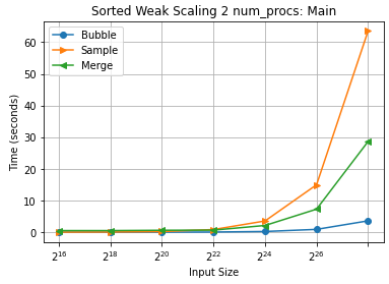
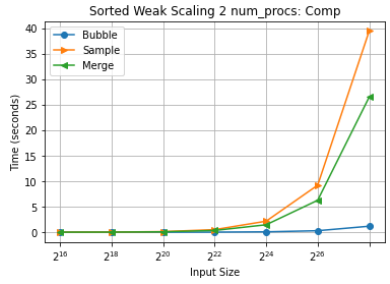
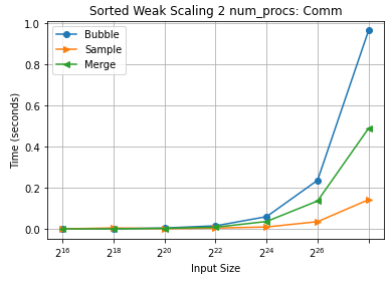
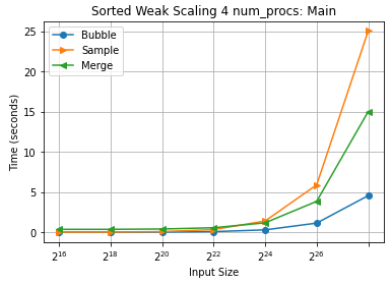
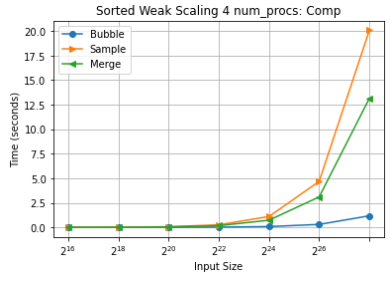
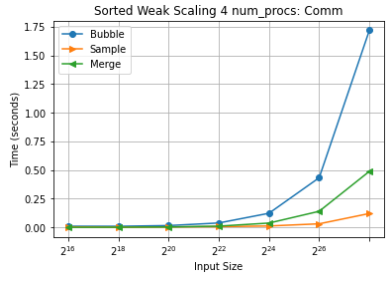
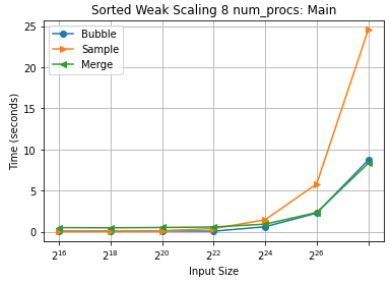
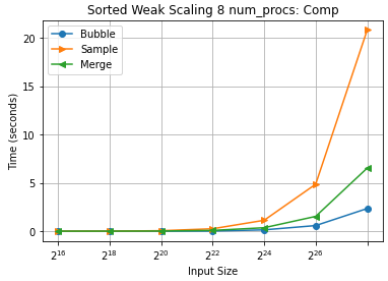
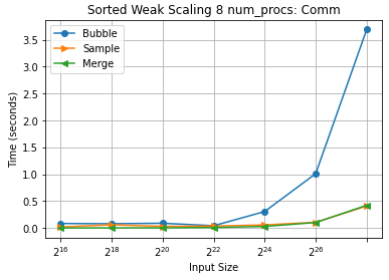
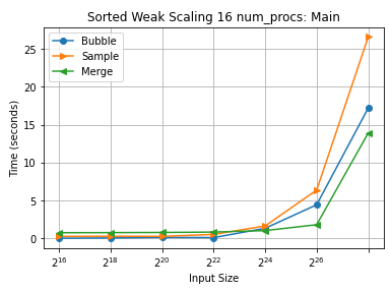
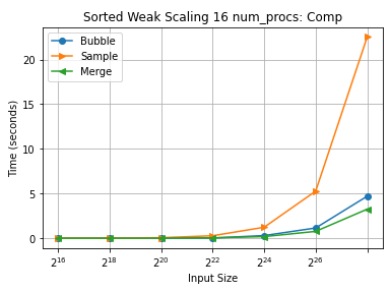
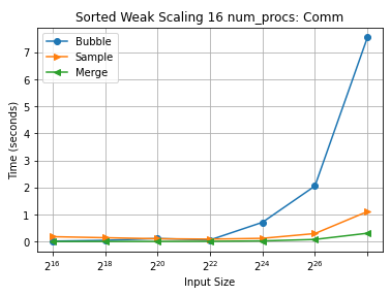
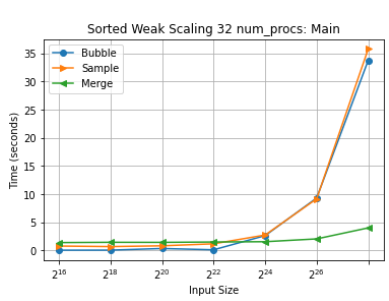
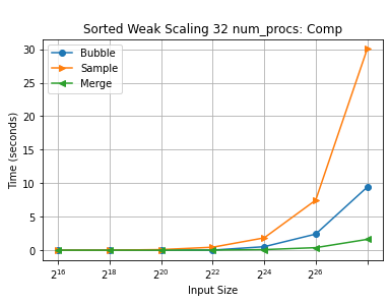
