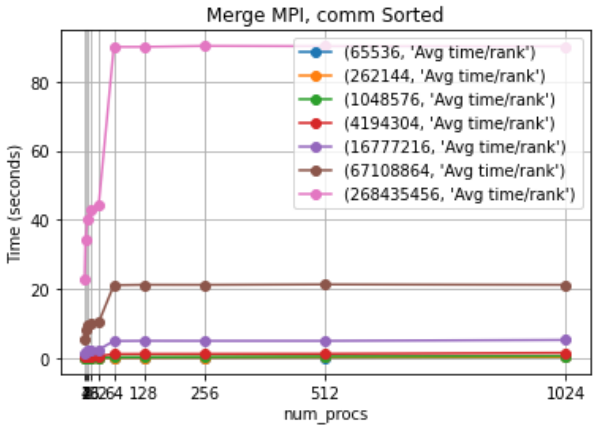


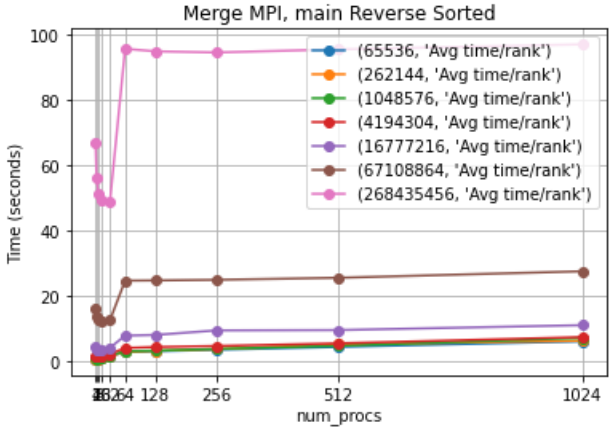
In the merge sort MPI implementation for strong scaling, the computation large decreases exponentially towards a runtime of 0 seconds. This means that my implementation scales really well since this is the ideal trend for strong scaling. My algorithm is efficient at parallelization. For communication, the trend is very different. For input sizes 2^16 to 2^20, the runtime increases almost linearly as the number of processors increases. For input sizes 2^22 to 2^28, the runtime spikes up at 64 processors and then begins to plateau. This is due to communication overhead from MPI_Scatter and MPI_Gather. This is more noticeable in the smaller input sizes since the amount of work that each process performs isn't enough to amortize the overhead.

