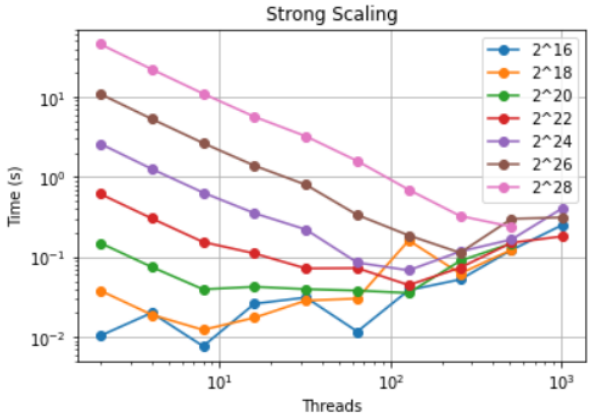
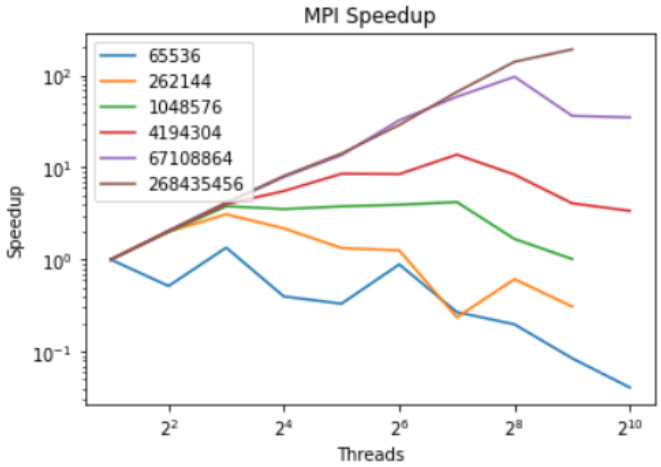
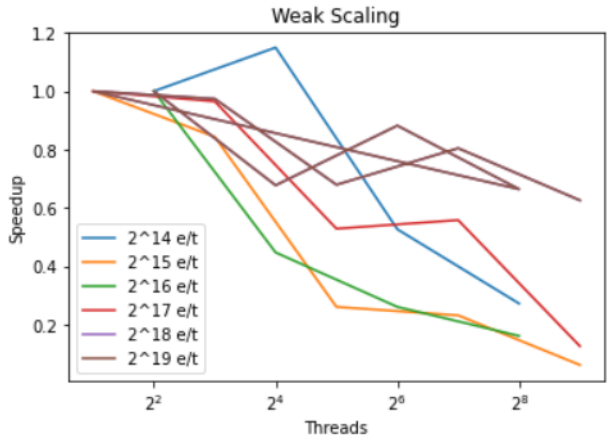
Bucket Sort MPI

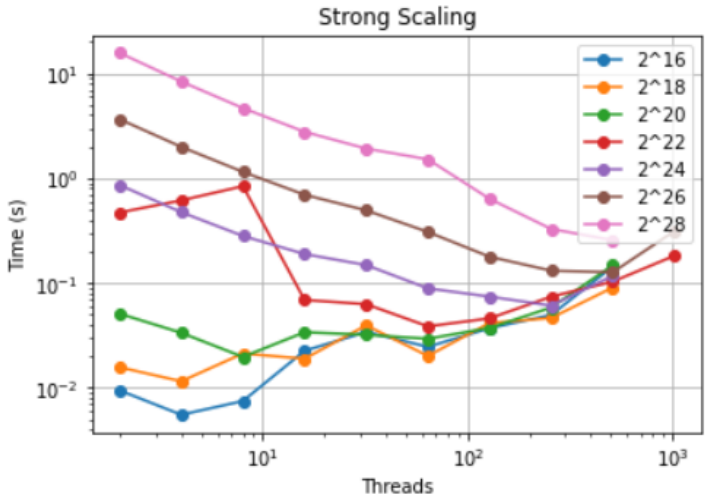
Random:

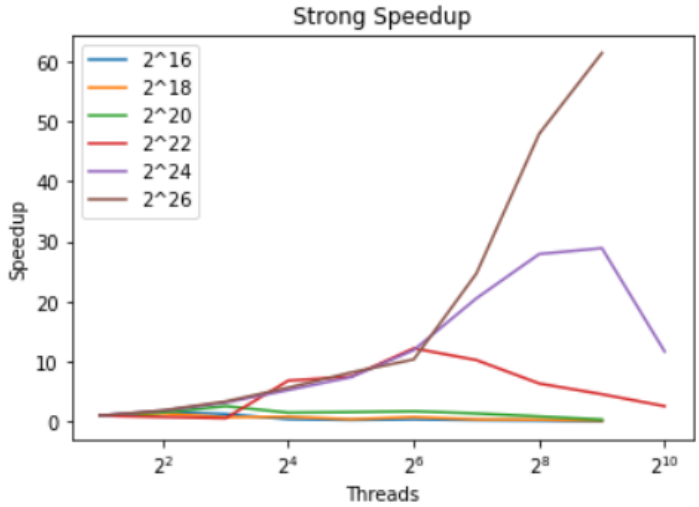


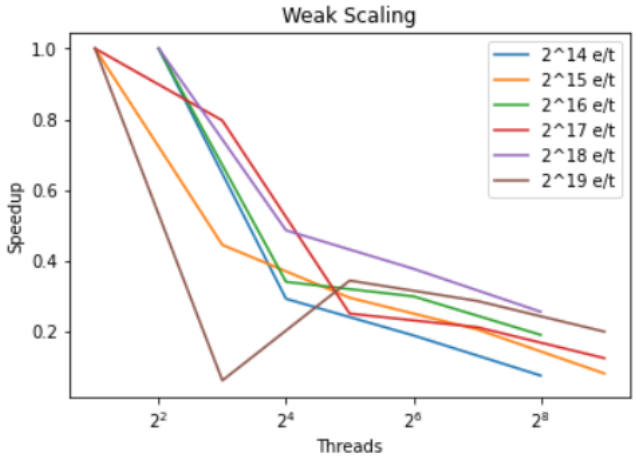




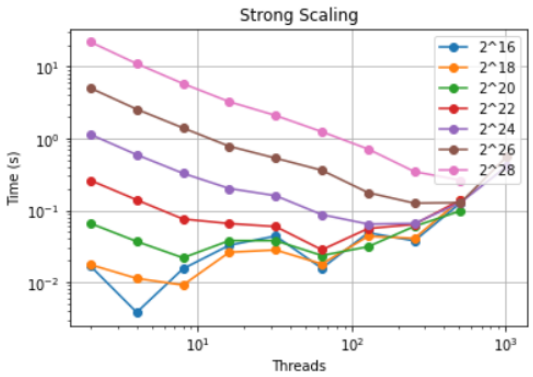
Sorted:

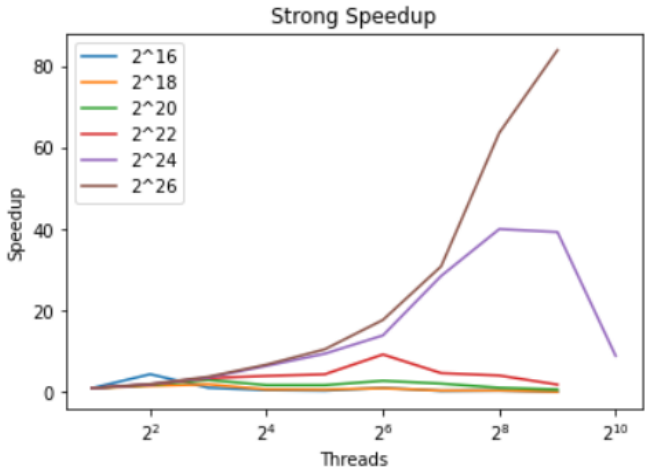


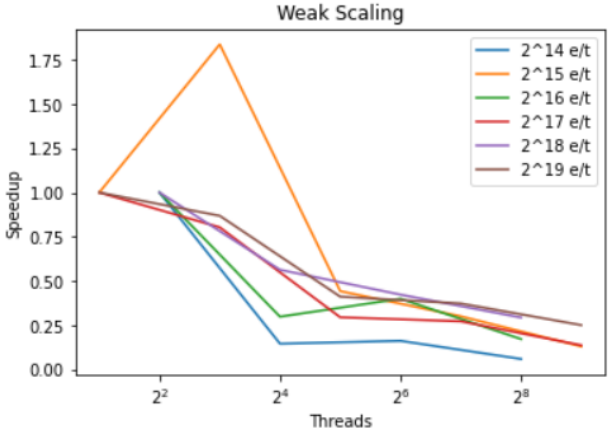




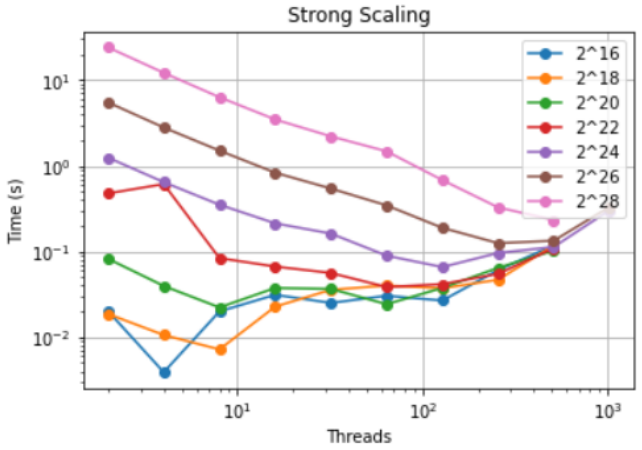
Reverse Sorted:

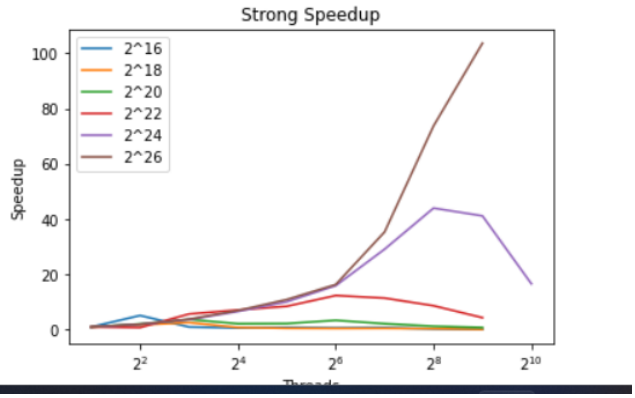


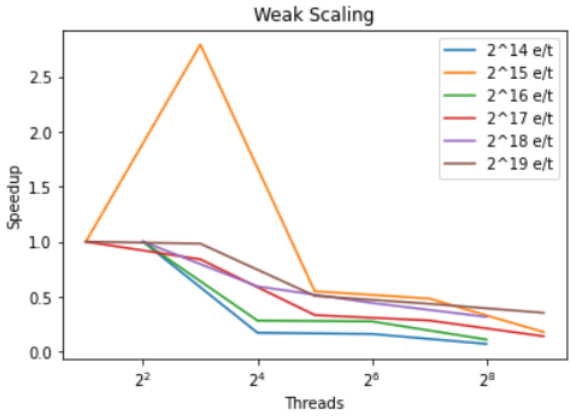




Sorted + 1% perturbed:

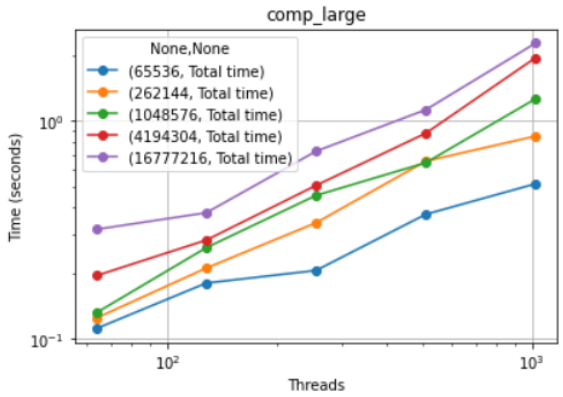

Strong Scaling
Time (s)
Threads
2^16
2^18
2^20
2^22
2^24
2^26
2^28

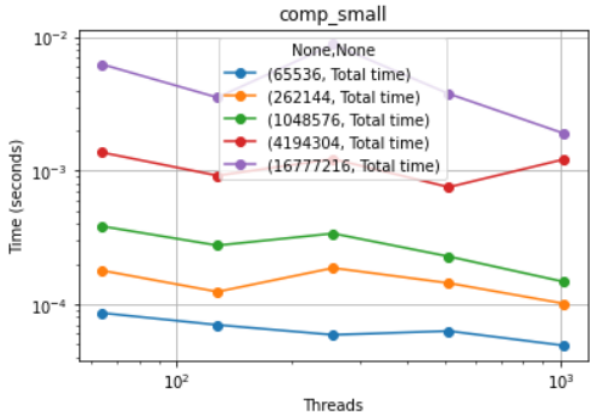

Strong Speedup
Speedup
2^16
2^18
2^20
2^22
2^24
2^26

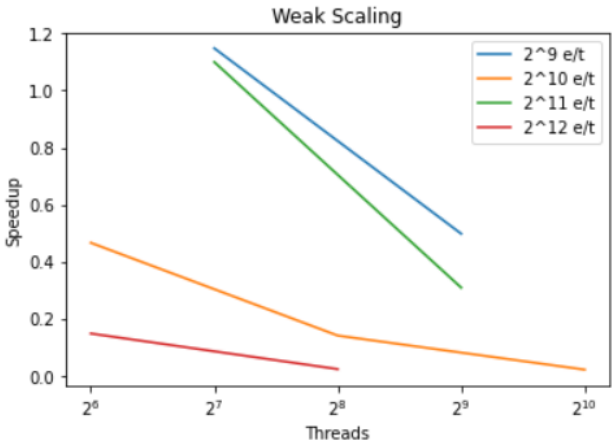

Weak Scaling
Speedup
Threads
2^14 e/t
2^15 e/t
2^16 e/t
2^17 e/t
2^18 e/t
2^19 e/t

Bucket Sort CUDA

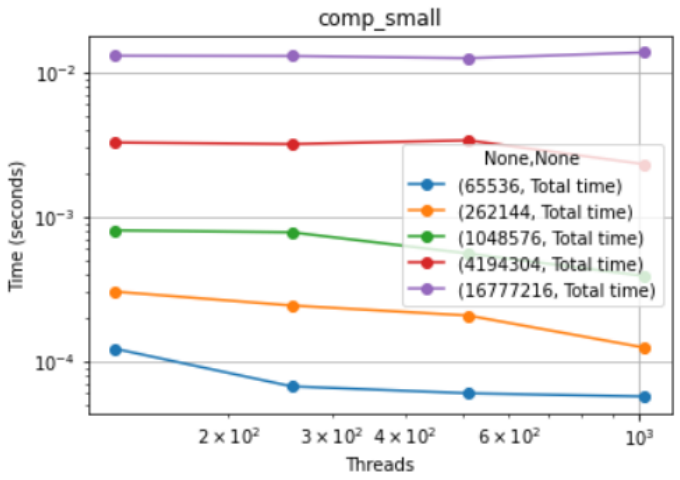
Random:

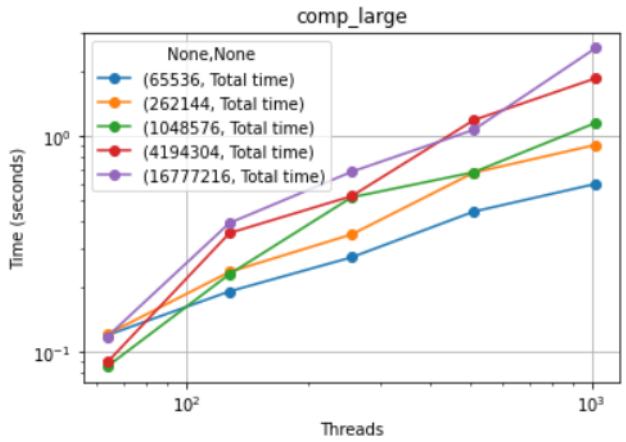


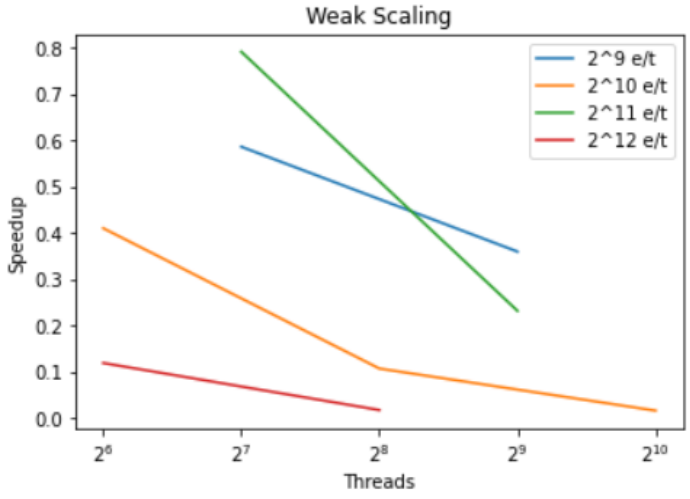




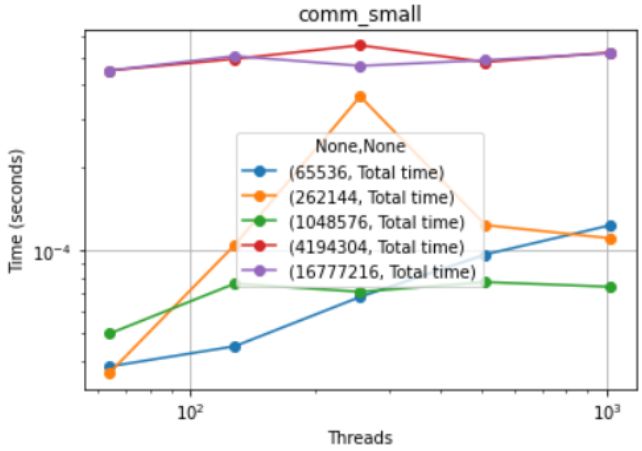
Sorted:

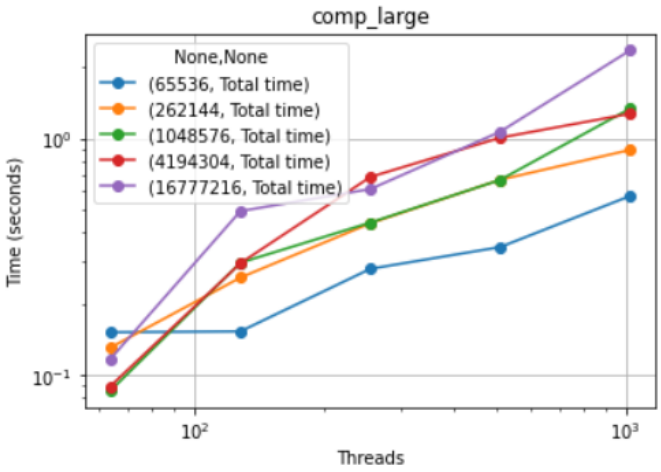


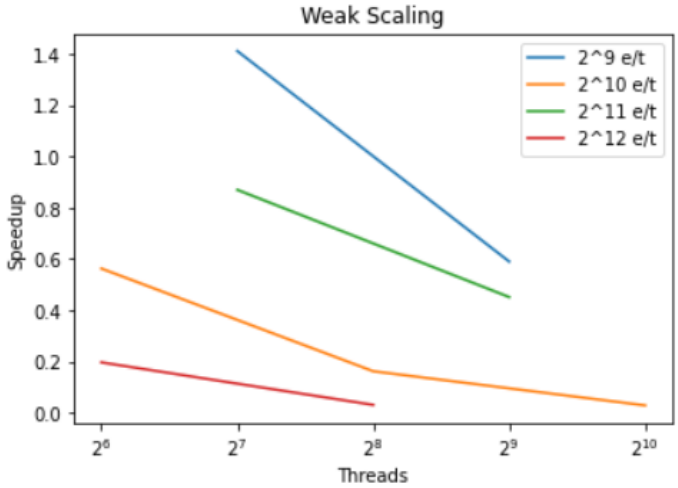




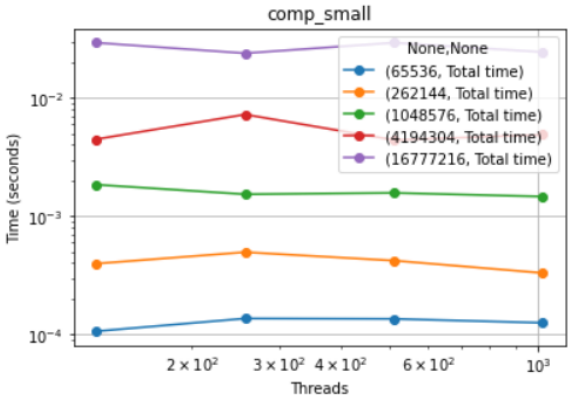
Reverse Sorted:

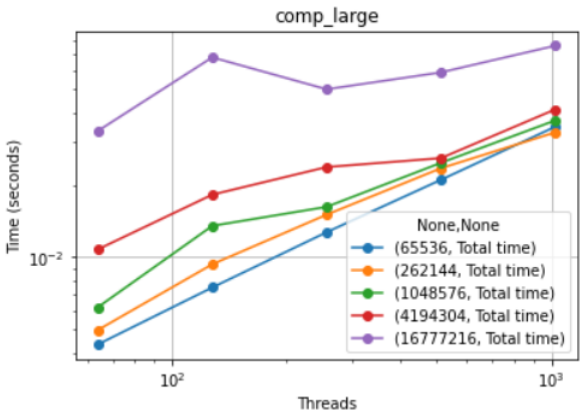


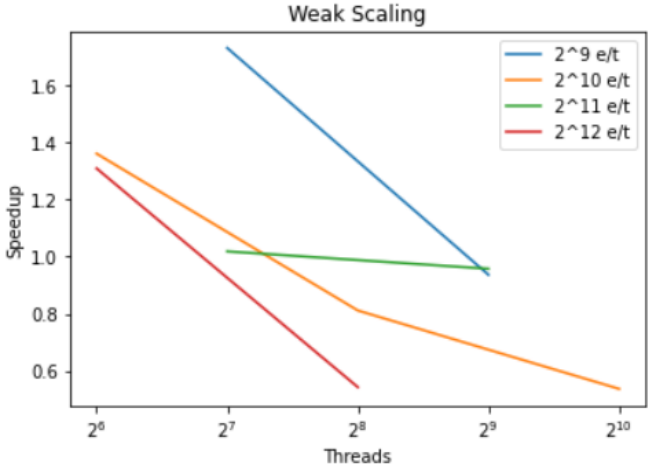




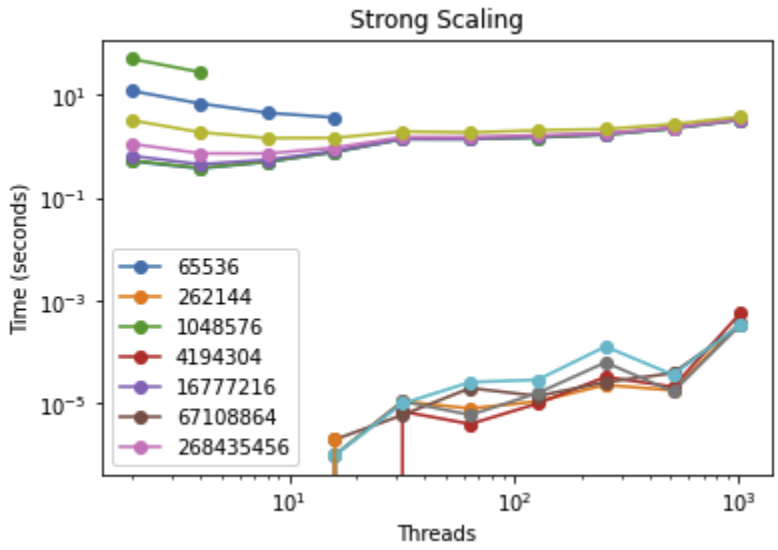
Sorted + 1% Perturbed:

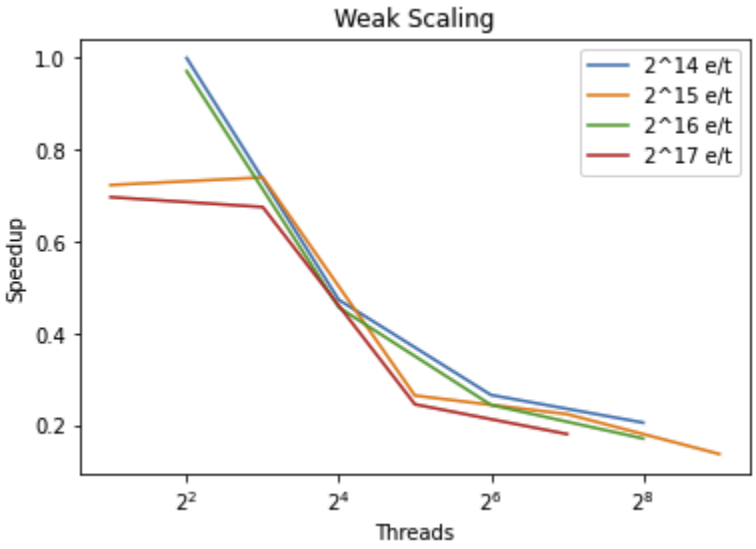

comp_small
None,None
(65536, Total time)
(262144, Total time)
(1048576, Total time)
(4194304, Total time)
(16777216, Total time)
Time (seconds)
Threads


comp_large
None,None
(65536, Total time)
(262144, Total time)
(1048576, Total time)
(4194304, Total time)
(16777216, Total time)
Time (seconds)
Threads

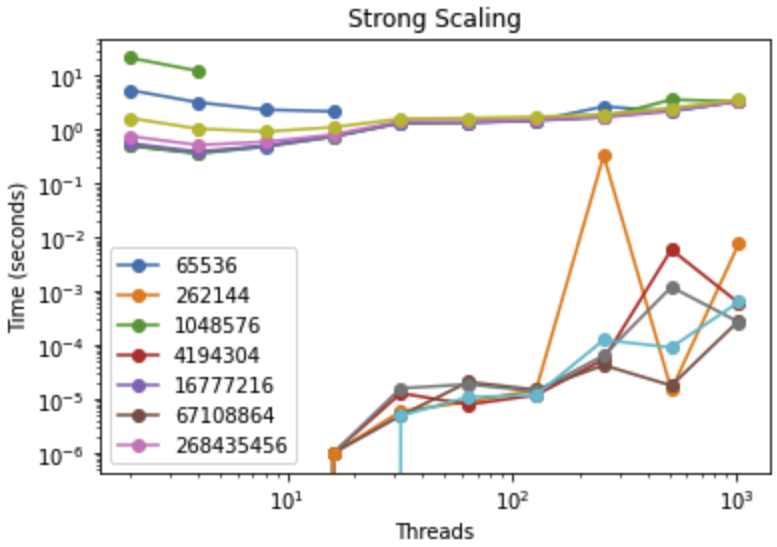

Weak Scaling
2^9 e/t
2^10 e/t
2^11 e/t
2^12 e/t
Speedup
Threads

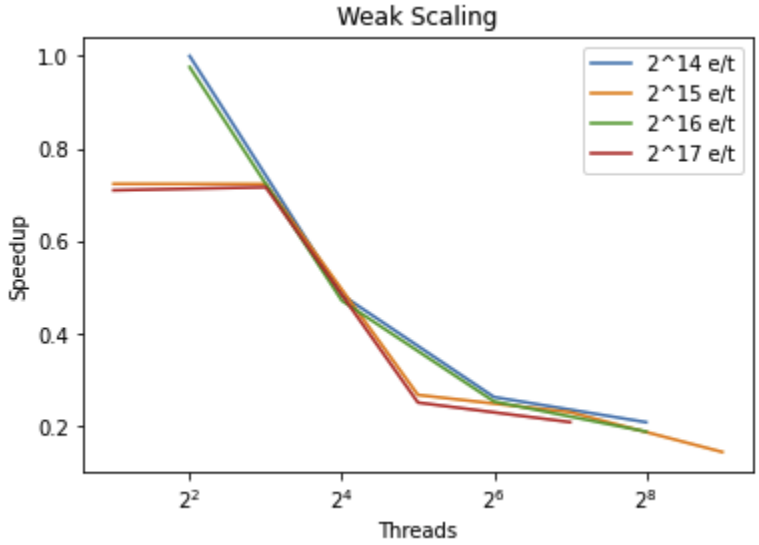
Quicksort mpi
Random:



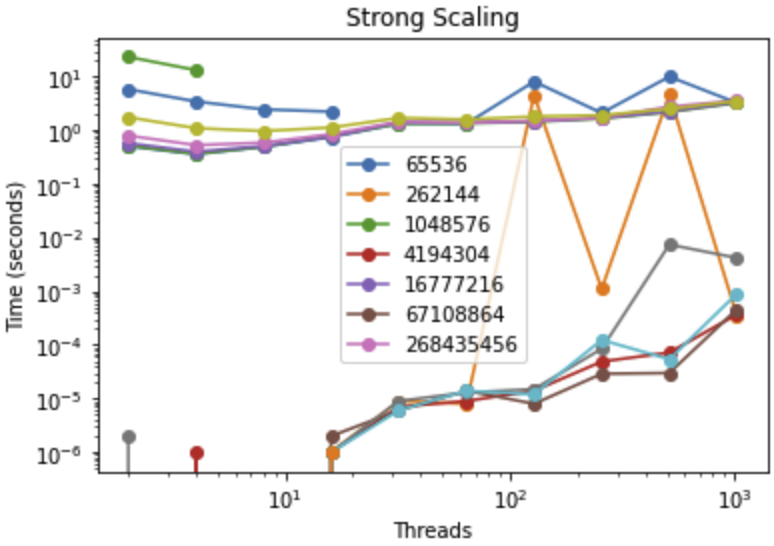


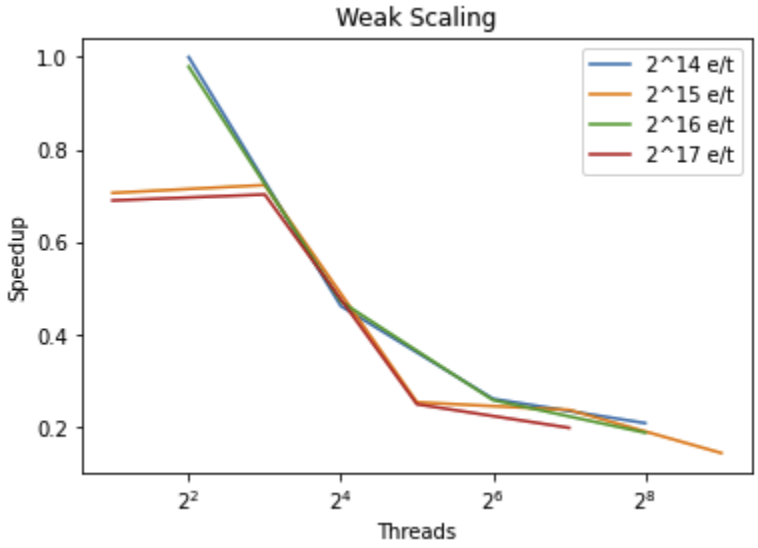
Sorted:


Strong Scaling
Time (seconds)
Threads
65536
262144
1048576
4194304
16777216
67108864
268435456

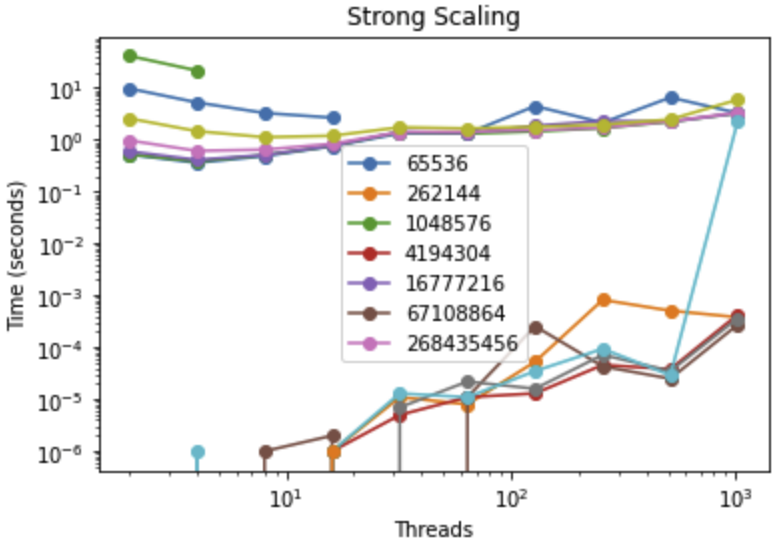

Weak Scaling
Speedup
Threads
2^14 e/t
2^15 e/t
2^16 e/t
2^17 e/t

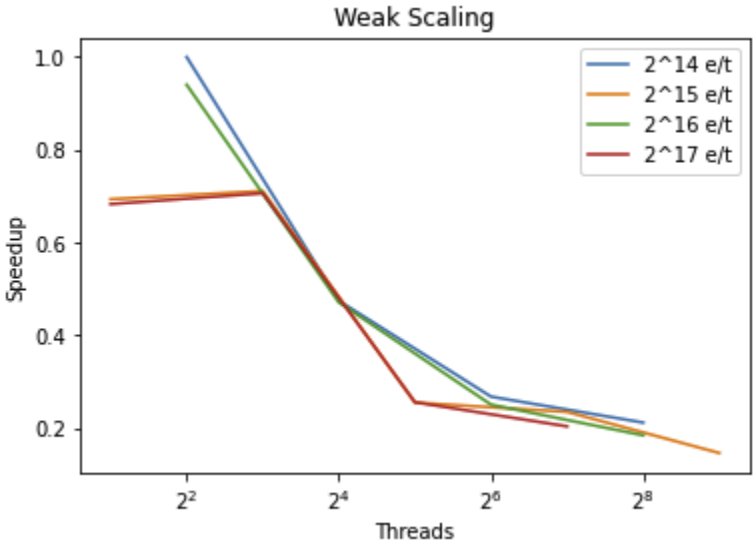
2:


Strong Scaling
Time (seconds)
Threads
65536
262144
1048576
4194304
16777216
67108864
268435456