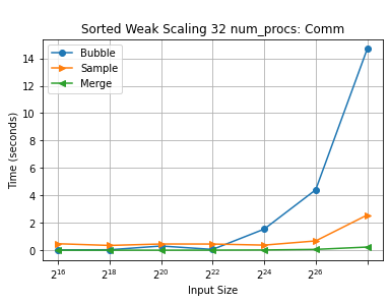
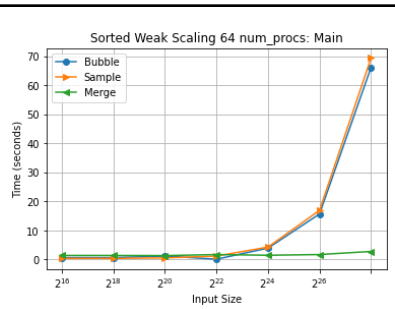
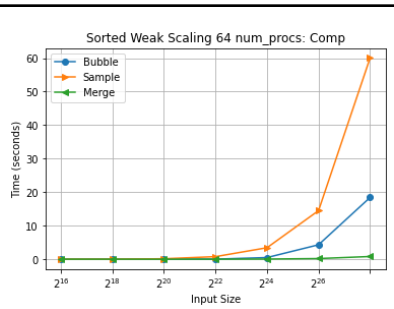
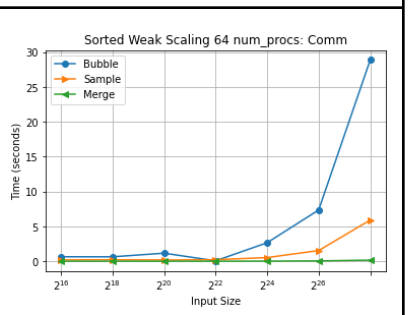
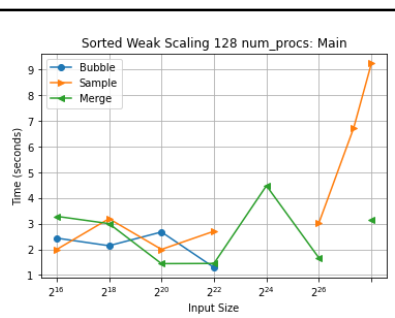
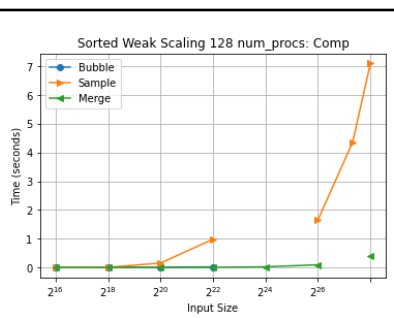
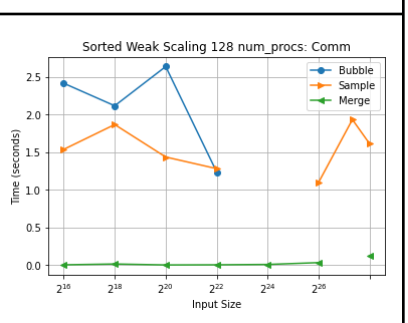
MPI Comparison Graphs

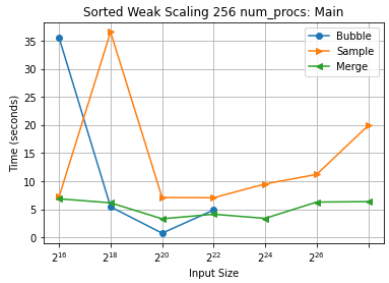
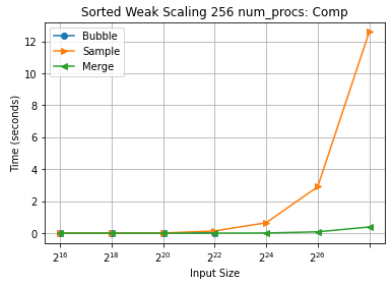
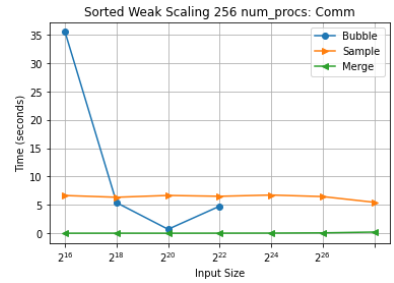
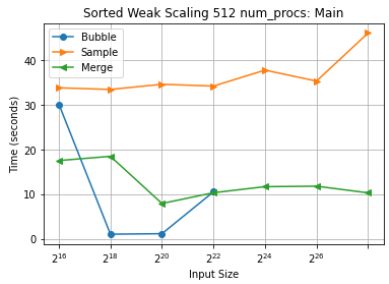
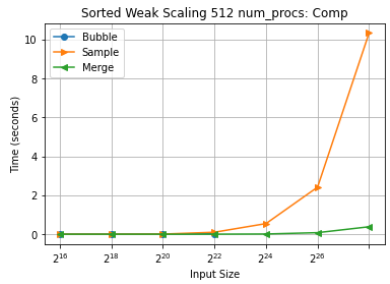
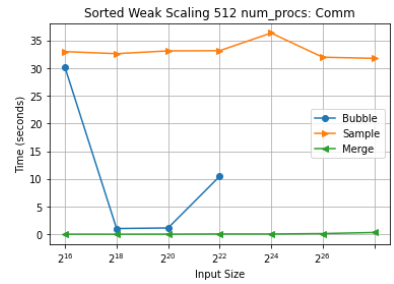