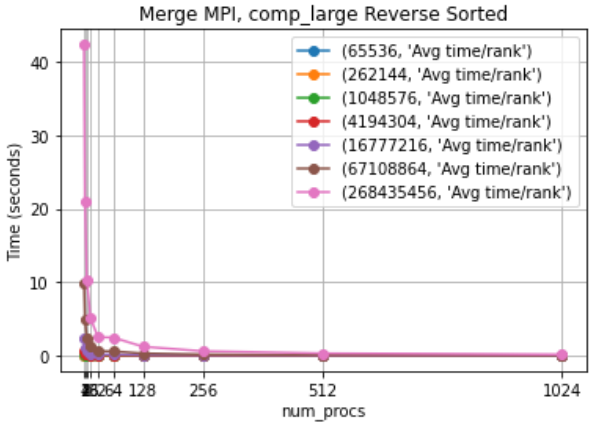
WEAK SCALING

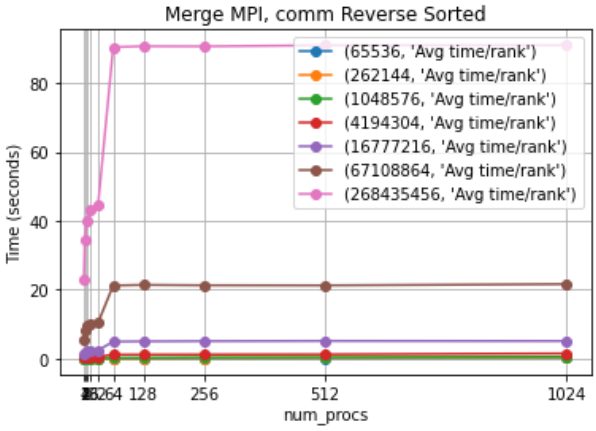


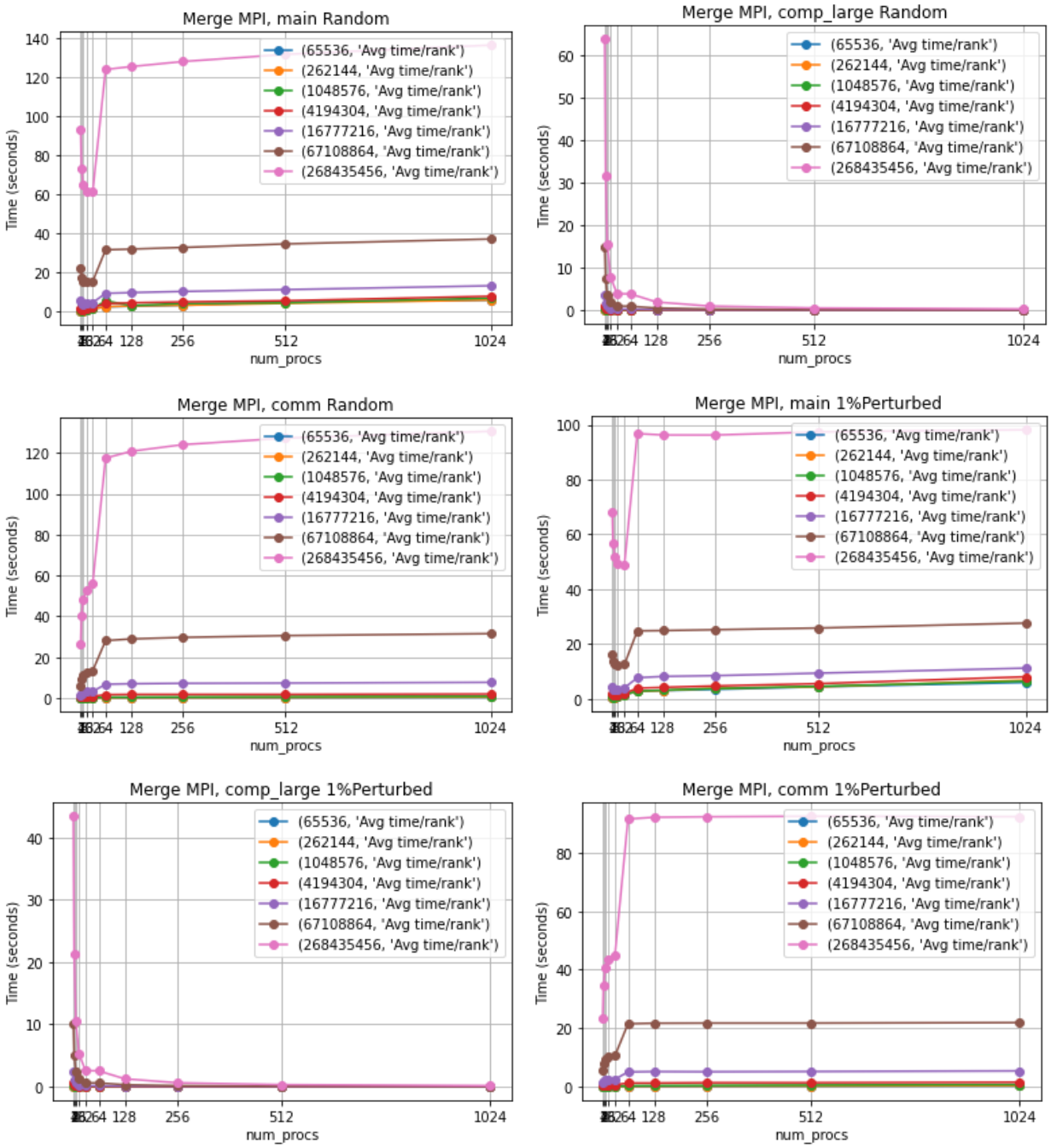








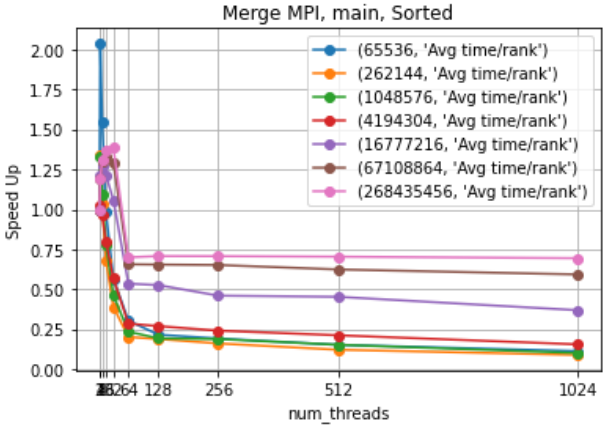


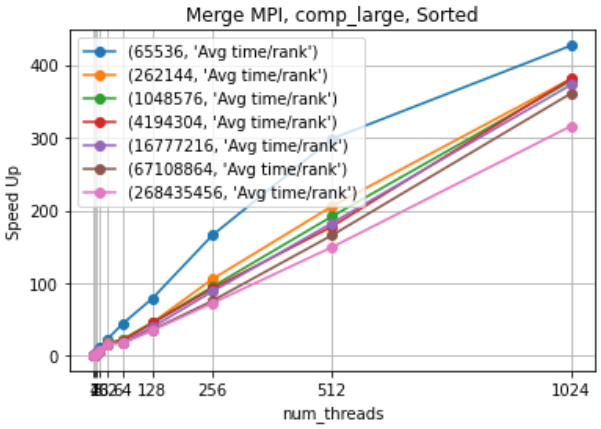


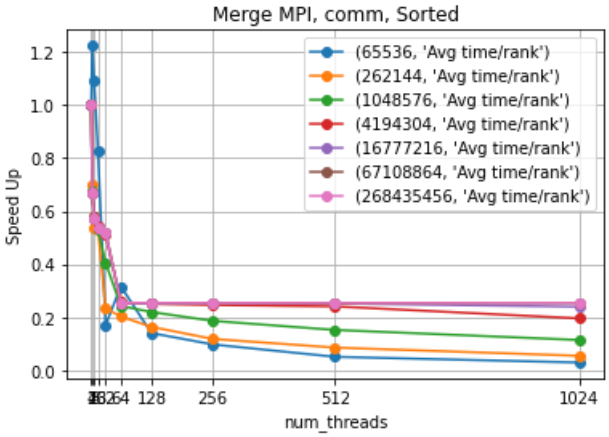
In the merge sort MPI implementation for weak scaling, the computation large shows a general trend of decreasing exponentially in runtime as the number of processors increases. This could be due to the algorithm being unable to scale well with the larger number of processors. This is inconsistent with my strong scaling being good though. For the communication, the runtime increases up to 64 processors and then stays constant. This could be due to the size of the messages being passed and causing communication overhead. However, when the number of processors increases, the overhead is less pronounced and results in it being constant.