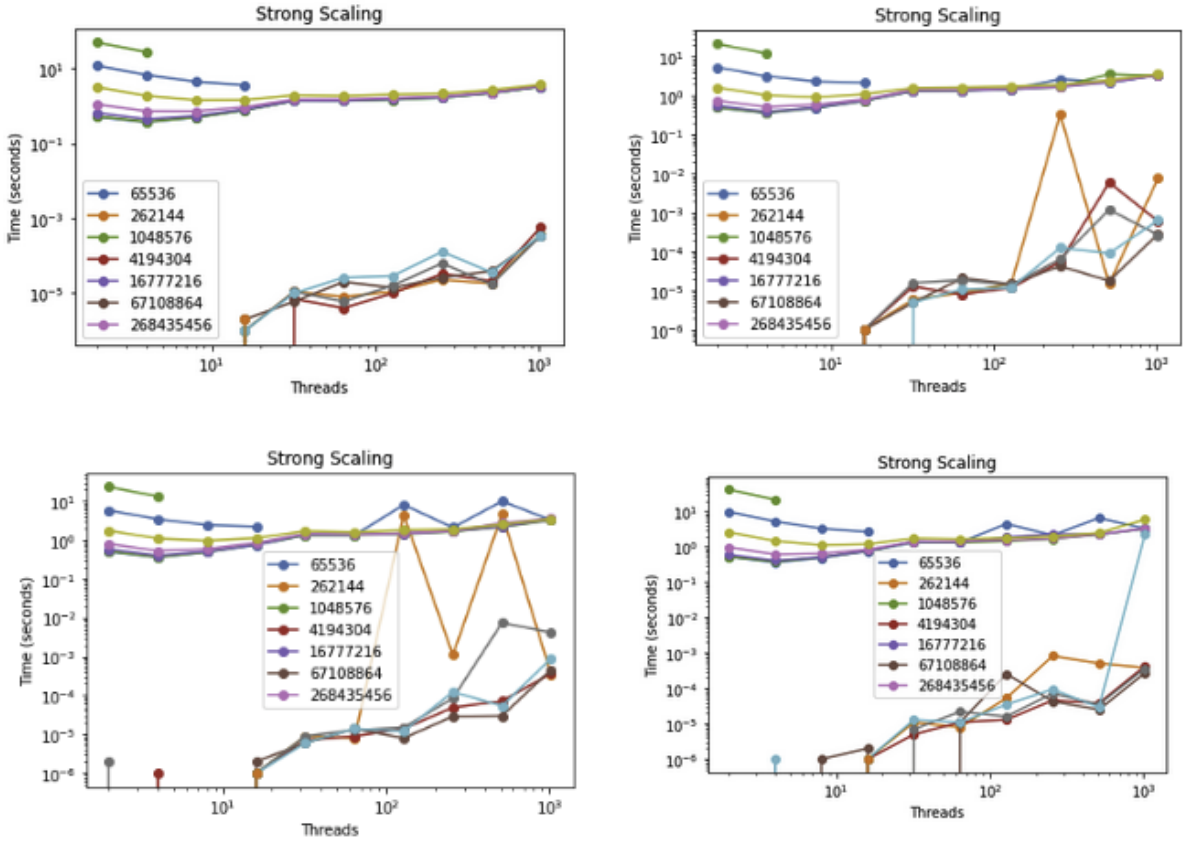

Weak Scaling
Speedup
Threads
2^14 e/t
2^15 e/t
2^16 e/t
2^17 e/t

3:

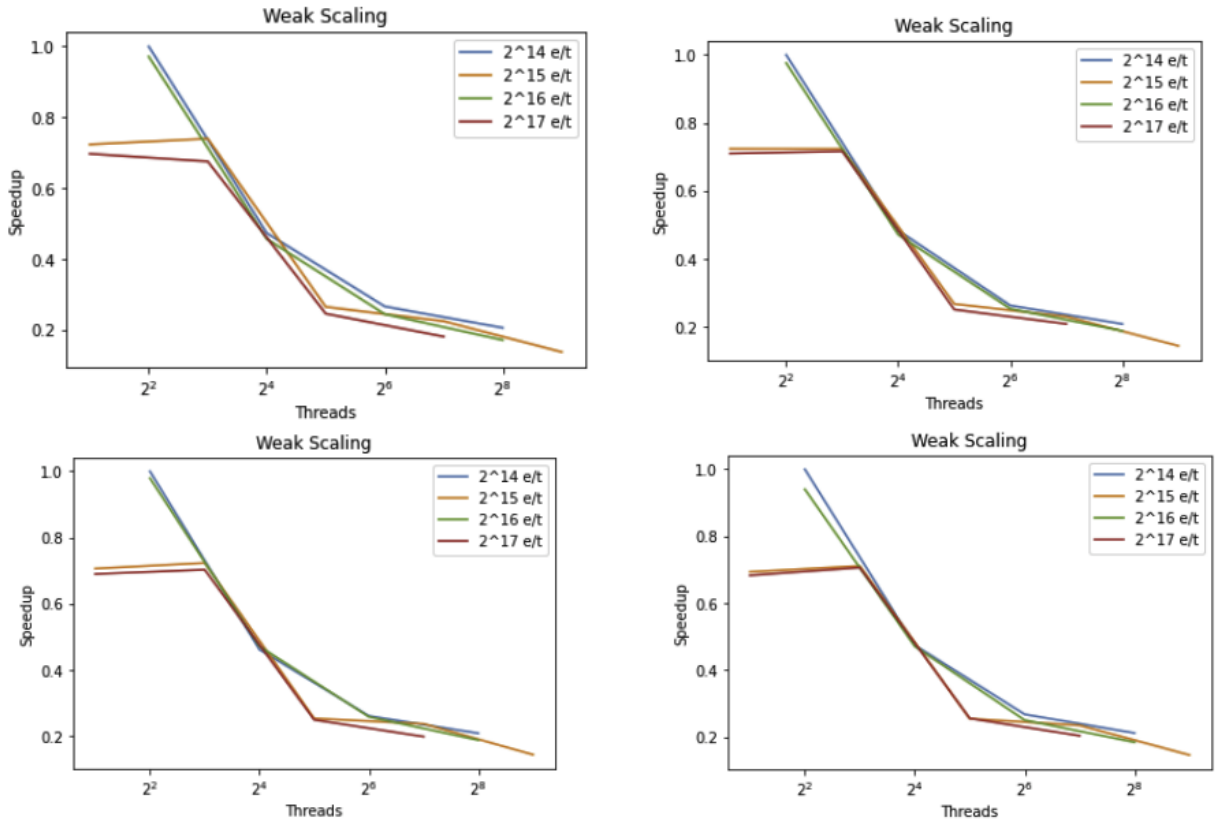




Strong scaling analysis:

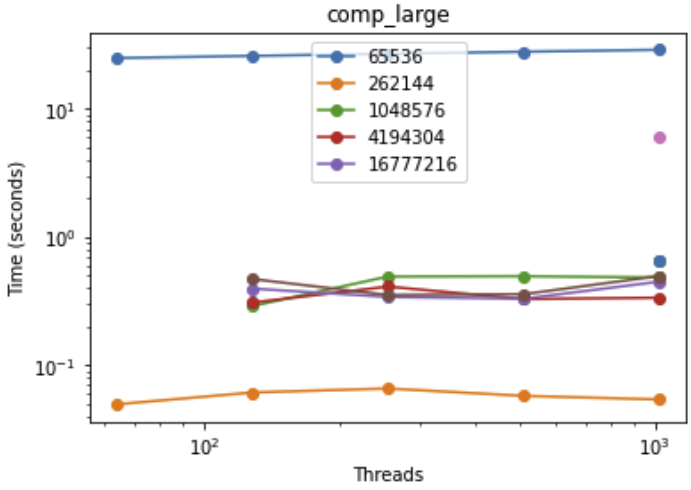


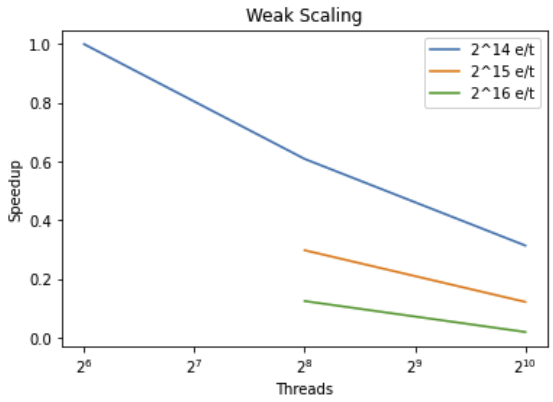
From the graphs above we can see that the mpi implementation of quicksort did an awful job at parallelizing. For some of the array sizes it had a constant time meaning it didnt parallelize while for others it increased in time with the number of threads however we would have expected it to be decreasing. Quicksort is a sequential algorithm that we tried to parallelize but upon seeing the results we either should do a much better job or just not parallelize it at all. Overall, it seems like the mode affected the performance pretty low, but it seems like it performed the best/most consistently with random setting.



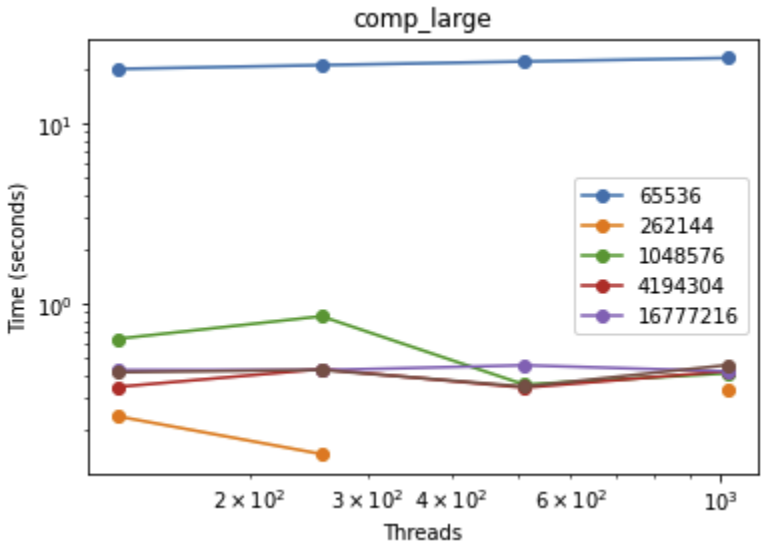
As for weak scaling, again we see the opposite trend of what we would have expected, meaning the performance of the algorithm is just bad in general. There isn't much difference between the performance in terms of modes, so I don't think the mode made much difference in it.

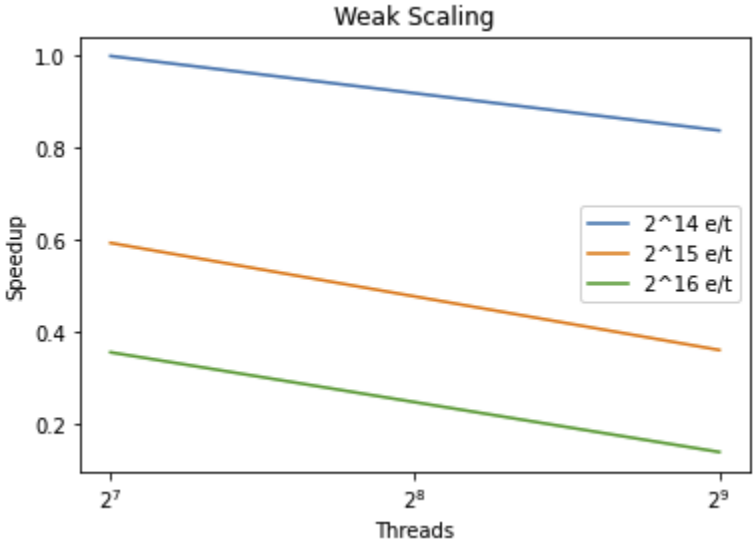
CUDA:
Random:



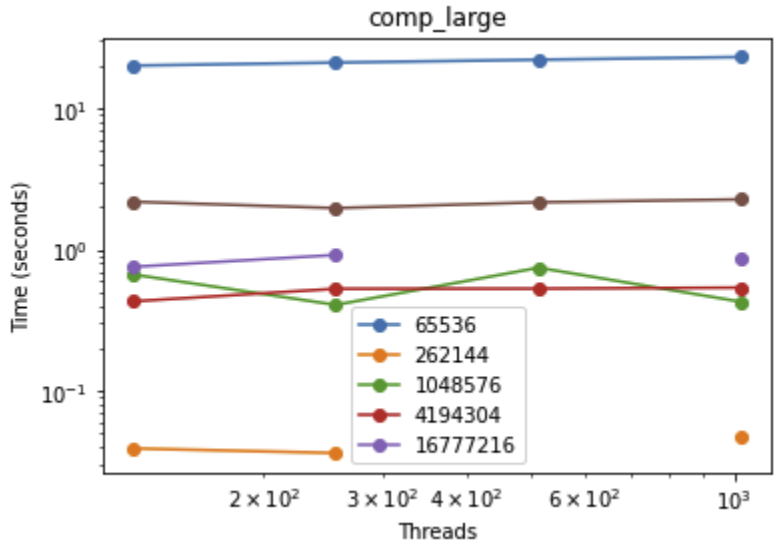

Weak Scaling
2^14 e/t
2^15 e/t
2^16 e/t
Speedup
1.0
0.8
0.6
0.4
0.2
0.0
2^6
2^7
2^8
2^9
2^{10}
Threads

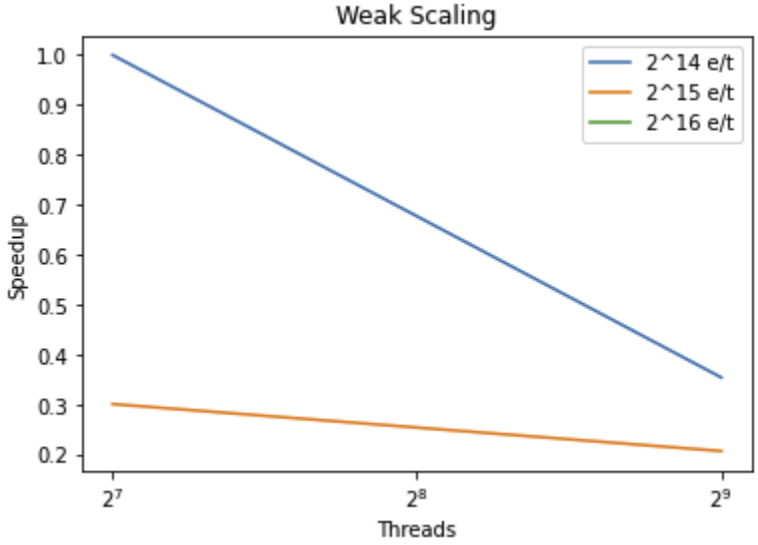
1:


comp_large
Time (seconds)
10^1
10^0
65536
262144
1048576
4194304
16777216
2×10^2
3×10^2
4×10^2
6×10^2
10^3
Threads

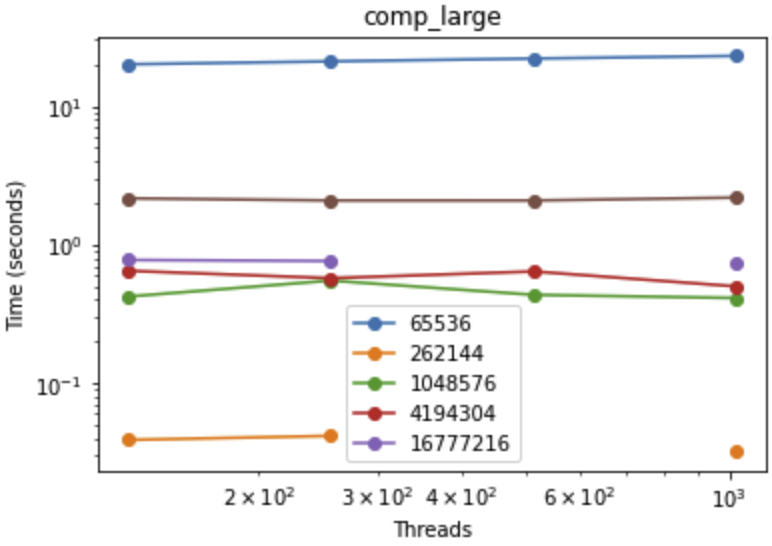

Weak Scaling
Speedup
1.0
0.8
0.6
0.4
0.2
2^14 e/t
2^15 e/t
2^16 e/t
2^7
2^8
2^9
Threads

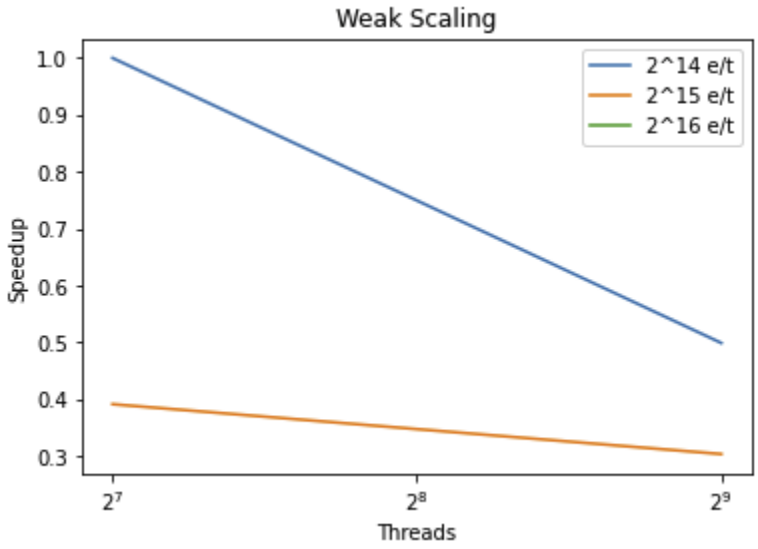
2:


comp_large
Time (seconds)
Threads
65536
262144
1048576
4194304
16777216

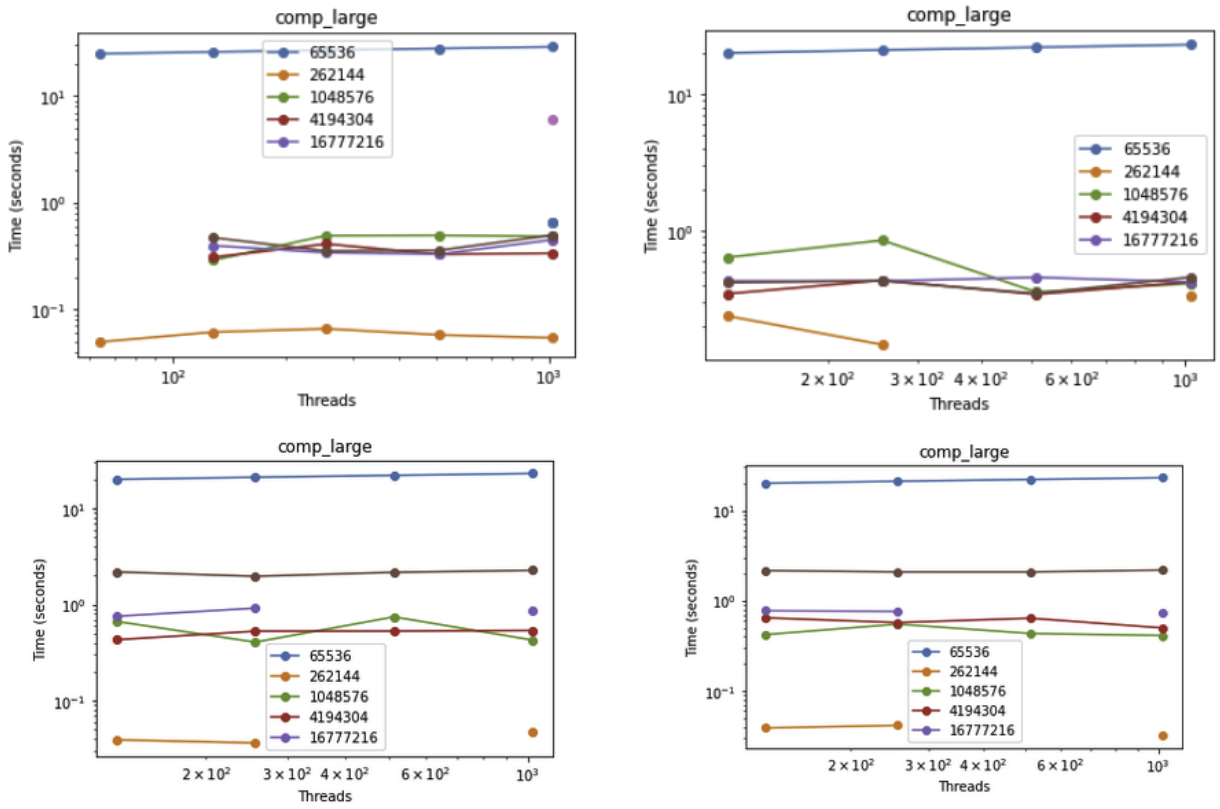

Weak Scaling
Speedup
Threads
2^14 e/t
2^15 e/t
2^16 e/t

3:

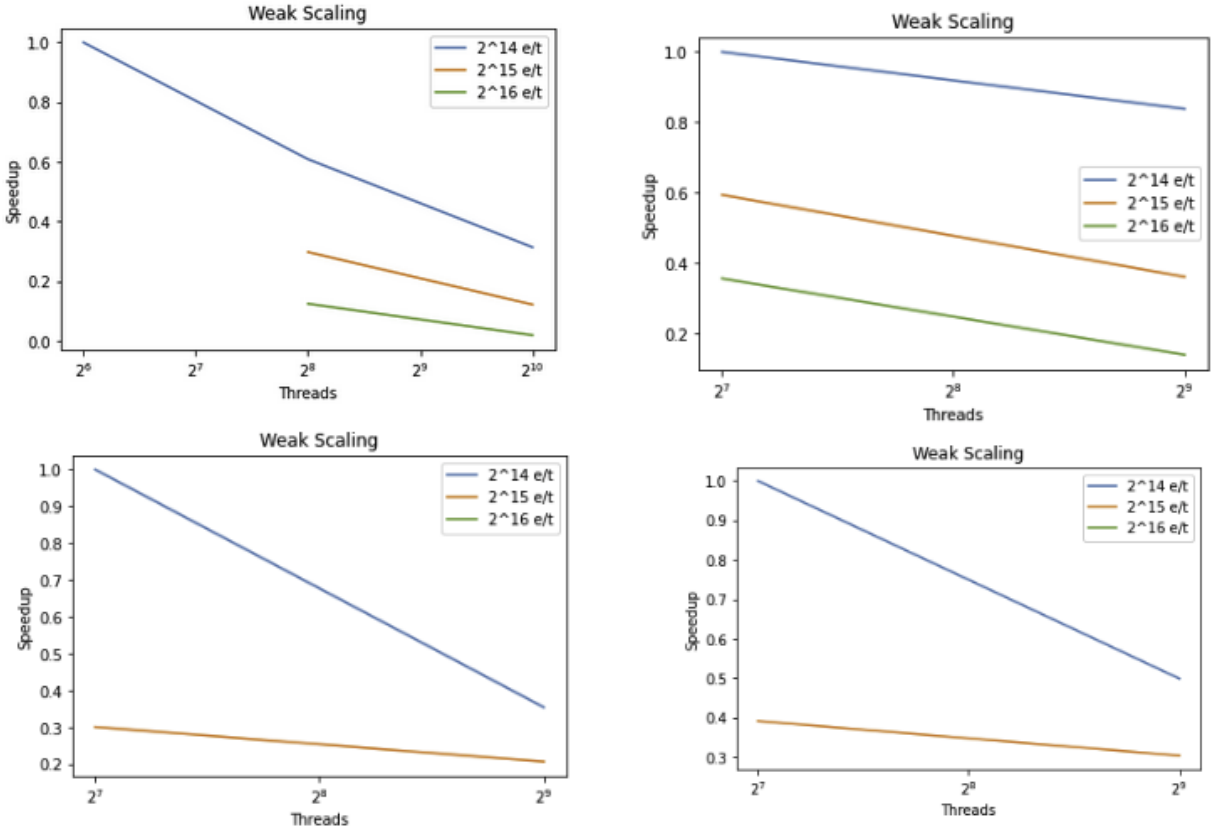

comp_large
Time (seconds)
Threads
65536
262144
1048576
4194304
16777216


Weak Scaling
Speedup
Threads
2^14 e/t
2^15 e/t
2^16 e/t

Analysis:



Overall, by looking at the strong scaling graphs it seems as though the program did not really parallelize. We would have expected the time to decrease as the number of threads increased, which would make sense, unfortunately that wasn't the case for our program. Upon comparing all 4 modes, it seems like mode 3 was the most consistent across all modes, leaving mode 1(top right) to be the most varying.



Again, as we saw in MPI implementation, weak scaling is just bad overall. We expected it to have a positive linear relationship but we mostly see a negative relationship between them or no relationship at all, my guess is best mode is 1(top right).

Bubble Sort:

Random MPI:

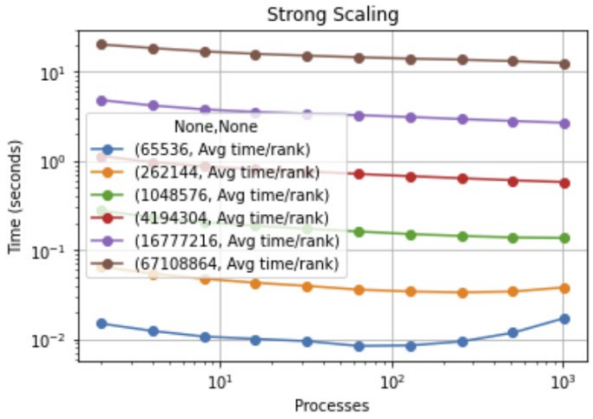


Figure 1: Comp

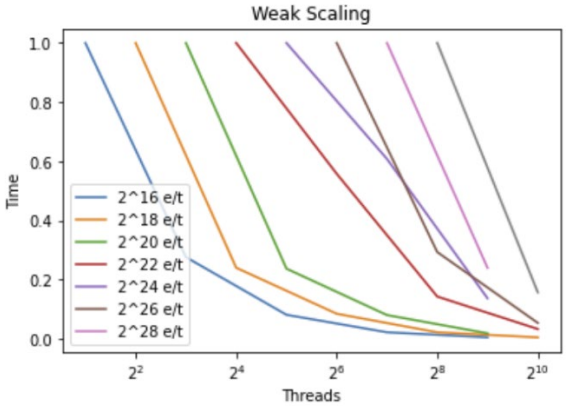
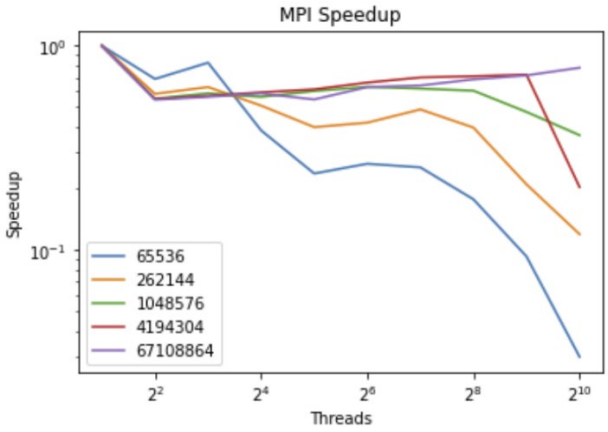


Figure 2: Main



Sorted MPI:

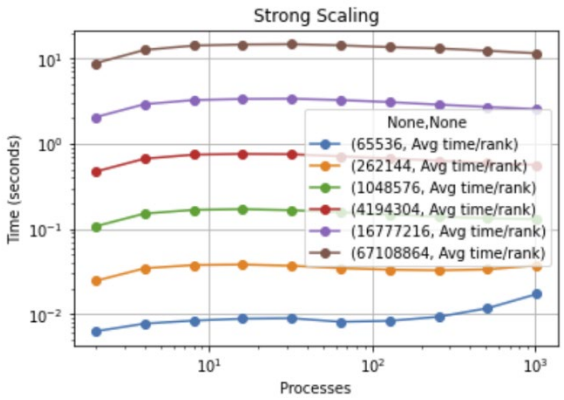


Figure 3: Comp

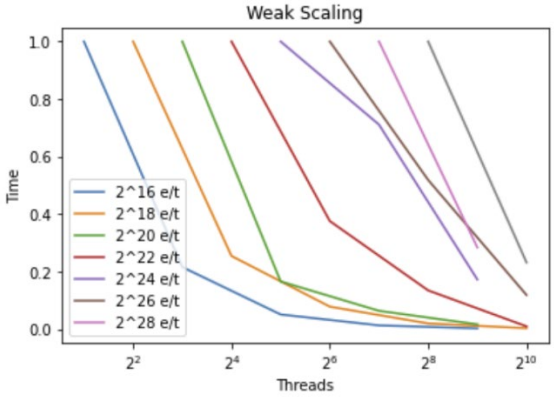
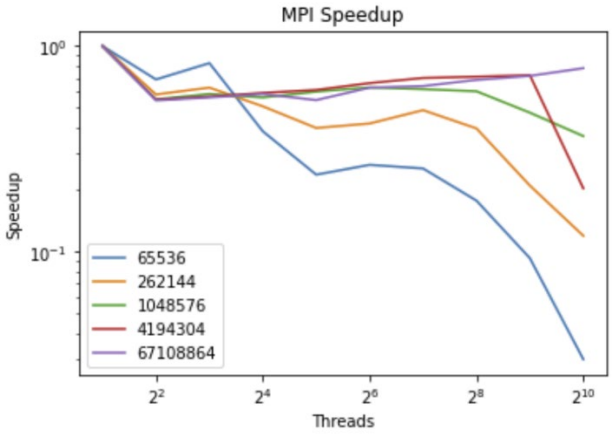


Figure 4: Main



Reverse Sorted MPI:

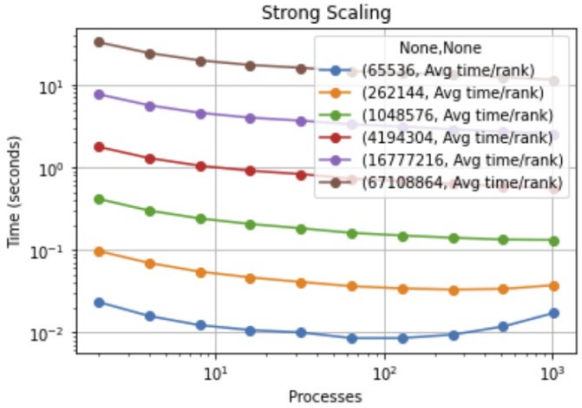


Figure 5: Comp

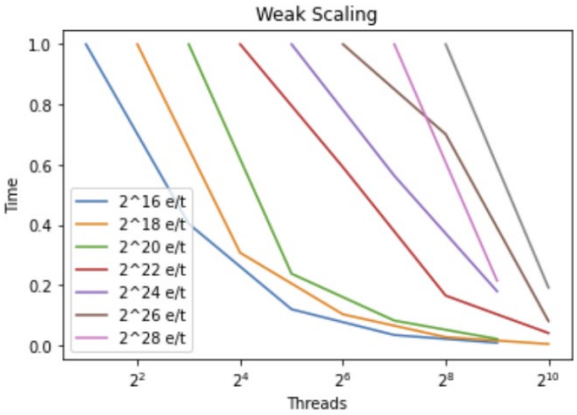
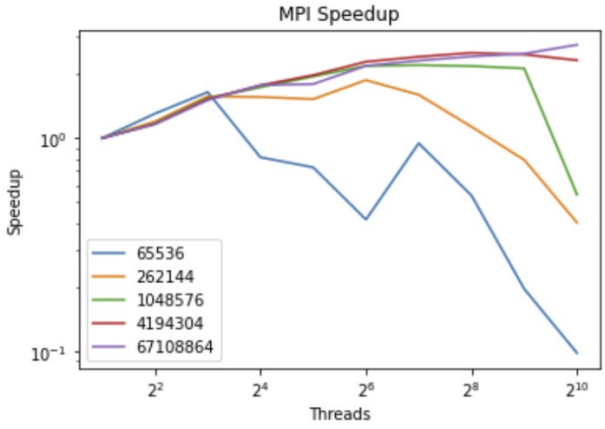


Figure 6: Main



1% Perturbed MPI:

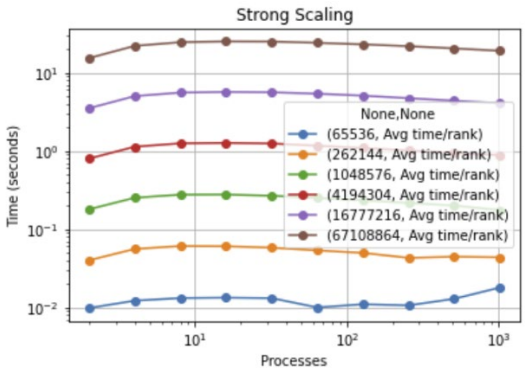


Figure 7: Comp

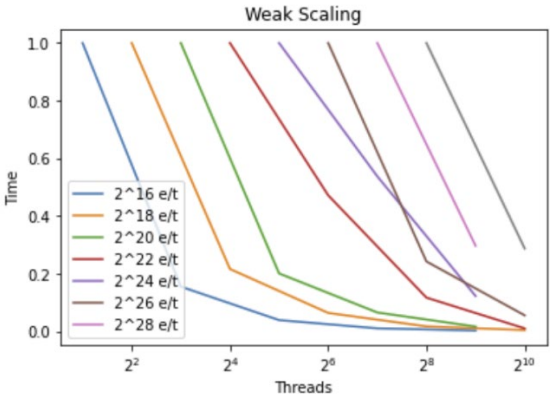
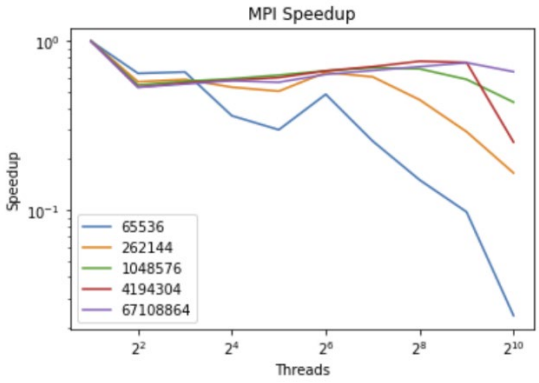


Figure 8: Main



Random CUDA:

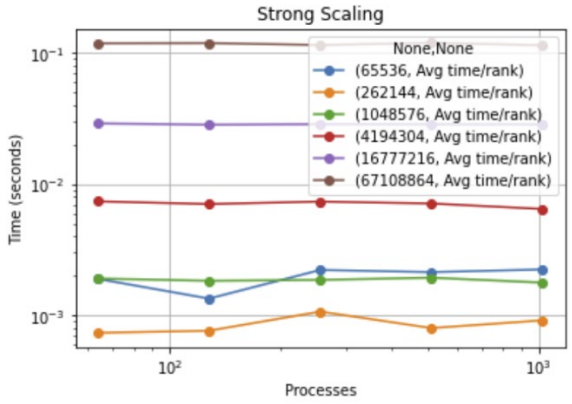


Figure 9: Comm

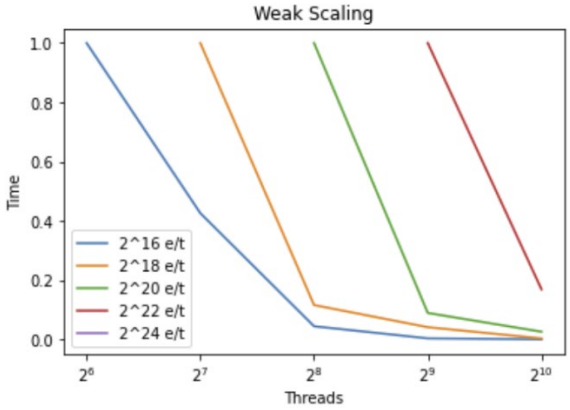
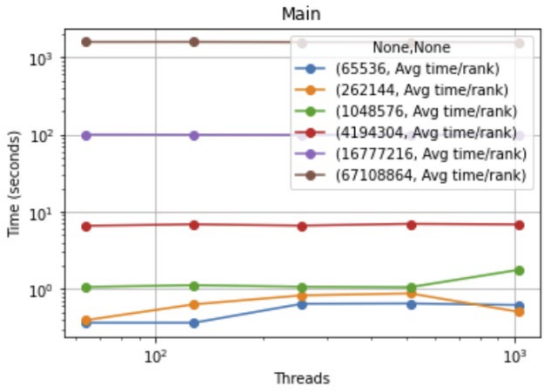


Figure 10: Comp



Sorted CUDA:

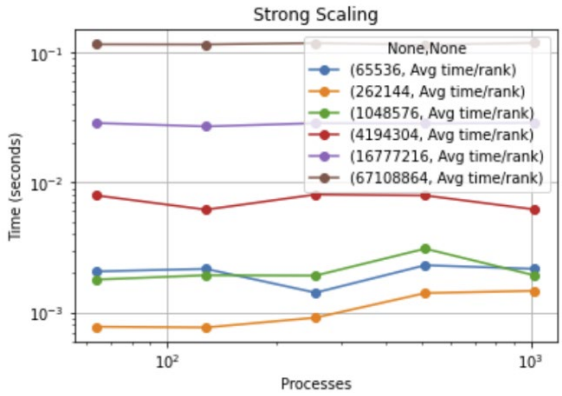


Figure 11: Comm

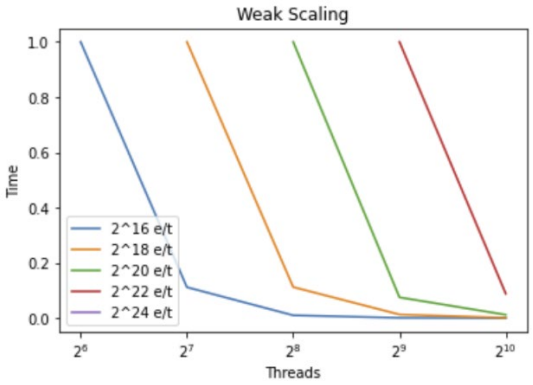
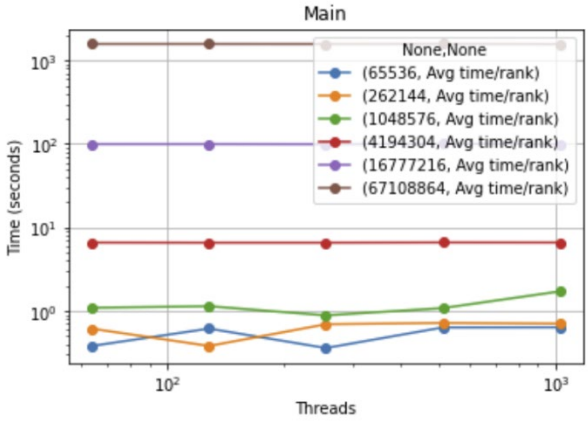


Figure 12: Comp



Reverse Sorted CUDA:

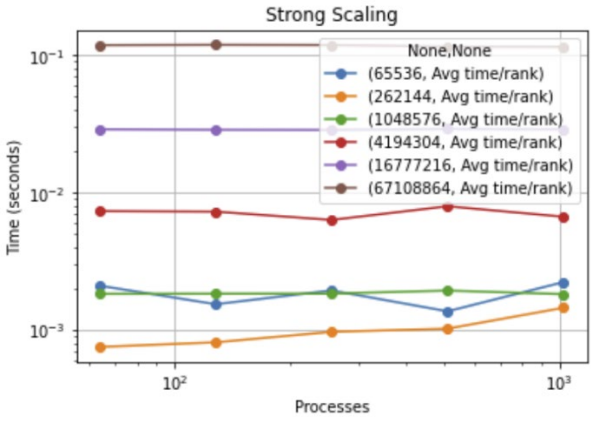


Figure 13: Comm

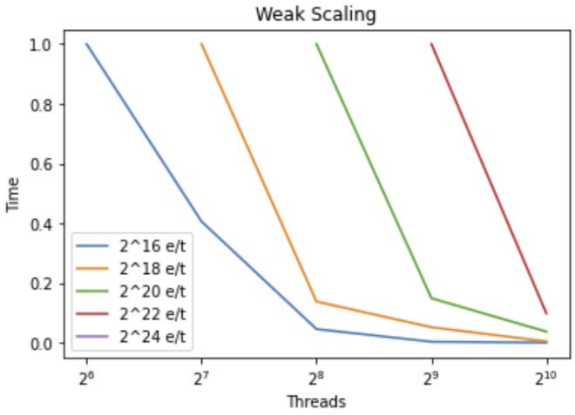
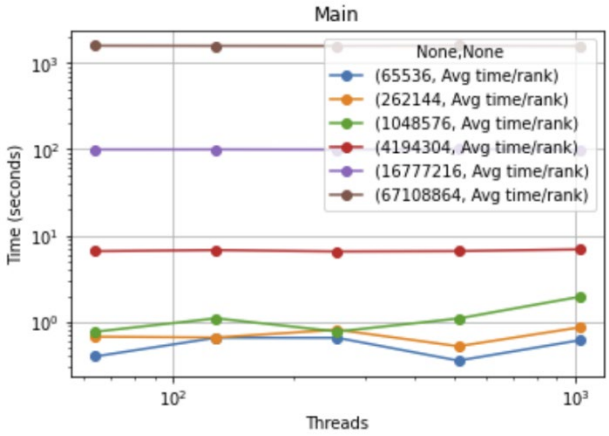


Figure 14: Comp



1% Perturbed CUDA:

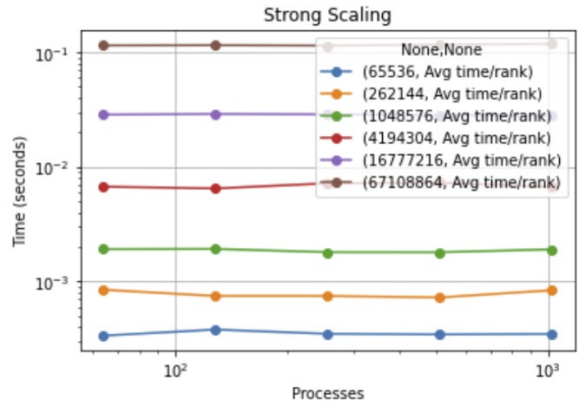


Figure 15: Comm

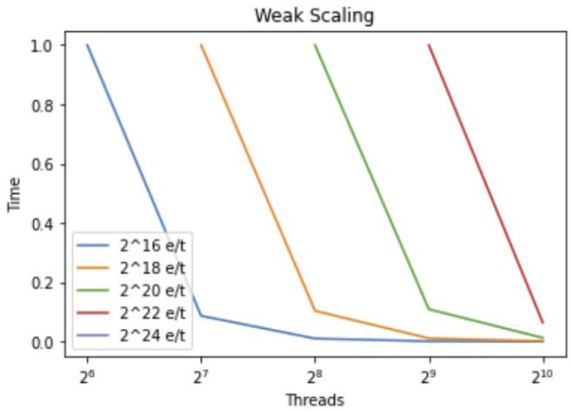
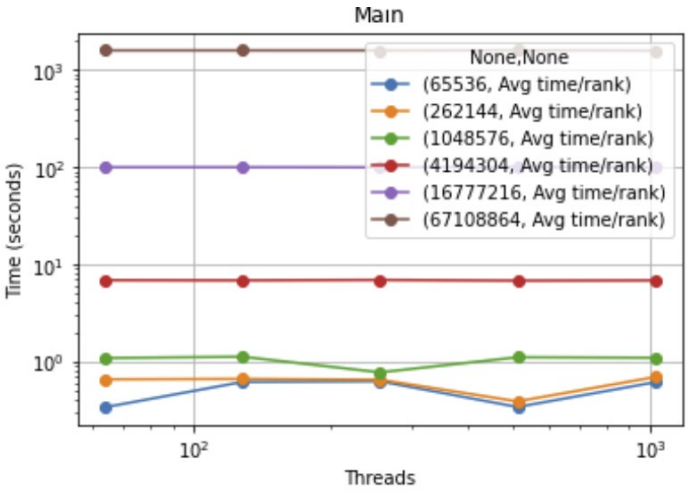


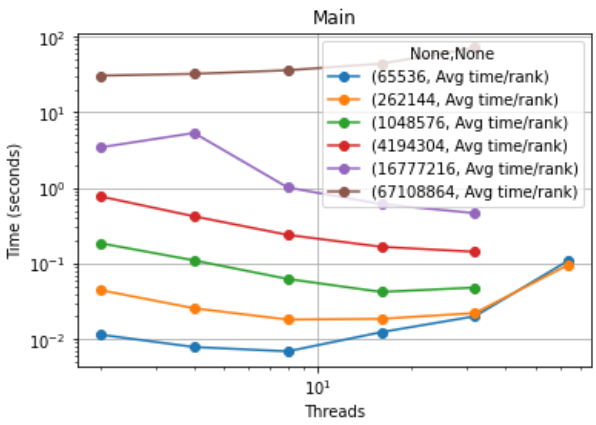
Figure 16: Comp

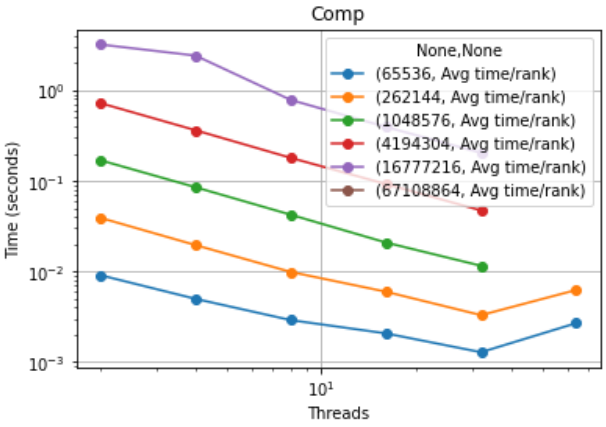

Main
None,None
(65536, Avg time/rank)
(262144, Avg time/rank)
(1048576, Avg time/rank)
(4194304, Avg time/rank)
(16777216, Avg time/rank)
(67108864, Avg time/rank)
Time (seconds)
Threads
10^3
10^2
10^1
10^0
10^2
10^3

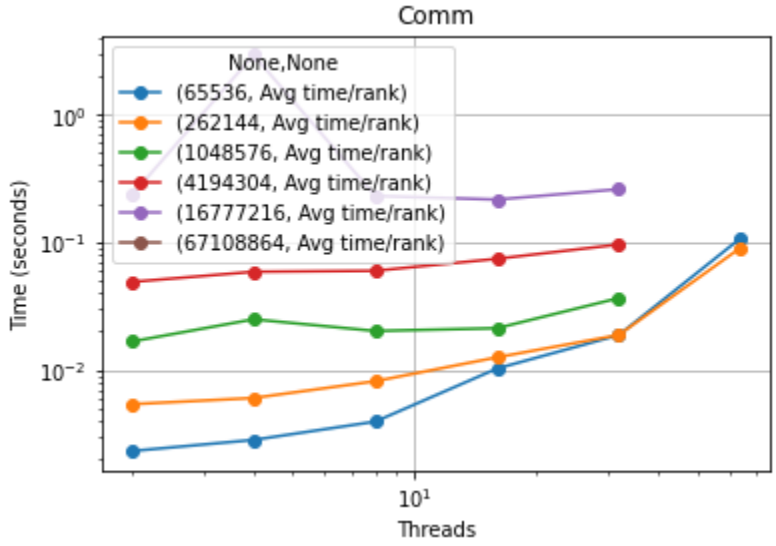
MPI SAMPLE SORT

Random

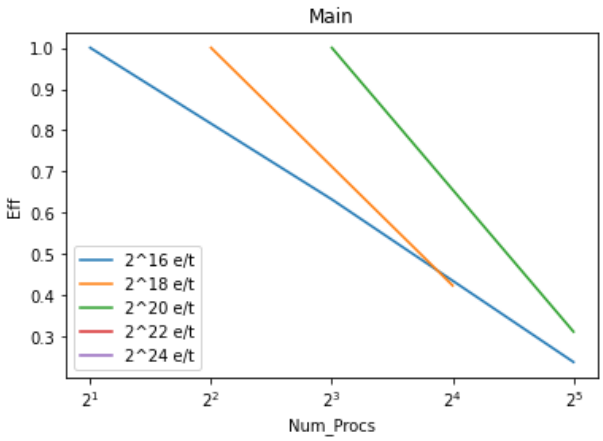
strong scaling:

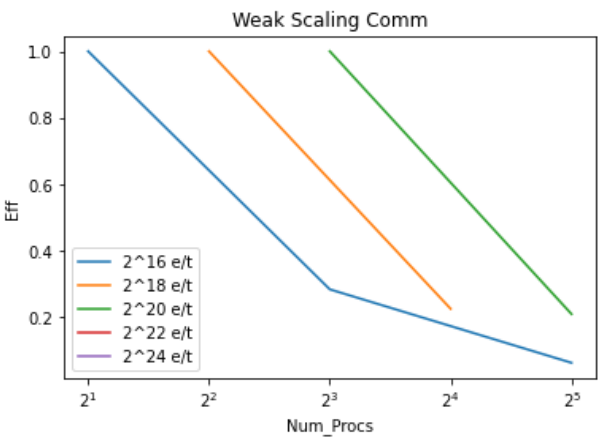


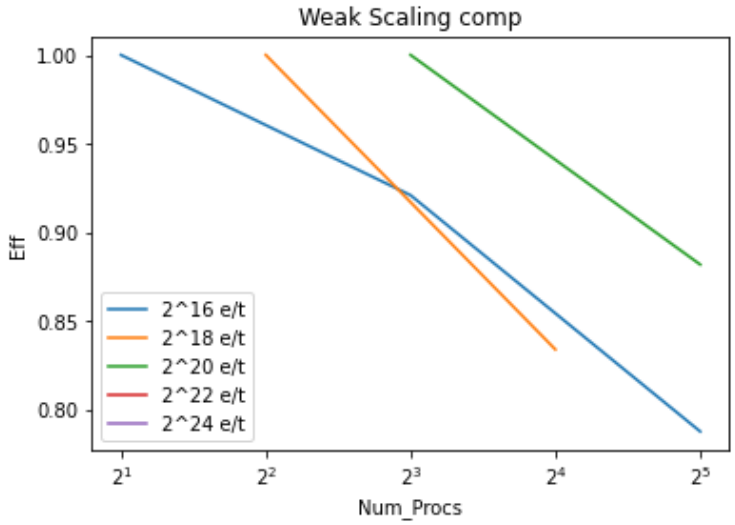




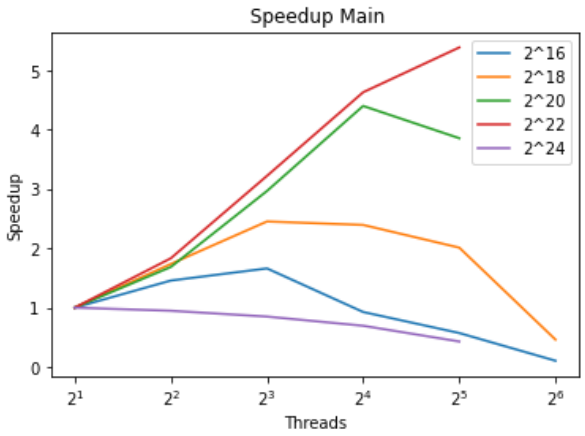
Weak scaling:

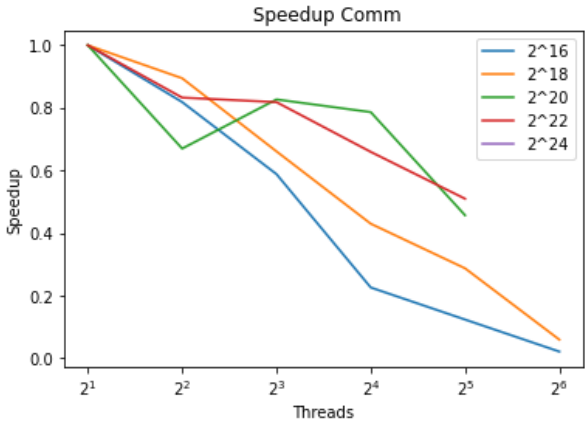

Main
Eff
Num_Procs
2^16 e/t
2^18 e/t
2^20 e/t
2^22 e/t
2^24 e/t

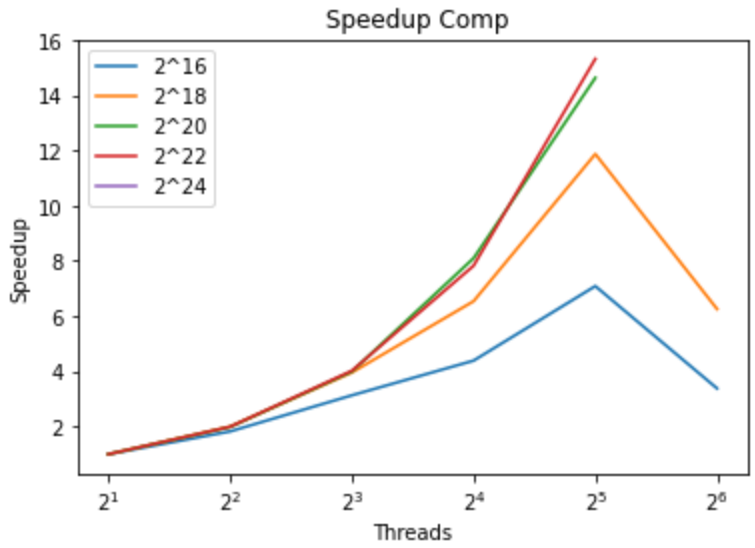

Weak Scaling Comm
Eff
Num_Procs
2^16 e/t
2^18 e/t
2^20 e/t
2^22 e/t
2^24 e/t


Weak Scaling comp
Eff
Num_Procs
2^16 e/t
2^18 e/t
2^20 e/t
2^22 e/t
2^24 e/t

Speedup:

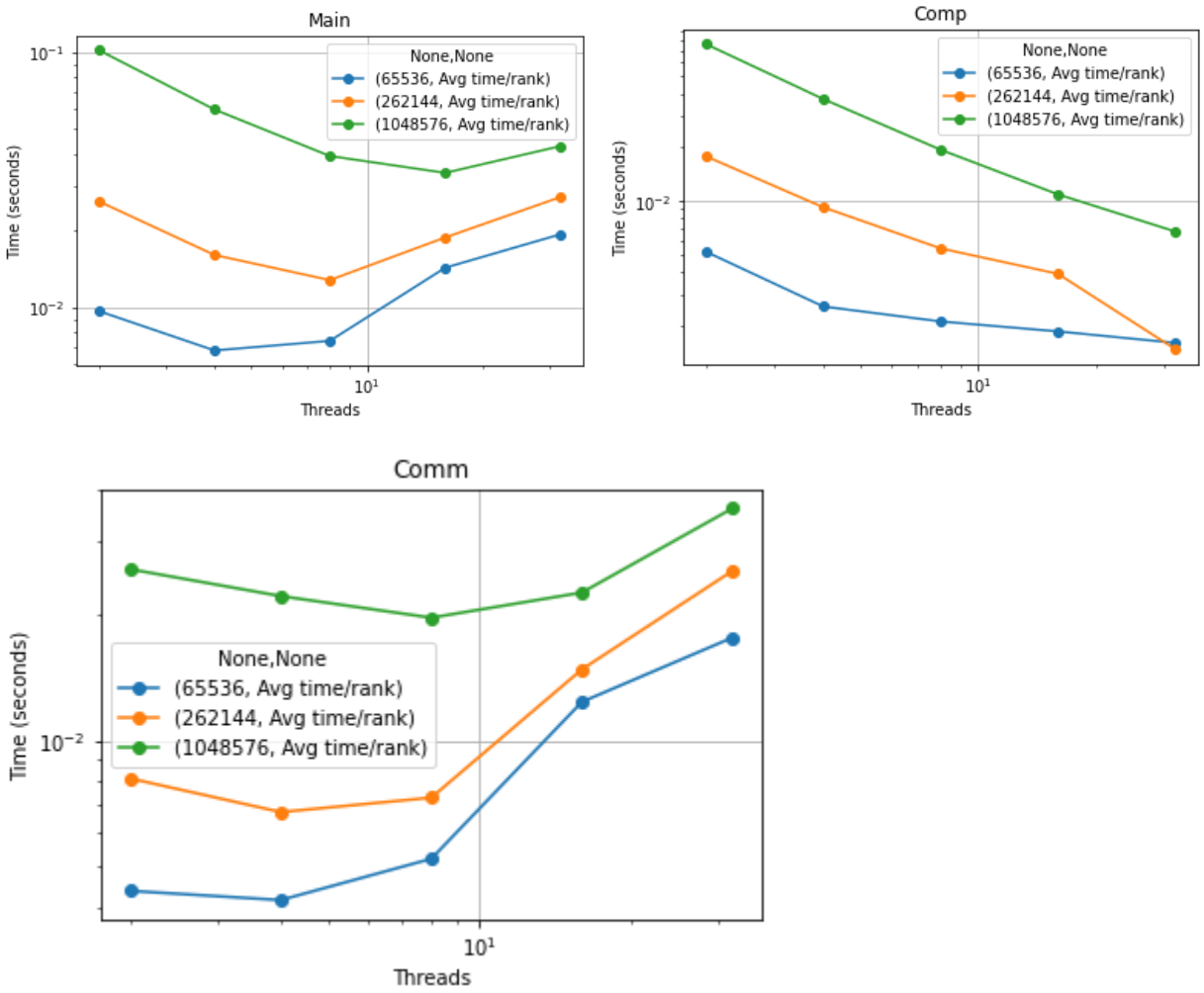

Speedup Main
Speedup
Threads
2^16
2^18
2^20
2^22
2^24


Speedup Comm
Speedup
Threads
2^16
2^18
2^20
2^22
2^24

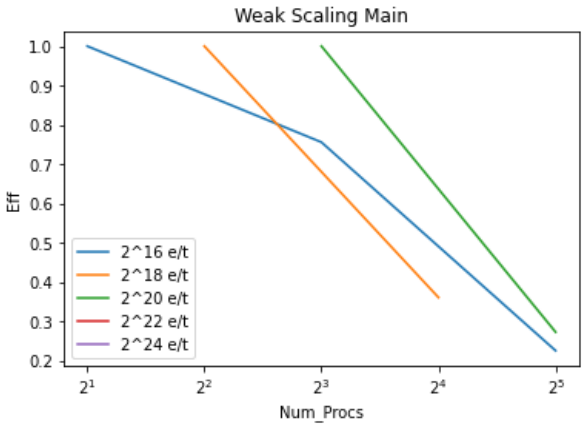


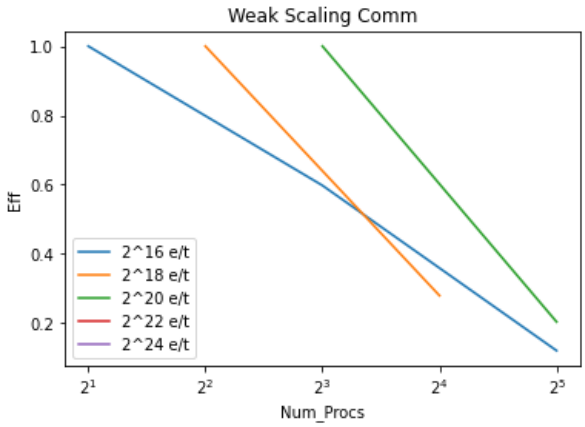
Reverse Sorted

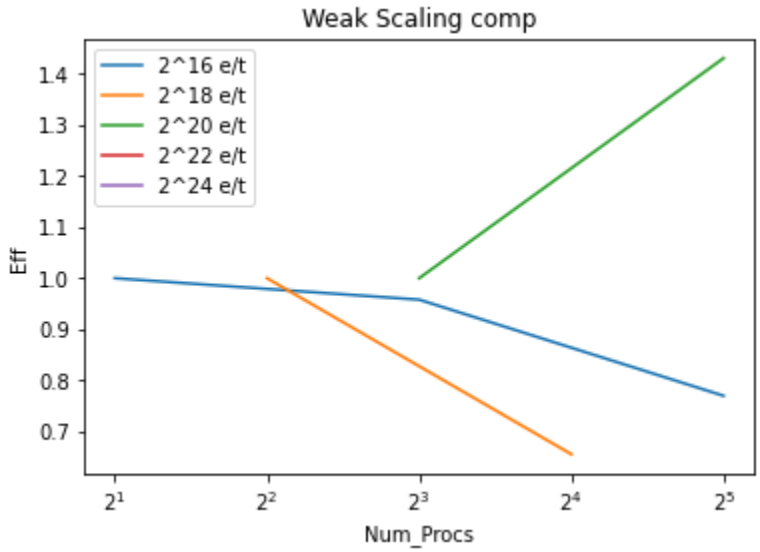
strong scaling:



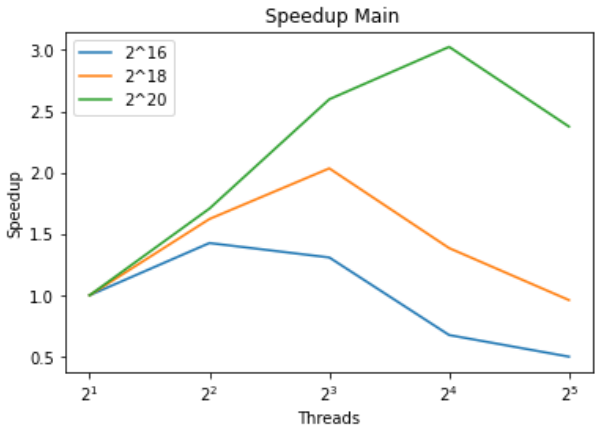
Weak scaling:

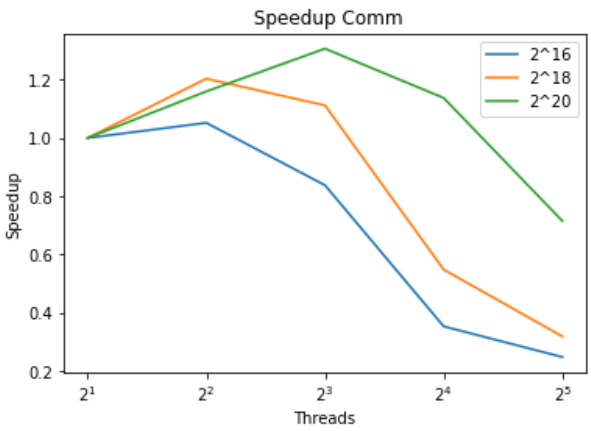

Weak Scaling Main
Eff
Num_Procs
2^16 e/t
2^18 e/t
2^20 e/t
2^22 e/t
2^24 e/t

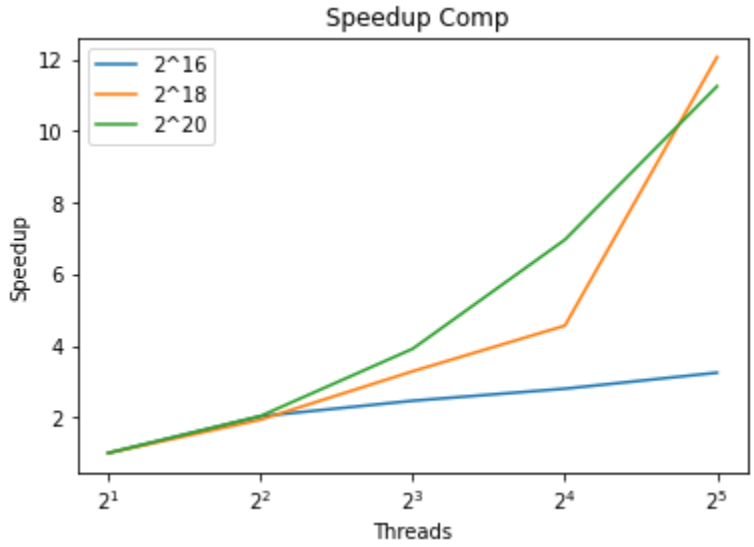

Weak Scaling Comm
Eff
Num_Procs
2^16 e/t
2^18 e/t
2^20 e/t
2^22 e/t
2^24 e/t


Weak Scaling comp
Eff
Num_Procs
2^16 e/t
2^18 e/t
2^20 e/t
2^22 e/t
2^24 e/t

Speedup:

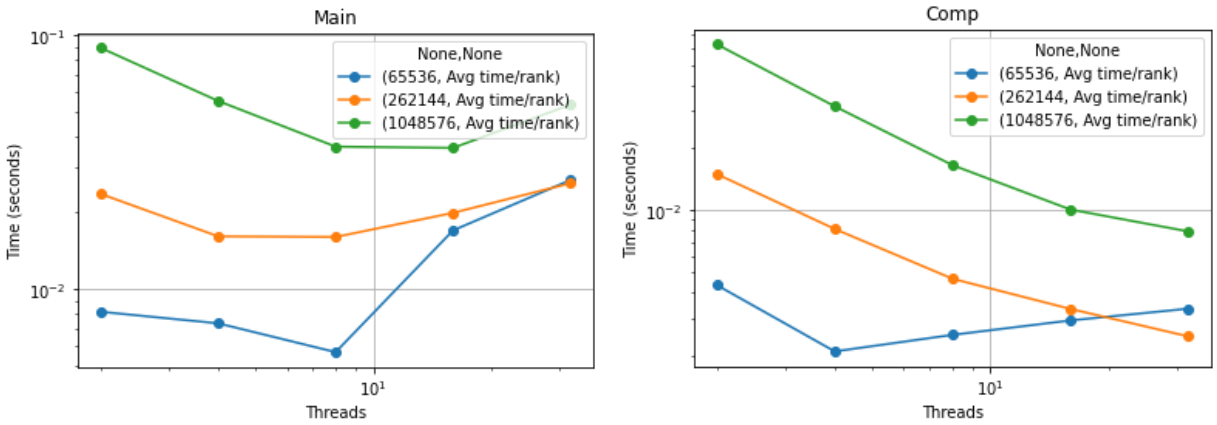


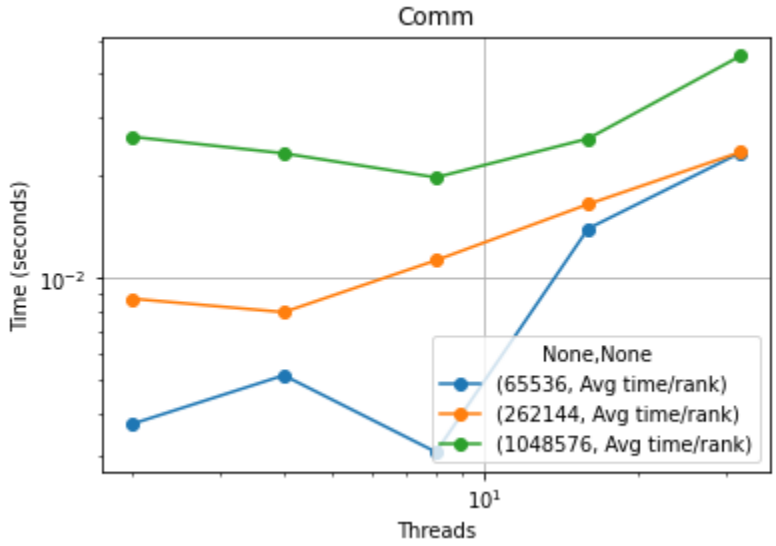




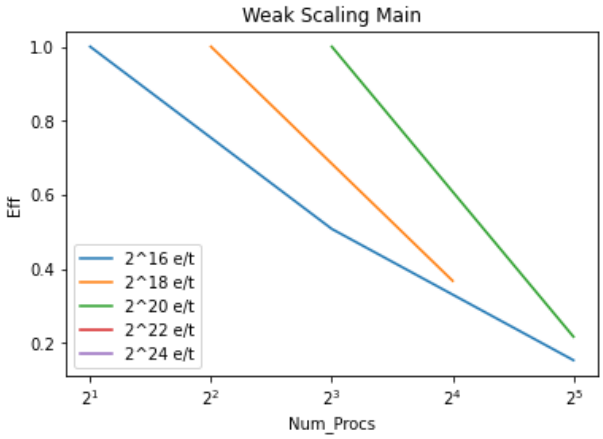
Sorted

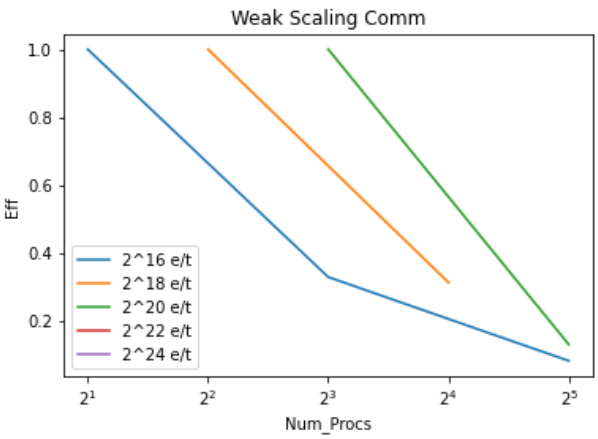
strong scaling:

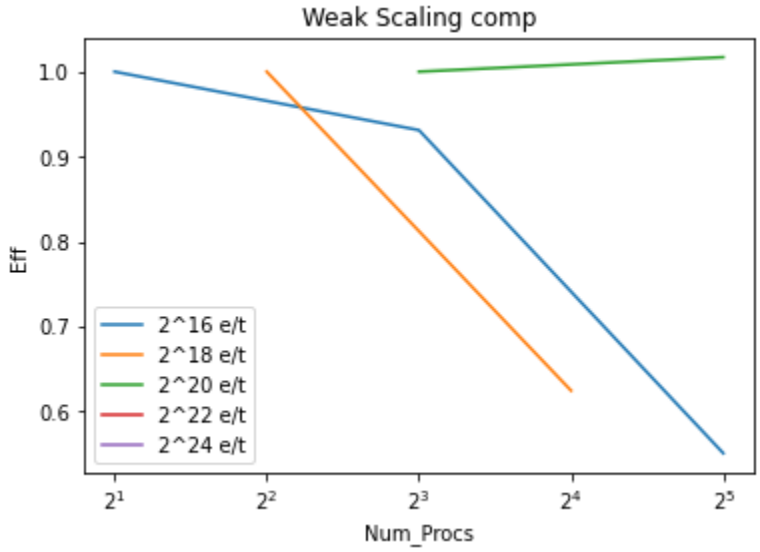




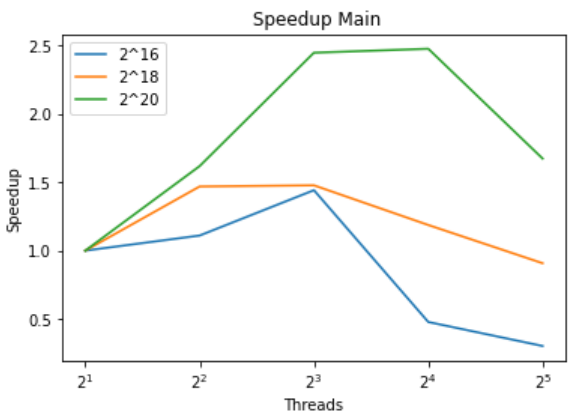
Weak scaling:

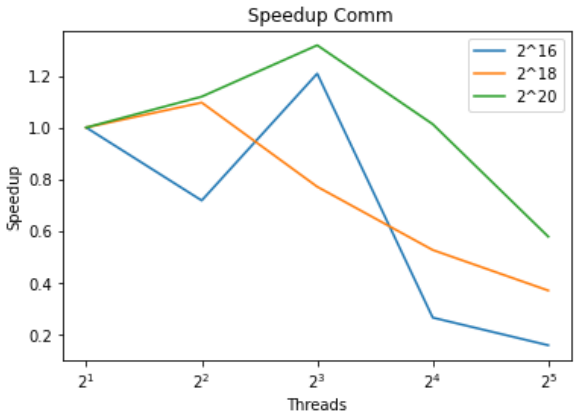

Weak Scaling Main
Eff
Num_Procs
2^16 e/t
2^18 e/t
2^20 e/t
2^22 e/t
2^24 e/t

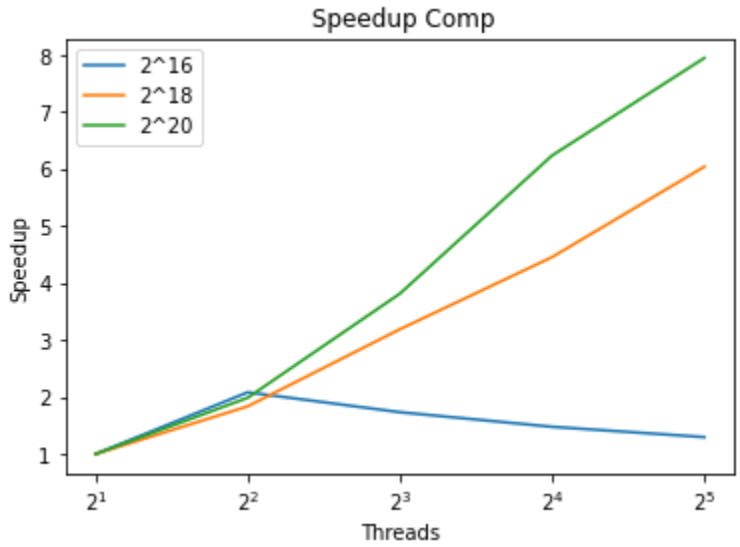

Weak Scaling Comm
Eff
Num_Procs
2^16 e/t
2^18 e/t
2^20 e/t
2^22 e/t
2^24 e/t


Weak Scaling comp
Eff
Num_Procs
2^16 e/t
2^18 e/t
2^20 e/t
2^22 e/t
2^24 e/t

Speedup:

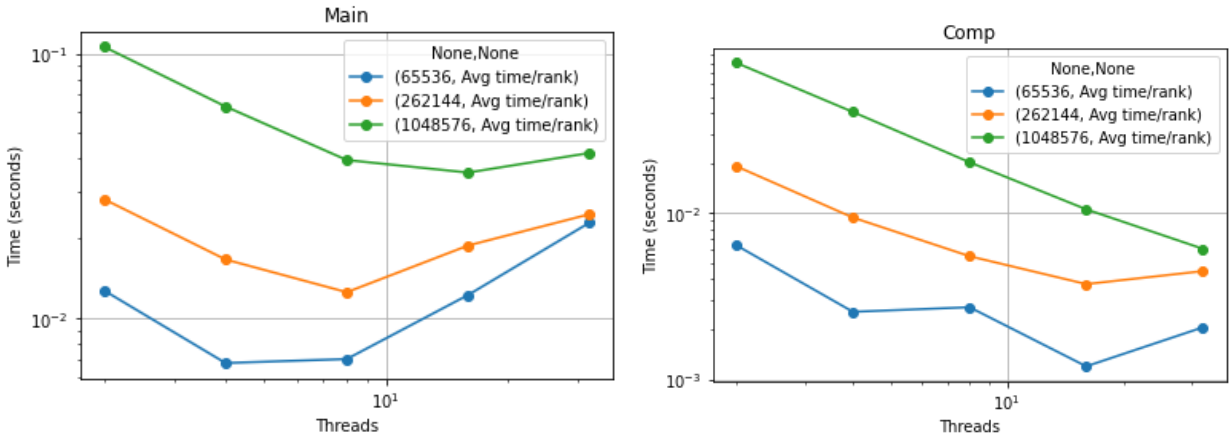


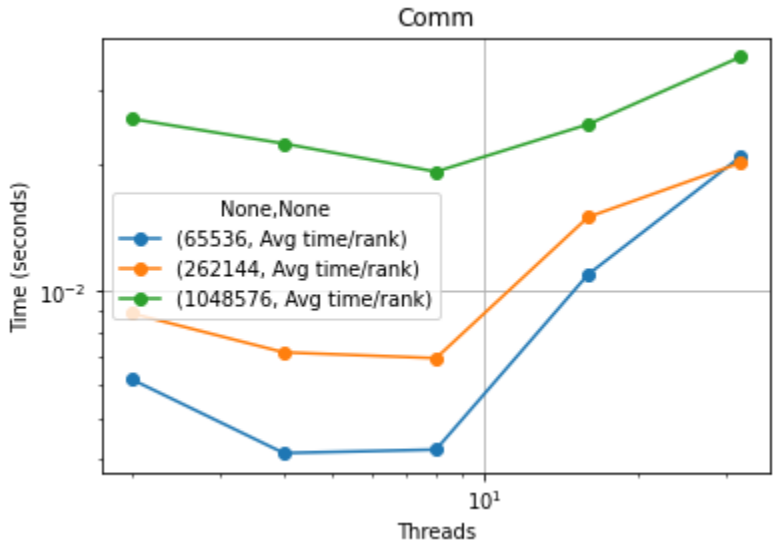




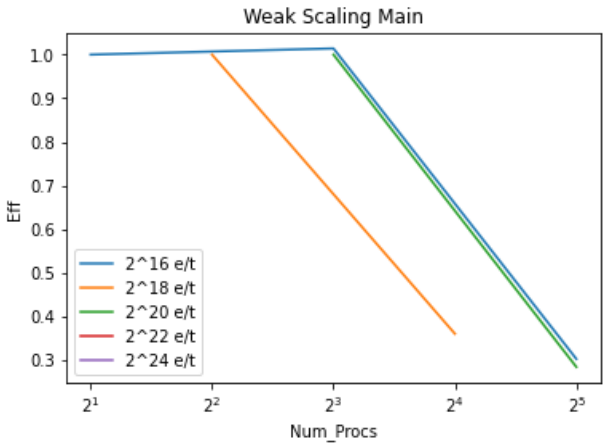
1% Pertubed

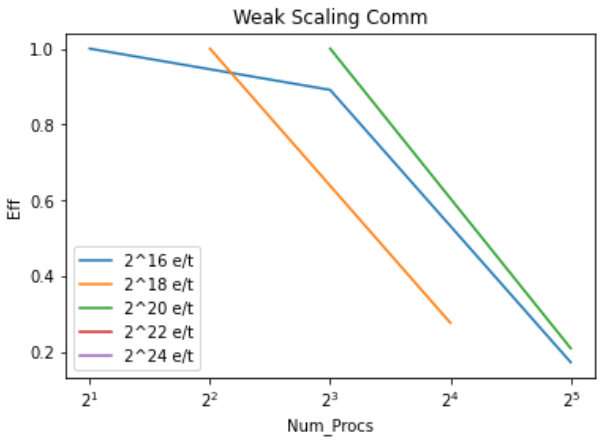
strong scaling:

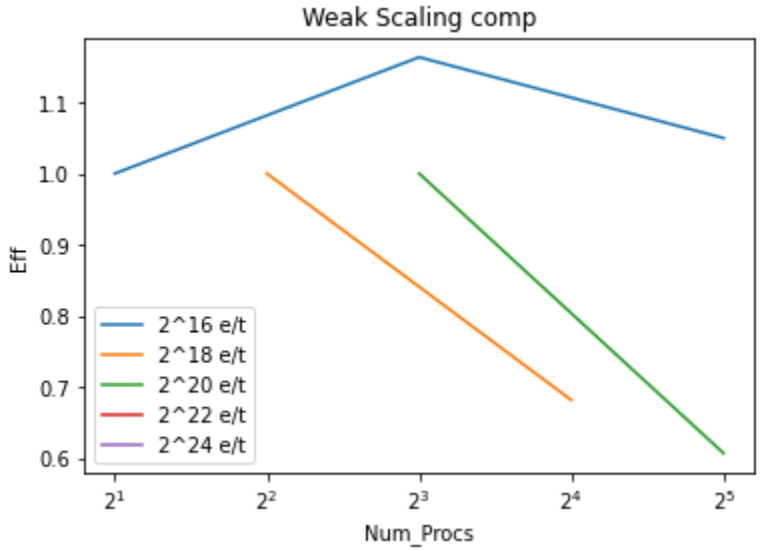




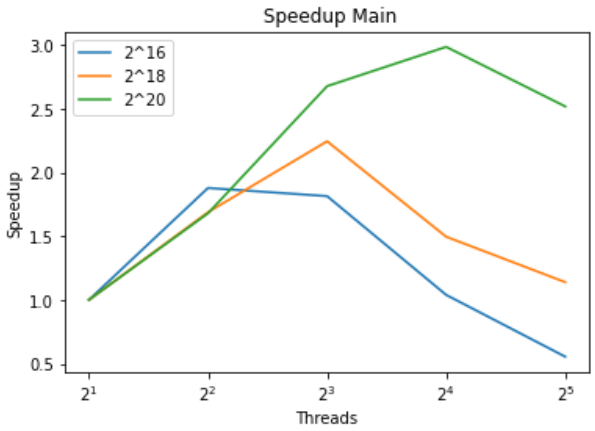
Weak scaling:

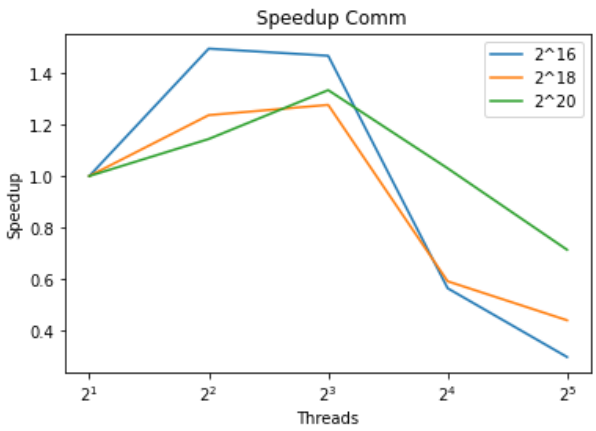

Weak Scaling Main
Eff
Num_Procs
2^16 e/t
2^18 e/t
2^20 e/t
2^22 e/t
2^24 e/t

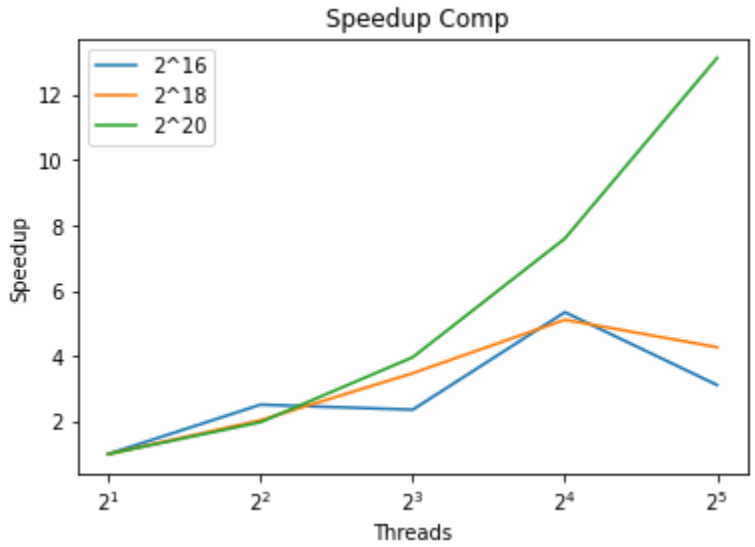

Weak Scaling Comm
Eff
Num_Procs
2^16 e/t
2^18 e/t
2^20 e/t
2^22 e/t
2^24 e/t


Weak Scaling comp
Eff
Num_Procs
2^16 e/t
2^18 e/t
2^20 e/t
2^22 e/t
2^24 e/t

Speedup:







MPI Sample Sort Analysis:

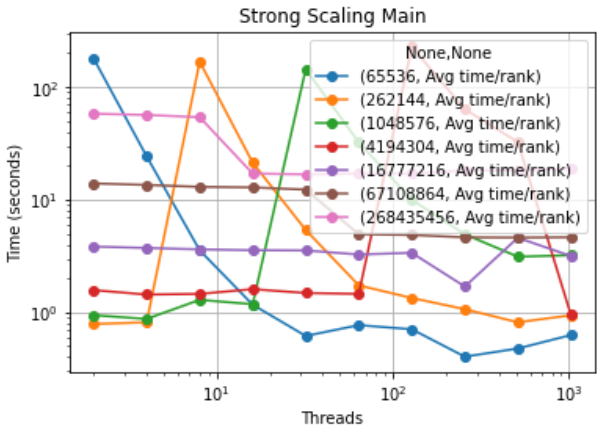
From all of these different graphs, one inference that can be made from the graph correlations is that additional number of processors decreases the computational time, but also increases the communication time. This makes sense, because the way parallel sample sort works, there is a lot of communication between processors to share splitters, and the processors also must communicate which bucket they will be sorting in which order. Likely, since there is also a lot of synchronization in the mpi sample sort, processors may end up having to wait for each other to finish. The decrease in computational time also makes sense, since the work is divided between processors. Each process takes a partition of the array to be sorted, locally sorts and eventually create global splitters. The processors then again take a specific bucket to be sorted and locally sorts their respective buckets, and then combines the buckets to finish the sort. More processors means more elements split into buckets for local sorting, which is why the computational time decreases as we increase processors.

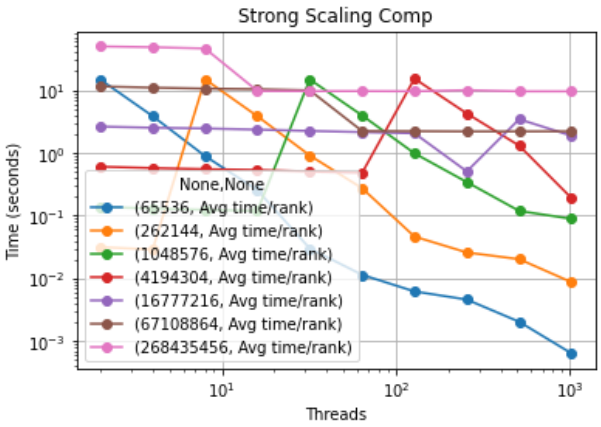
Another inference is that larger data sizes demonstrate a better correlation for the speedup of sample sort. This is likely due to the fact that for small input sizes, the work it takes to sort the array is so minimal that it isn't worth to add more processors since it would also increase the communication times between them. Furthermore, there is no significant difference between the times it take to sample sort for the different input types of sorted, random, reverse sorted, and one percent perturbed. This is likely due to the nature of sample sort, since it utilizes quick sort for local process sorting, the input type does not influence quick sort speed.

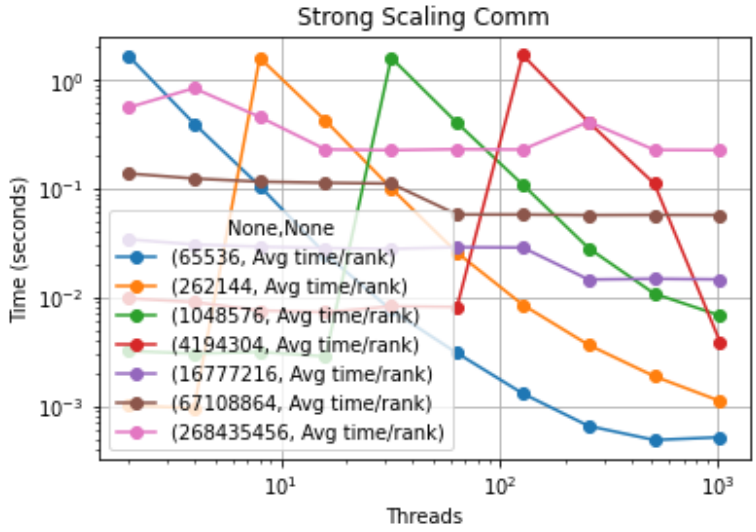
CUDA SAMPLE SORT

Random

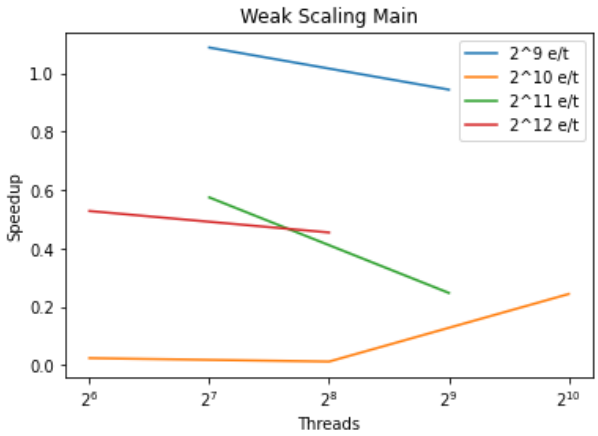
Strong Scaling:

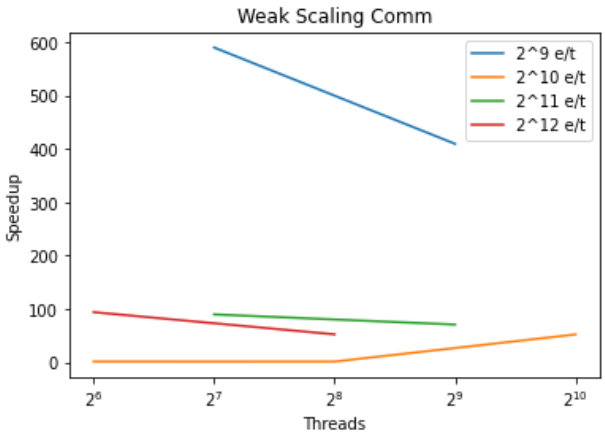


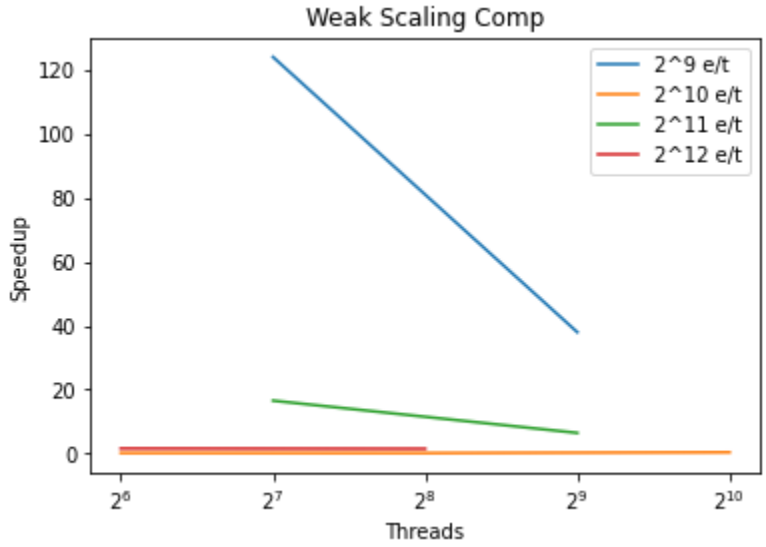




Weak Scaling:

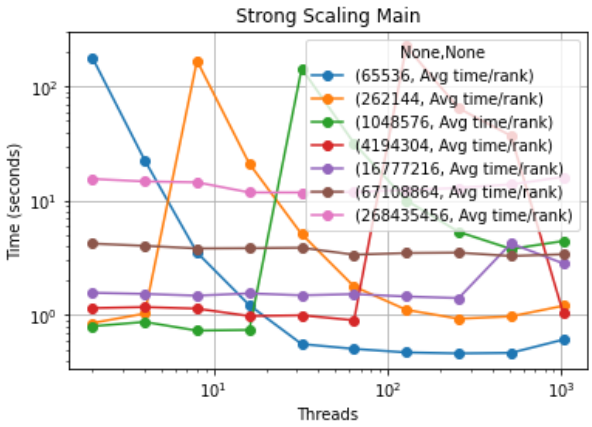


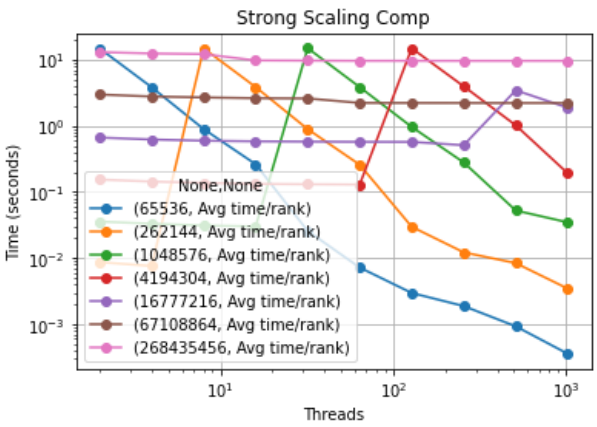


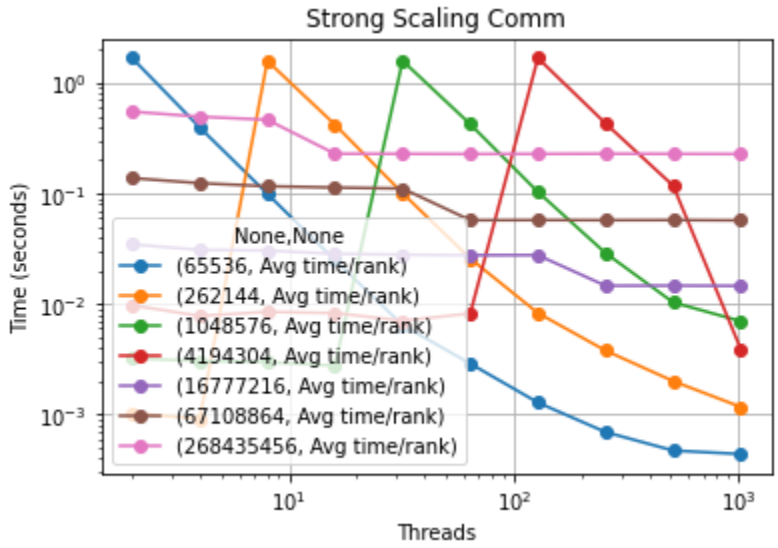


Reverse Sorted

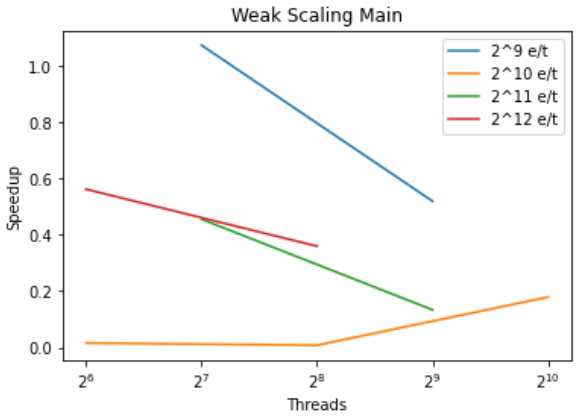
Strong Scaling:

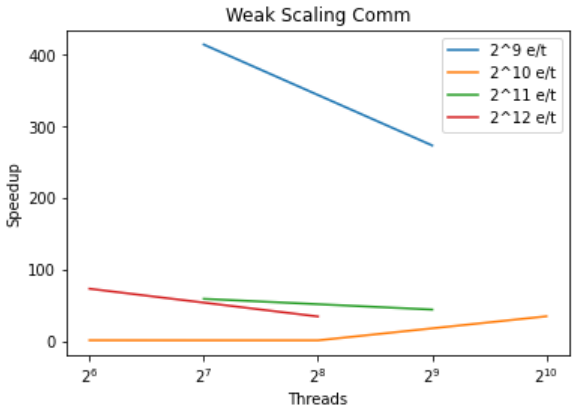


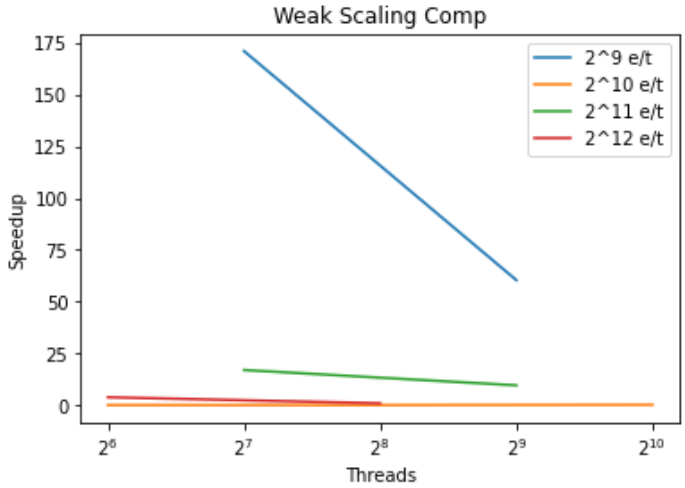




Weak scaling:

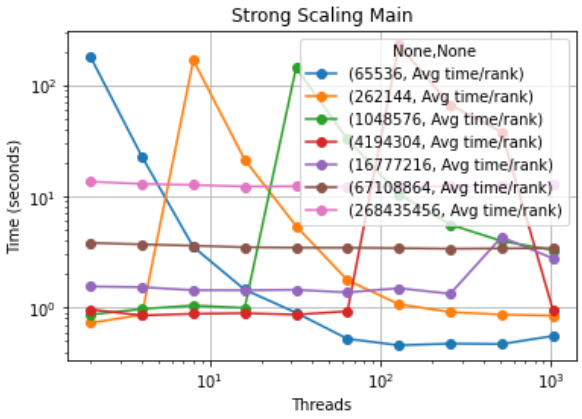


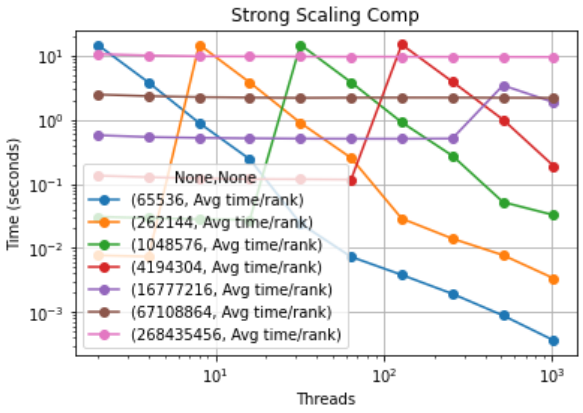


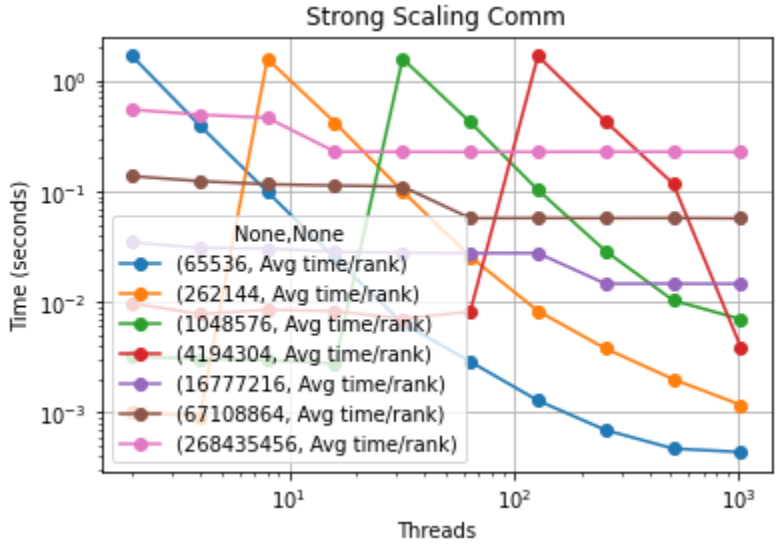


Sorted

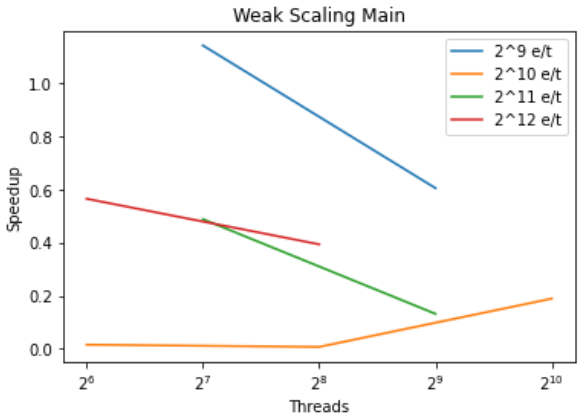
Strong Scaling:

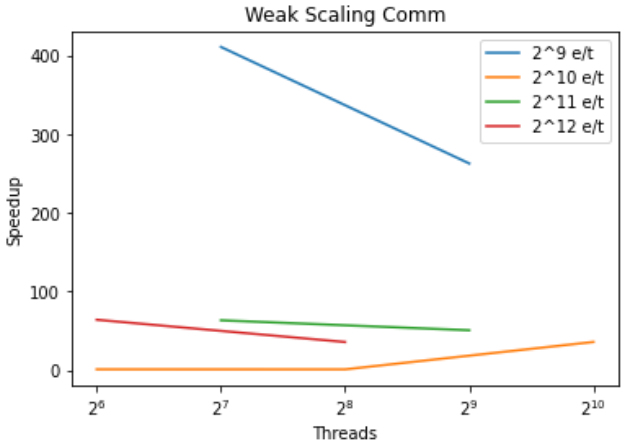


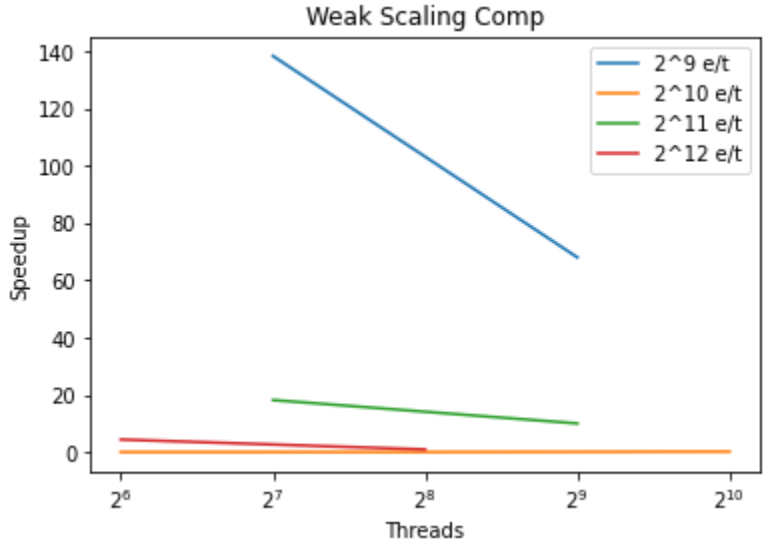




Weak Scaling:

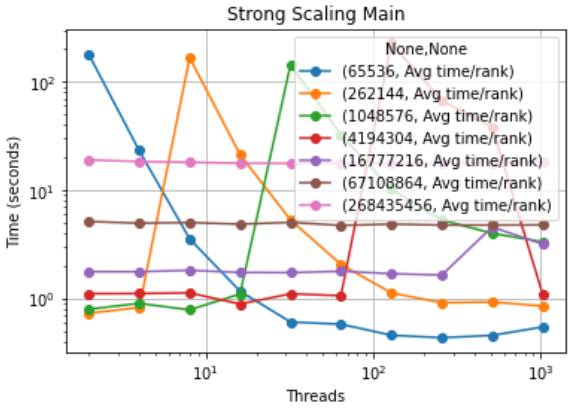


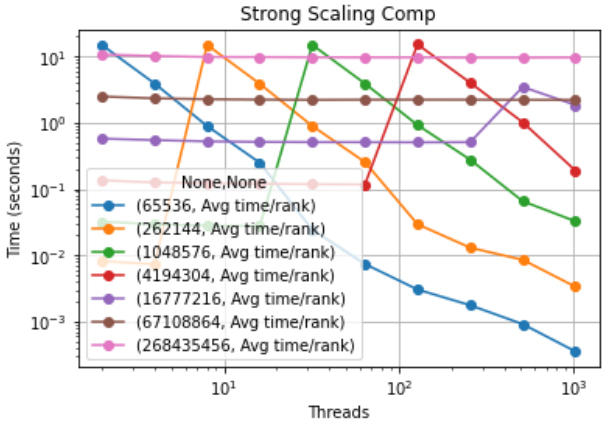


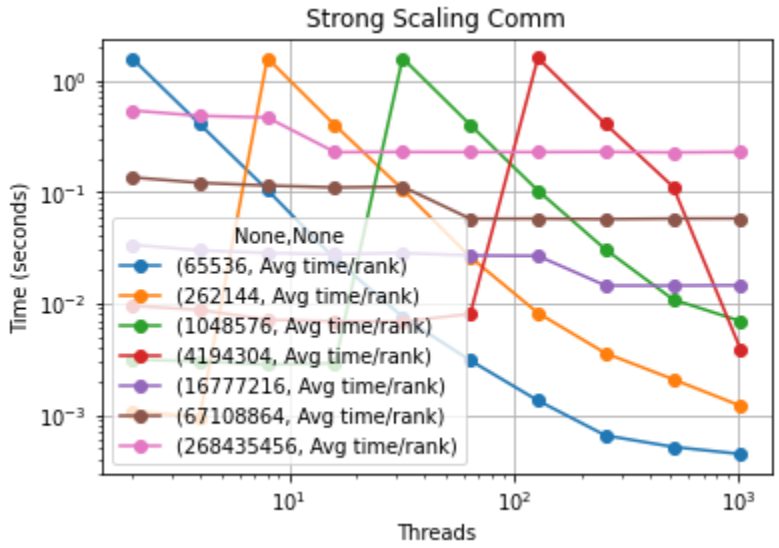


1% Pertubed

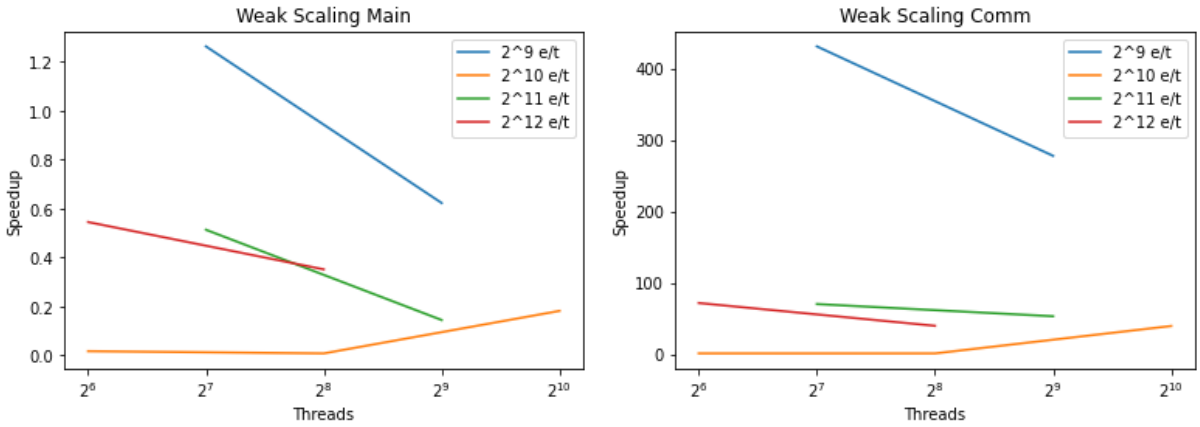
Strong Scaling:

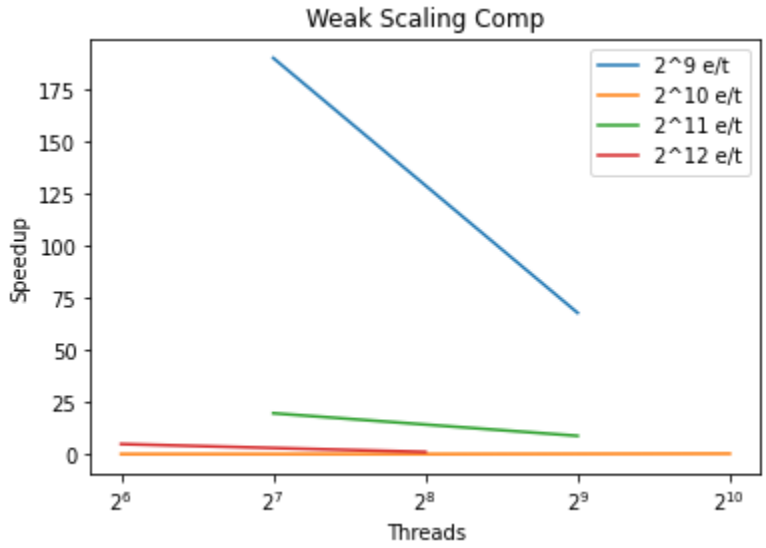






Weak Scaling:





CUDA Sample Sort Analysis:

From the different graphs, there are no noticeable differences in sorting performance with different input types. Furthermore, the strong scaling graphs demonstrate that there is good parallelization for smaller input sizes, but struggles with larger input sizes. I don't really know why this is the case, but it looks like to me that maybe there weren't enough resources to allocate between threads for the larger input sizes. The weak scaling shows there is good parallelization for only the smallest input size. I think I should have tested even more smaller input sizes to make sure that this isn't just an outlier but that my code actually parallizes for smaller input sizes.