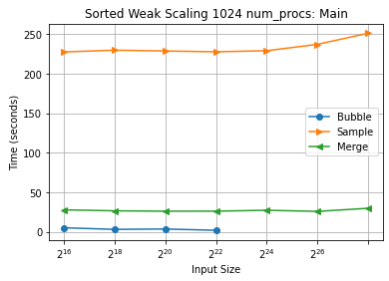
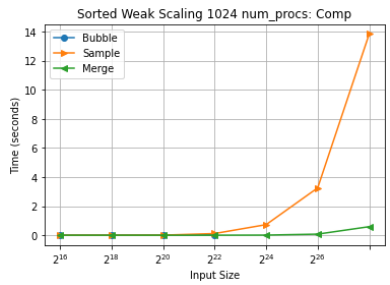
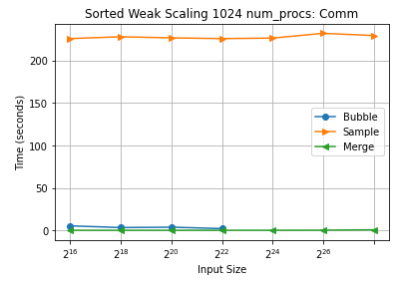
Weak Scaling

These weak scaling graphs demonstrate the performance of our different algorithms relative to each other. As you can see bubble sort is by far the worst except when it comes to sorted data where the algorithm has a nice early exit condition. For all the other sort types, merge sort performs the best while sample sort is right there with it, up until there are too many processors and the communication costs outweigh the added computation speedup.