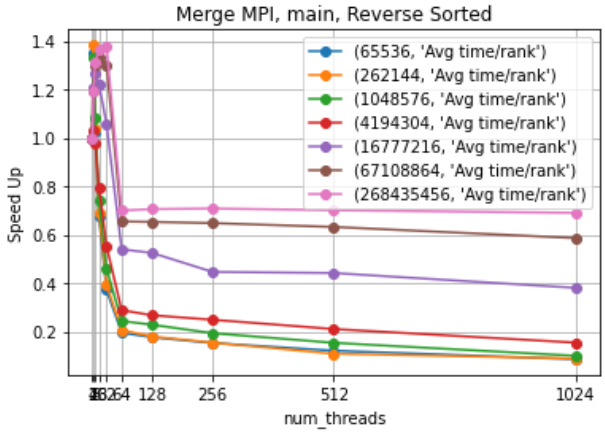
SPEEDUP

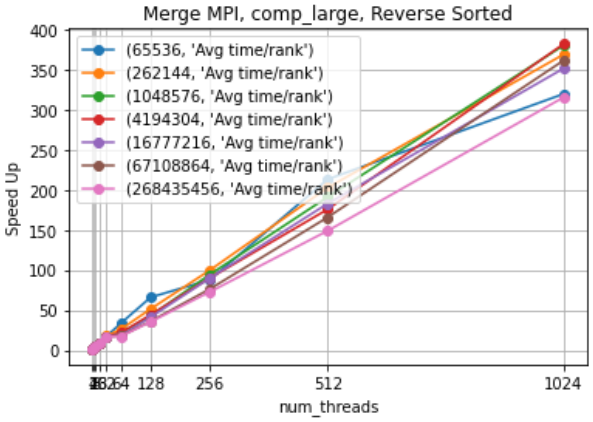
*the x axis should be number of processors