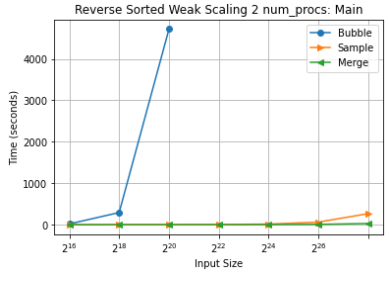
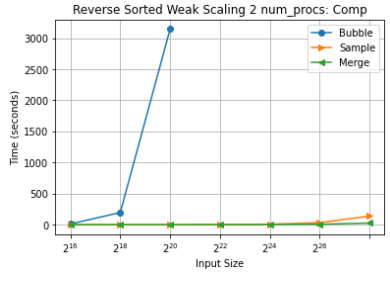
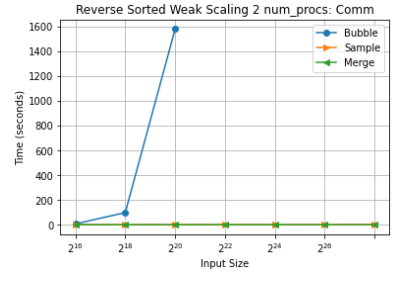
Sorted

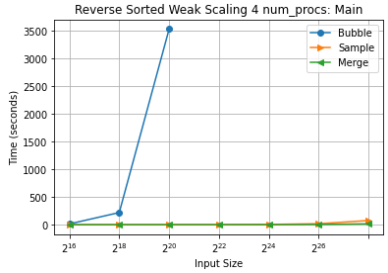
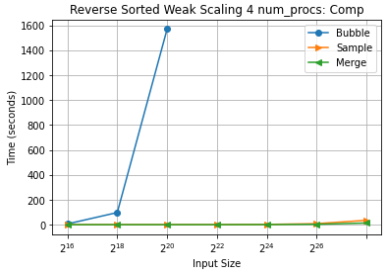
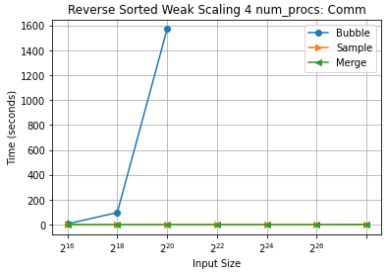
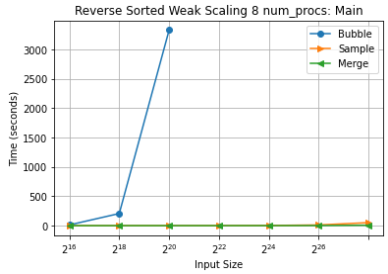
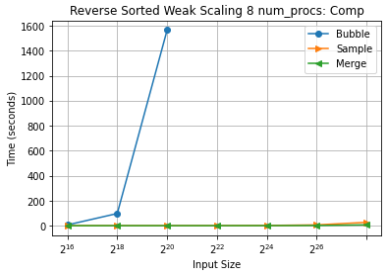
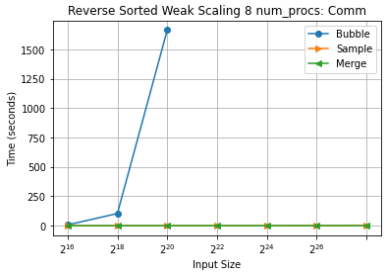
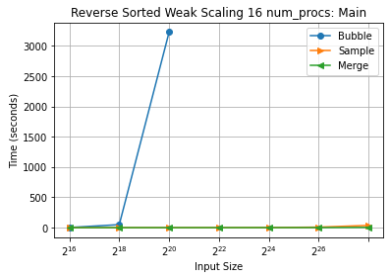
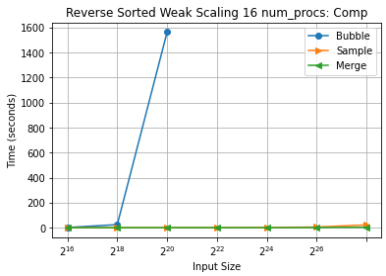
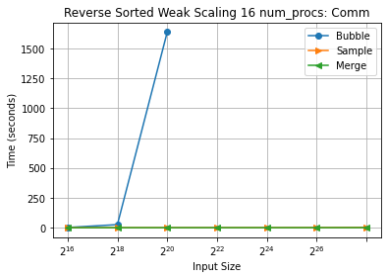
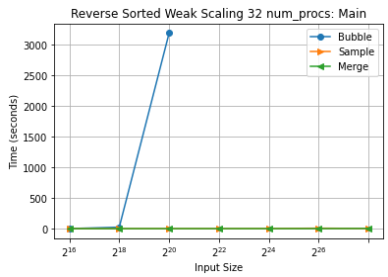
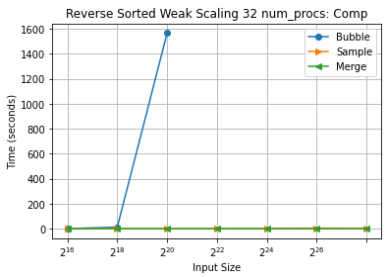
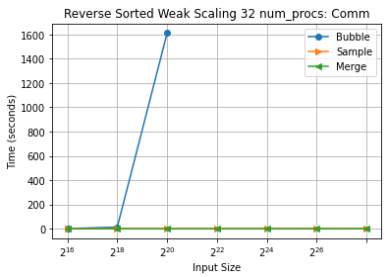
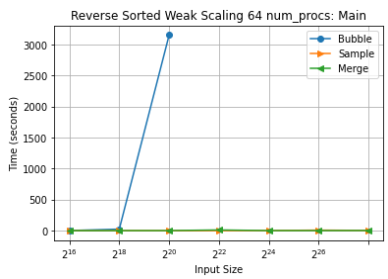
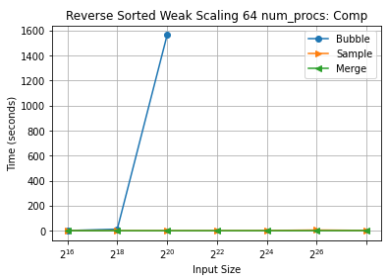
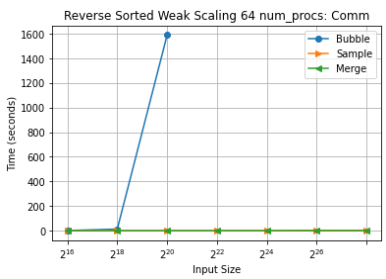
Main	Computation	Communication
		
		

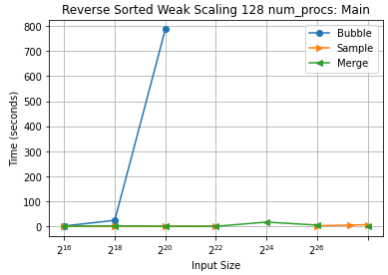
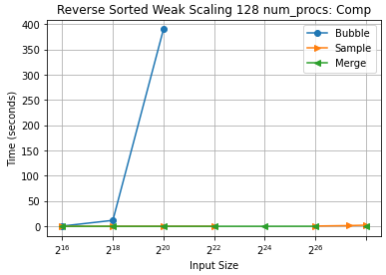