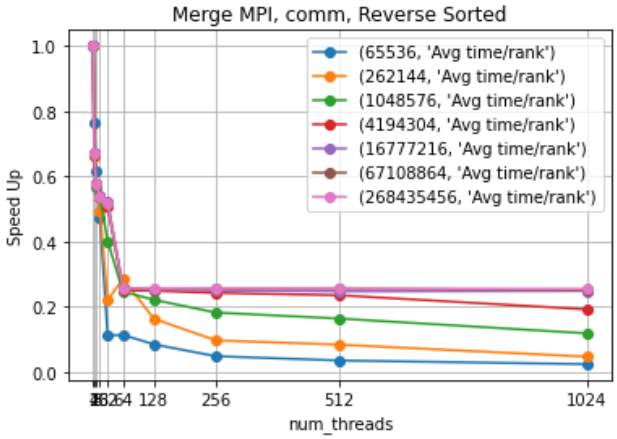


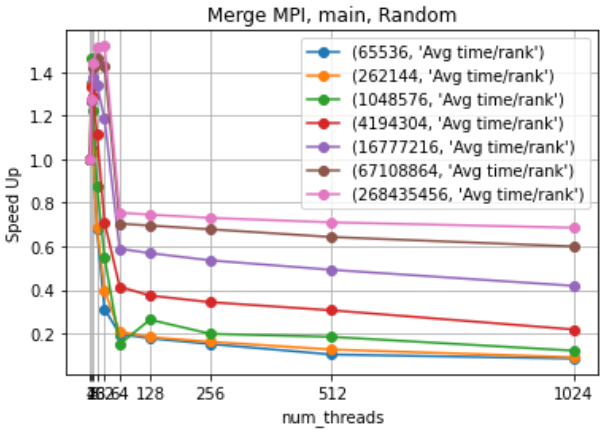


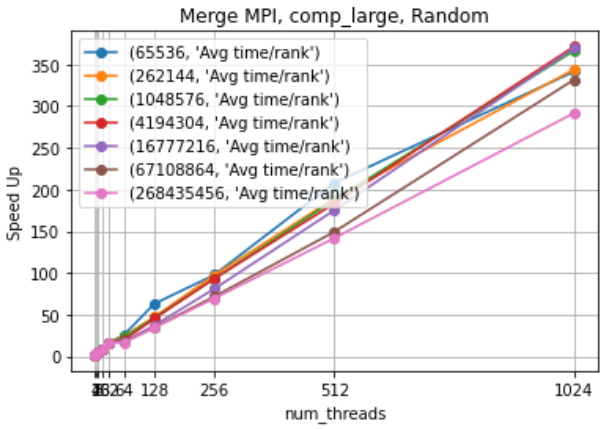


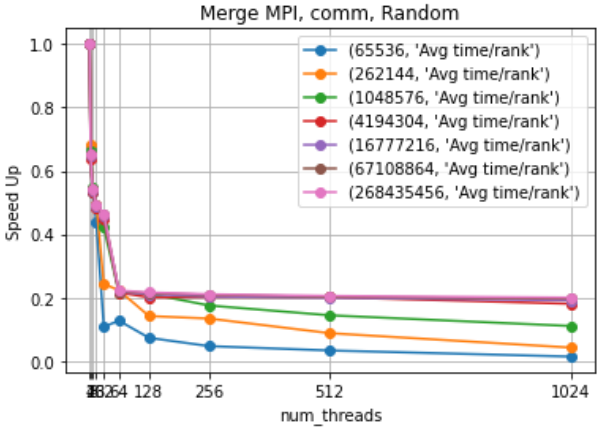


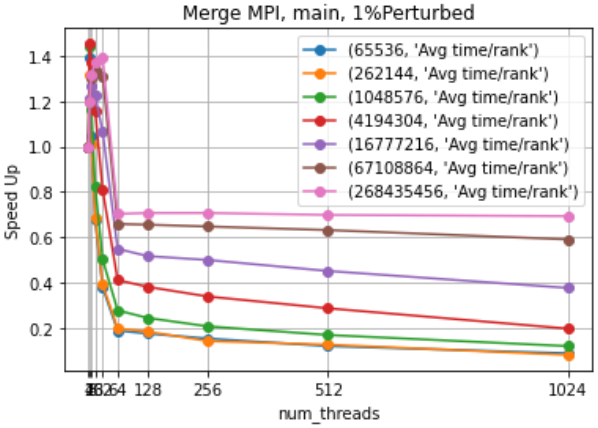


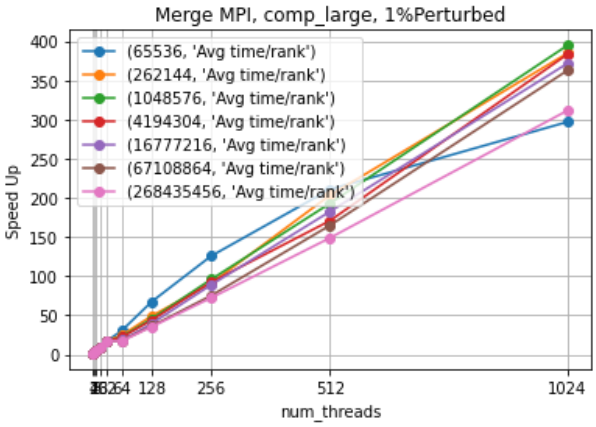


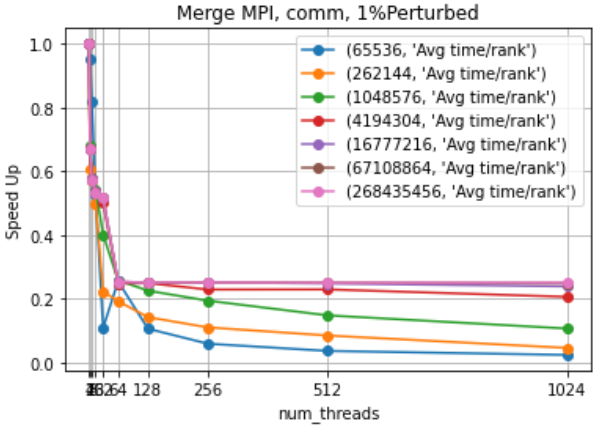






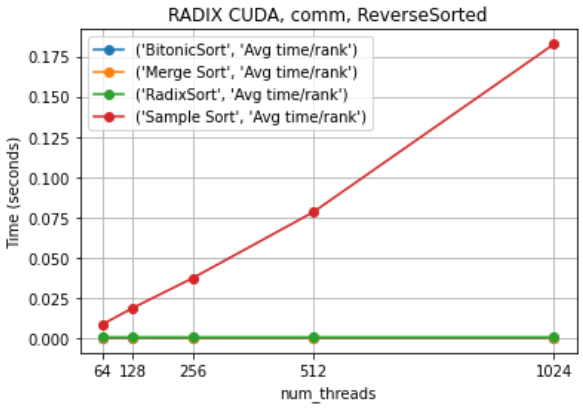


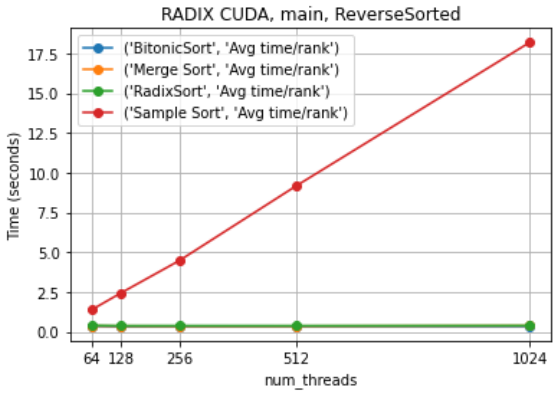


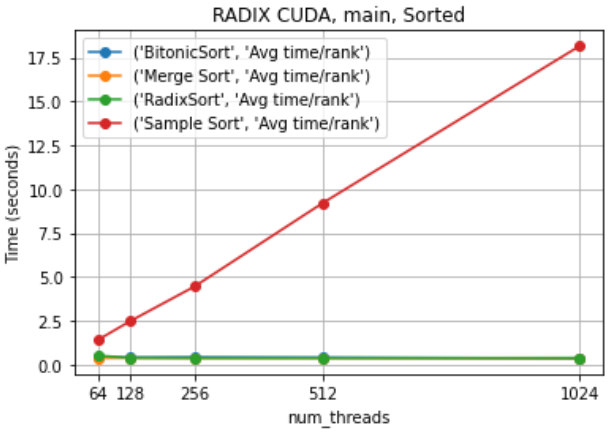


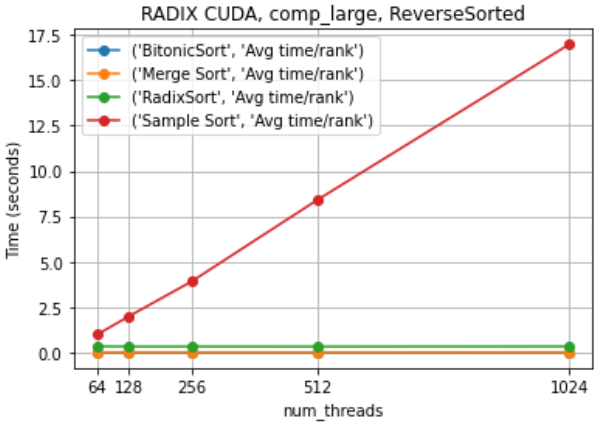
In the merge sort MPI implementation for speedup, the computation large shows a linearly increasing trend. This indicates that my implementation for merge sort has good scalability when it comes to computing and is really efficient. For communication, the speedup decreases really low and then maintains a constant speedup as the number of processors increases. Initially, the higher communication overhead is from the increased data exchange, but at a certain point(64 processors), the communication overhead becomes stable and more processors don't affect the communication time, which is why it is constant.