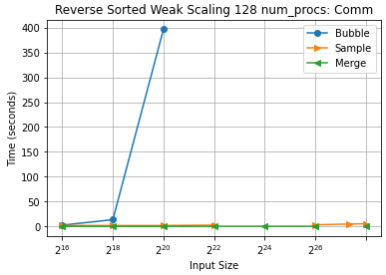
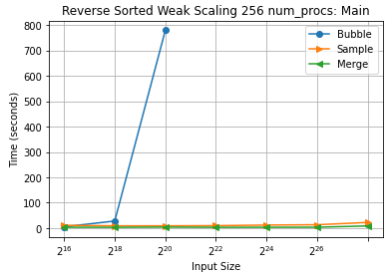
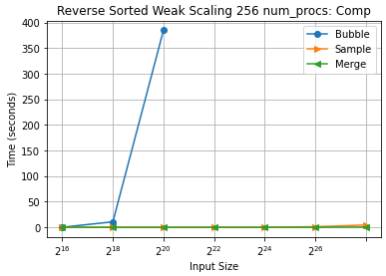
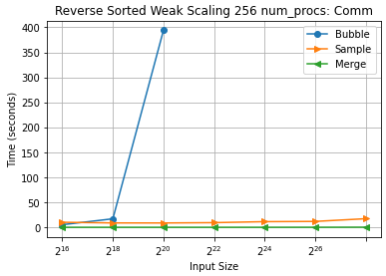
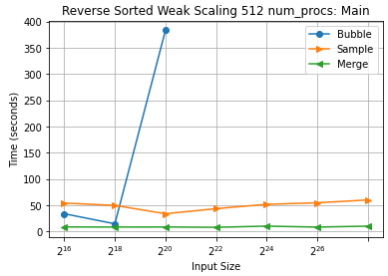
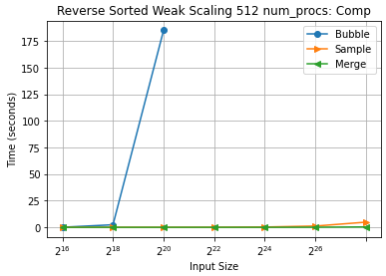
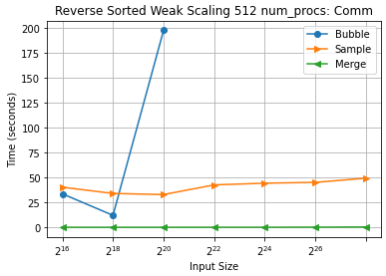
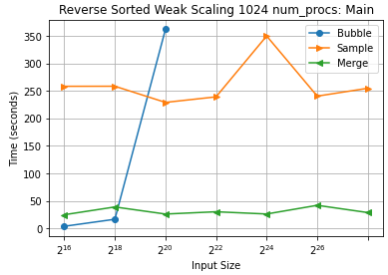
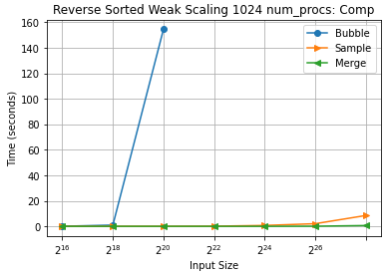
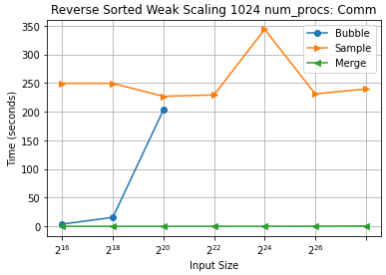
Main	Computation	Communication
		
		
		
		
		