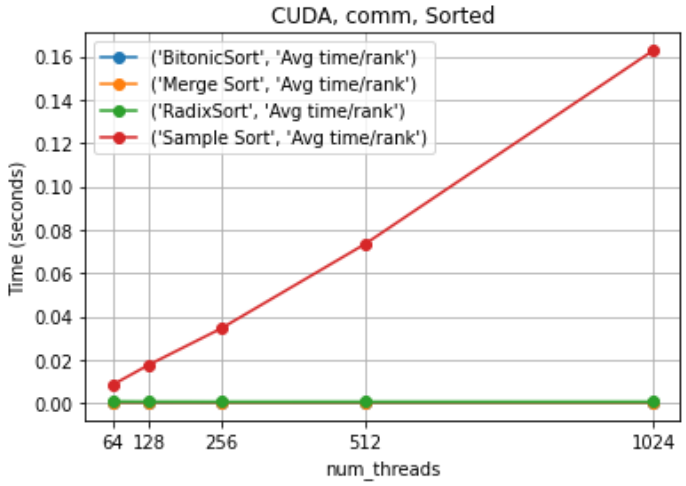
COMPARISONS CUDA

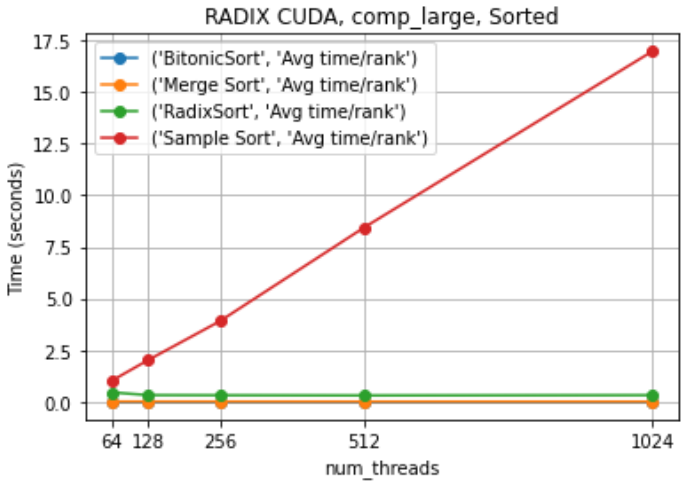


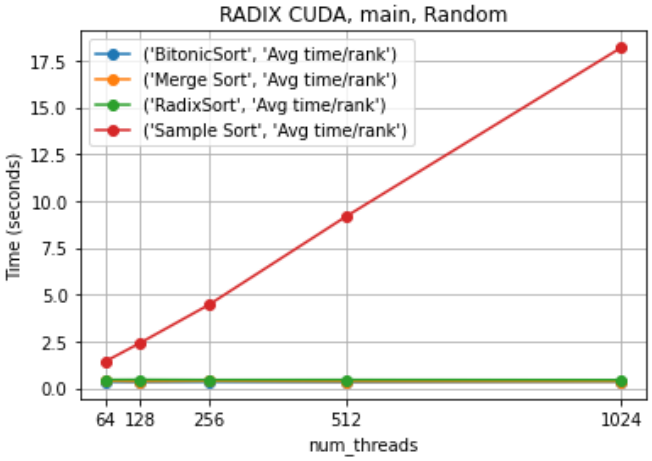


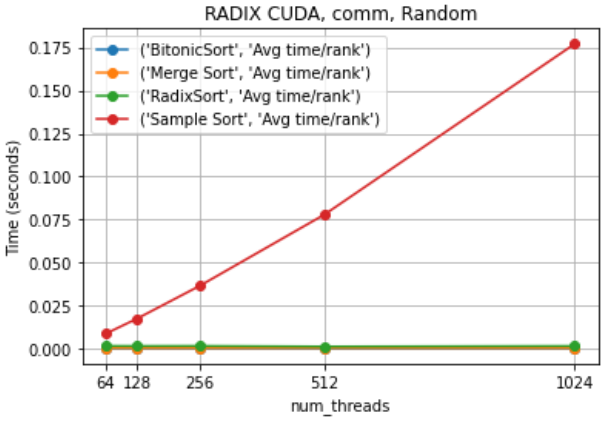


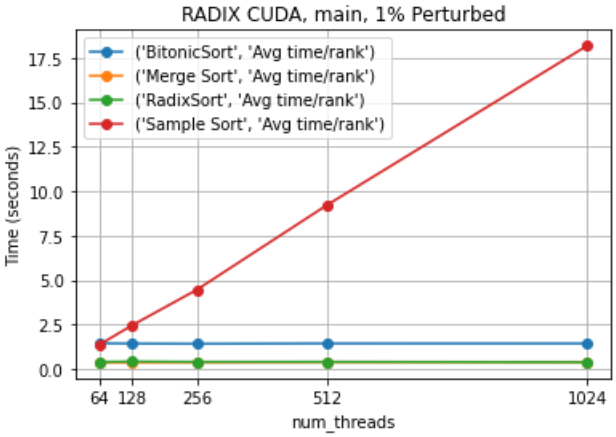


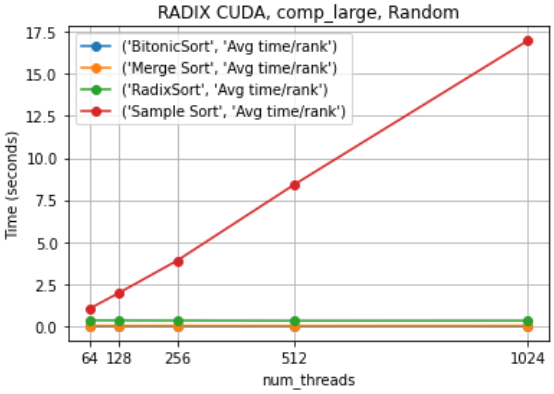


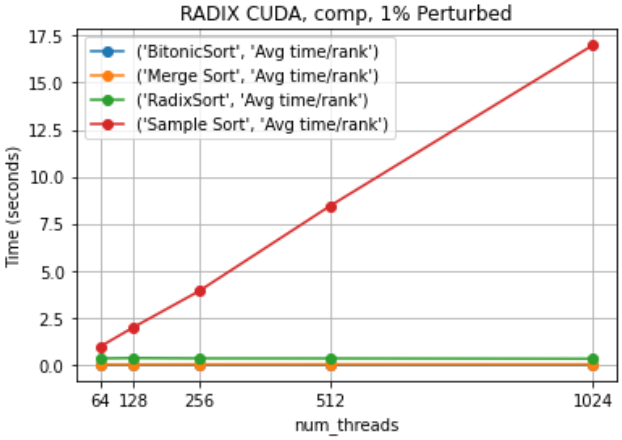


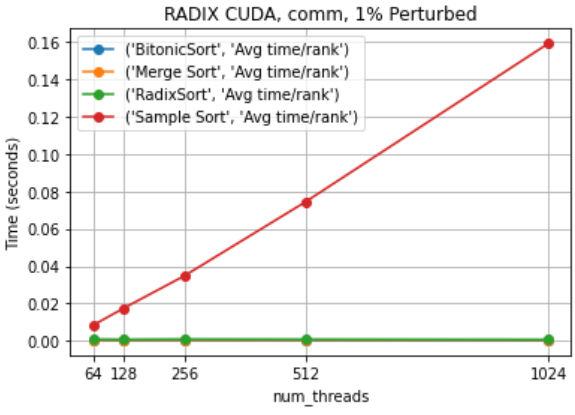

RADIX CUDA, main, Random
('BitonicSort', 'Avg time/rank')
('Merge Sort', 'Avg time/rank')
('RadixSort', 'Avg time/rank')
('Sample Sort', 'Avg time/rank')
Time (seconds)
num_threads


RADIX CUDA, comm, Random
('BitonicSort', 'Avg time/rank')
('Merge Sort', 'Avg time/rank')
('RadixSort', 'Avg time/rank')
('Sample Sort', 'Avg time/rank')