Main	Computation	Communication
		
		
		

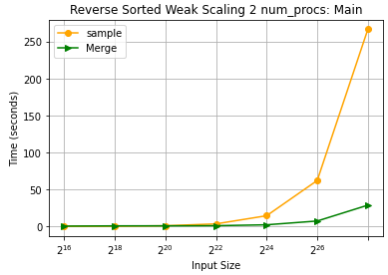
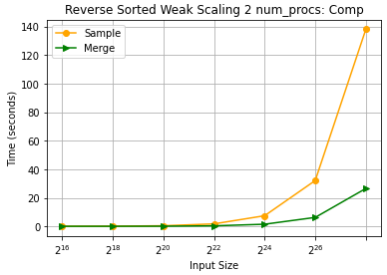
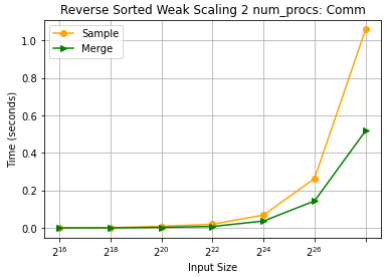
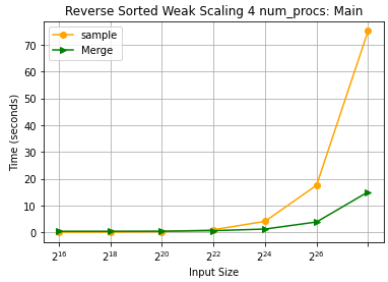
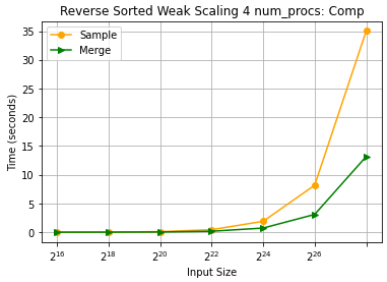
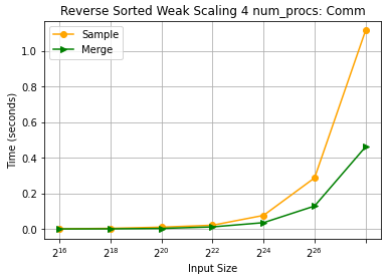
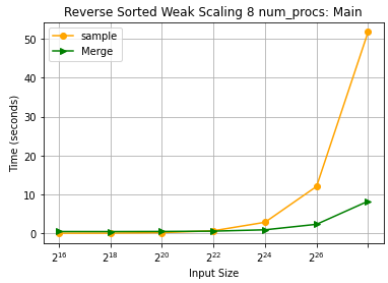
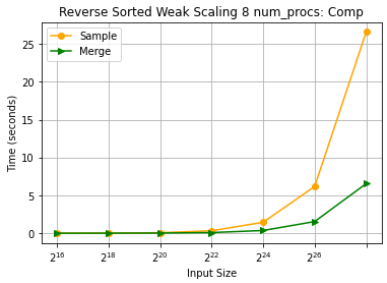
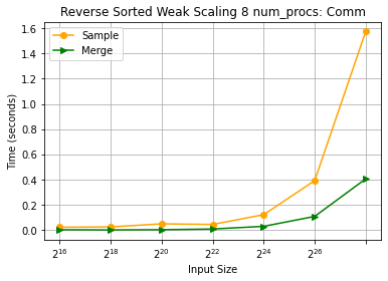
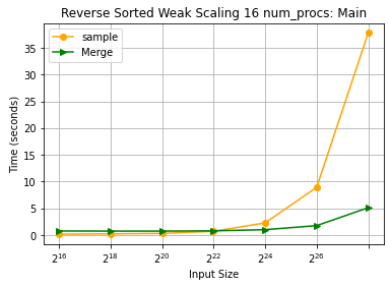