Time (seconds)
num_threads


RADIX CUDA, main, 1% Perturbed
('BitonicSort', 'Avg time/rank')
('Merge Sort', 'Avg time/rank')
('RadixSort', 'Avg time/rank')
('Sample Sort', 'Avg time/rank')
Time (seconds)
num_threads

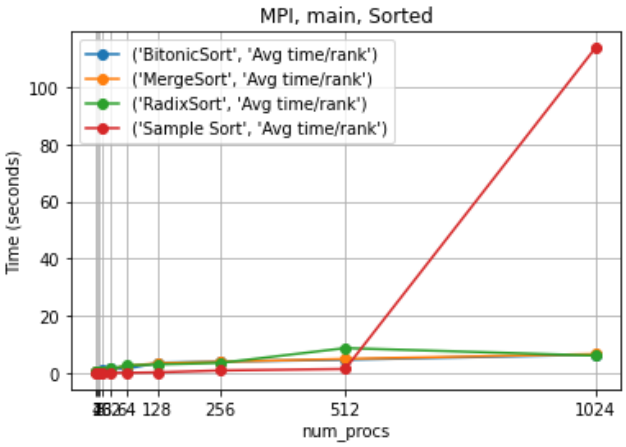

RADIX CUDA, comp_large, Random
('BitonicSort', 'Avg time/rank')
('Merge Sort', 'Avg time/rank')
('RadixSort', 'Avg time/rank')
('Sample Sort', 'Avg time/rank')
Time (seconds)
num_threads

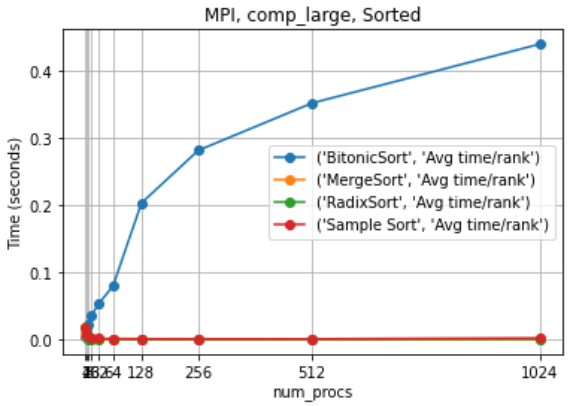

RADIX CUDA, comp, 1% Perturbed
('BitonicSort', 'Avg time/rank')
('Merge Sort', 'Avg time/rank')
('RadixSort', 'Avg time/rank')
('Sample Sort', 'Avg time/rank')
Time (seconds)
num_threads

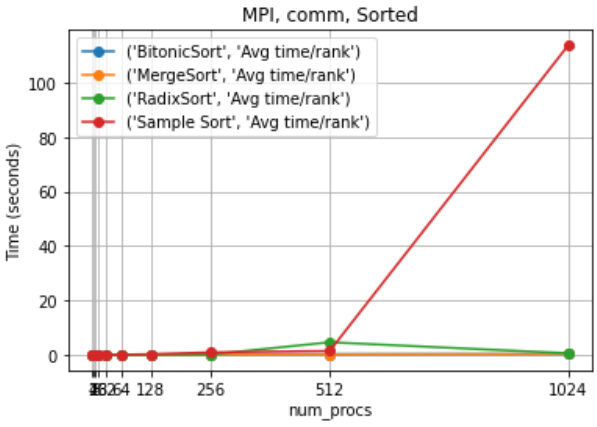

RADIX CUDA, comm, 1% Perturbed
('BitonicSort', 'Avg time/rank')
('Merge Sort', 'Avg time/rank')
('RadixSort', 'Avg time/rank')
('Sample Sort', 'Avg time/rank')
Time (seconds)
num_threads

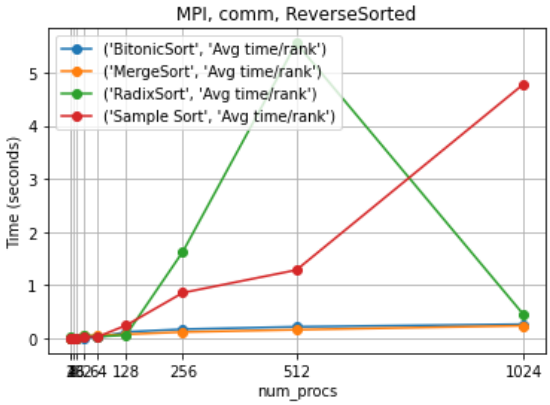
For our four sorting algorithms, bitonic, radix, merge, and sample sort, the Cuda implementations took similar times for radix, merge, and bitonic, with radix taking the most time. However, sample sort took significantly more time than the other three. Although most of the implementations did not scale well to the degree that we hoped for an increased number of threads, sample sort specifically was not parallelized as well for Cuda due to the above explanations. These graphs show that for our implementations, bitonic sort and merge sort scale the best in Cuda. Note: some the titles specify Radix, but the data is correct for all 4 sorts

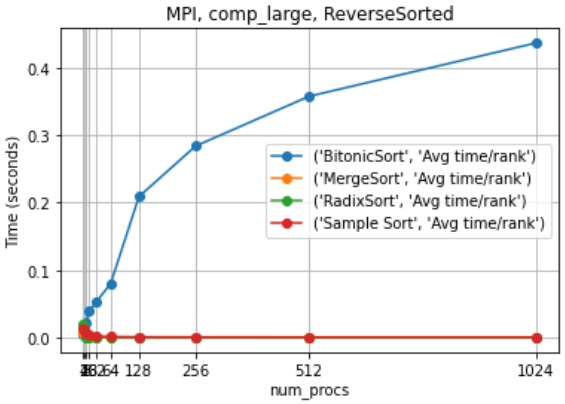
COMPARISONS MPI

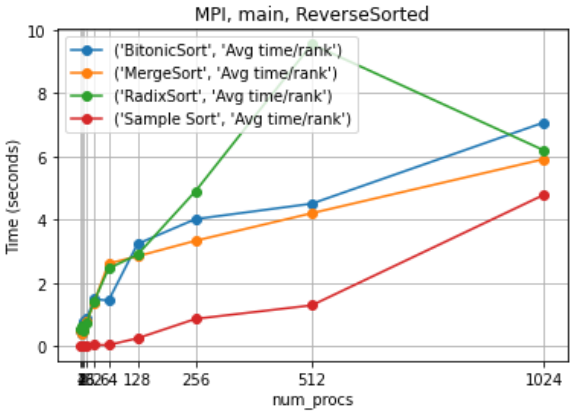


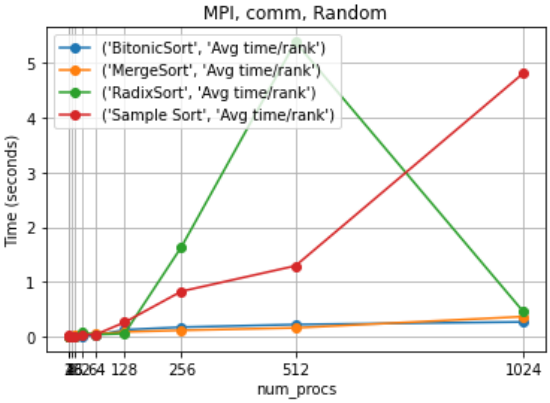


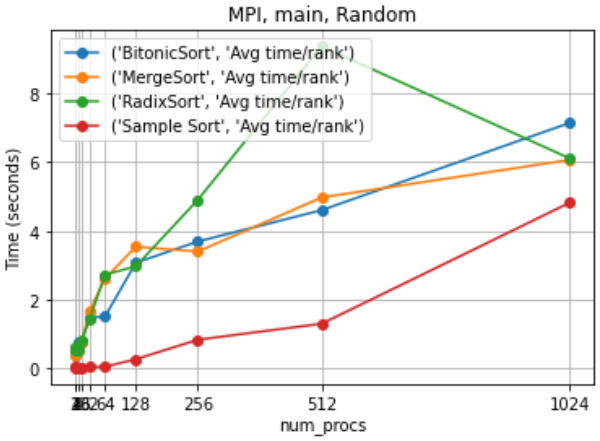


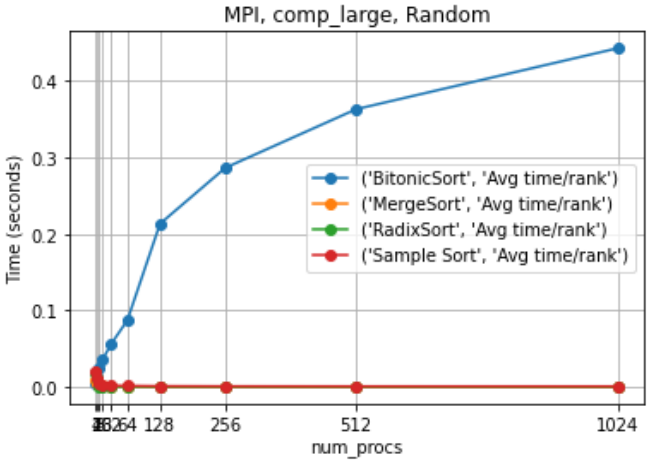


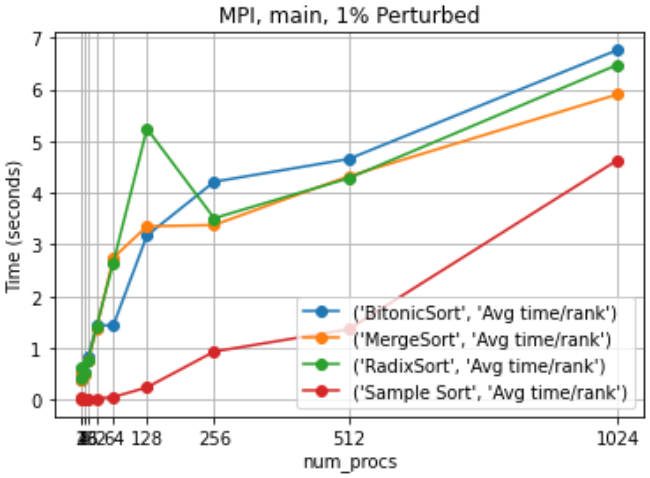

MPI, comp_large, ReverseSorted
Time (seconds)
num_procs
('BitonicSort', 'Avg time/rank')
('MergeSort', 'Avg time/rank')
('RadixSort', 'Avg time/rank')
('Sample Sort', 'Avg time/rank')

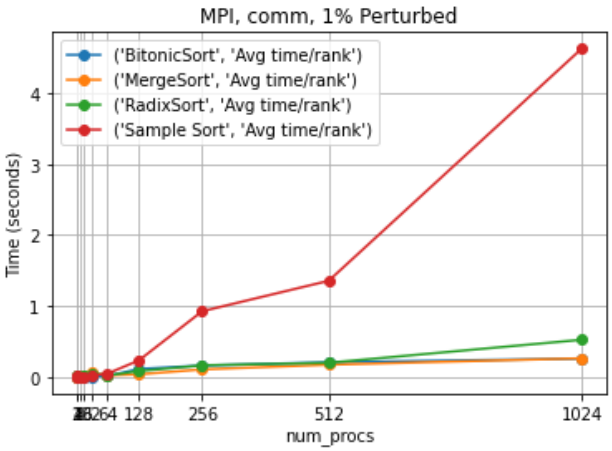

MPI, main, ReverseSorted
Time (seconds)
num_procs
('BitonicSort', 'Avg time/rank')
('MergeSort', 'Avg time/rank')
('RadixSort', 'Avg time/rank')
('Sample Sort', 'Avg time/rank')

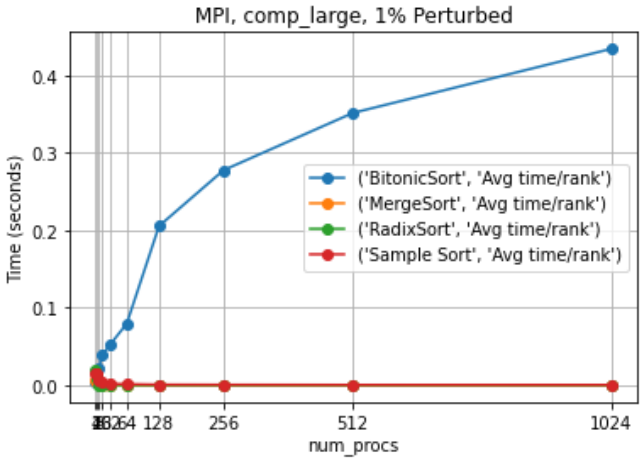

MPI, comm, Random
Time (seconds)
num_procs
('BitonicSort', 'Avg time/rank')
('MergeSort', 'Avg time/rank')
('RadixSort', 'Avg time/rank')
('Sample Sort', 'Avg time/rank')


MPI, main, Random
Time (seconds)
num_procs
('BitonicSort', 'Avg time/rank')
('MergeSort', 'Avg time/rank')
('RadixSort', 'Avg time/rank')
('Sample Sort', 'Avg time/rank')


MPI, comp_large, Random
Time (seconds)
num_procs
('BitonicSort', 'Avg time/rank')
('MergeSort', 'Avg time/rank')
('RadixSort', 'Avg time/rank')
('Sample Sort', 'Avg time/rank')


MPI, main, 1% Perturbed
Time (seconds)
num_procs
('BitonicSort', 'Avg time/rank')
('MergeSort', 'Avg time/rank')
('RadixSort', 'Avg time/rank')
('Sample Sort', 'Avg time/rank')





The comparisons between the four sorts differ a little more for the MPI implementations than Cuda. The communication time is greater again for sample sort by a lot followed by radix, and then bitonic and merge. However, the computation times for bitonic sort MPI far exceed any of the other sorting algorithms. Overall, for the main program, bitonic has the worst time followed by radix and merge and then sample sort has the best time by a fair amount. This means that for MPI, sample sort was parallelized the most efficiently and bitonic the least efficiently.