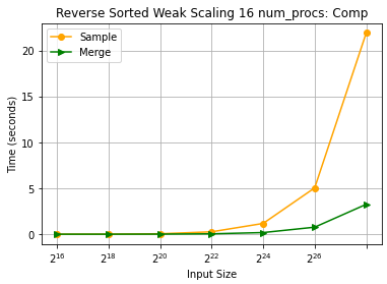
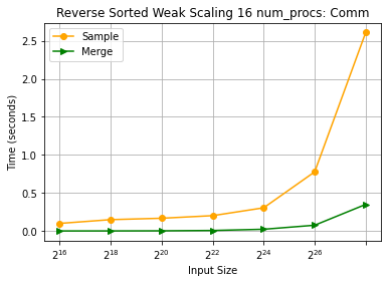
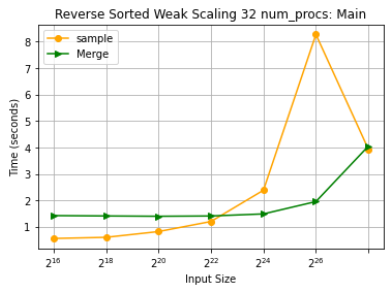
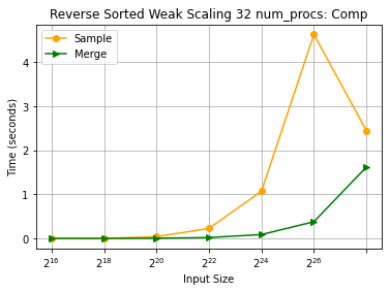
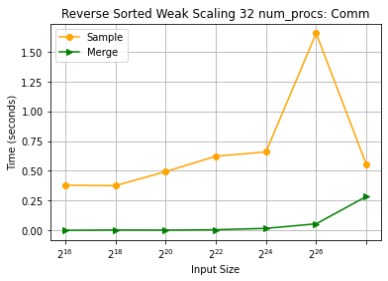
Reverse Sorted

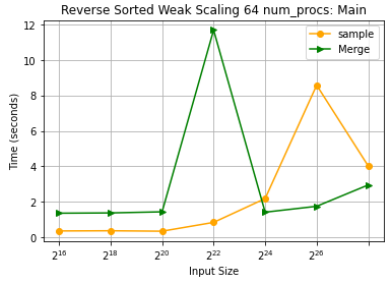
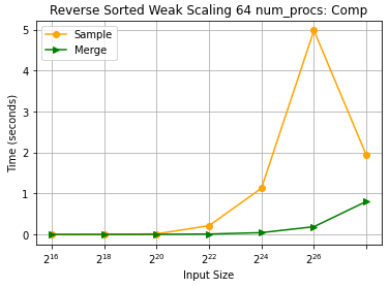
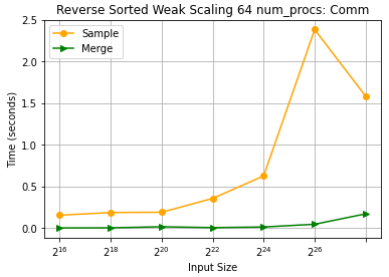
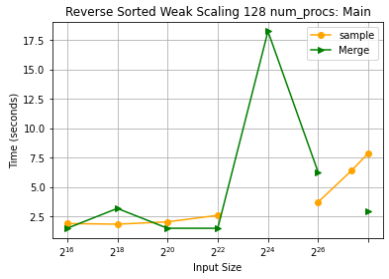
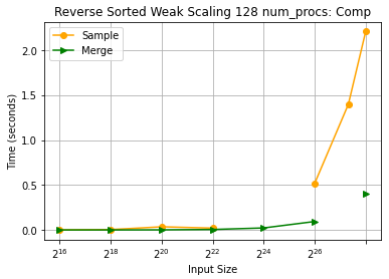
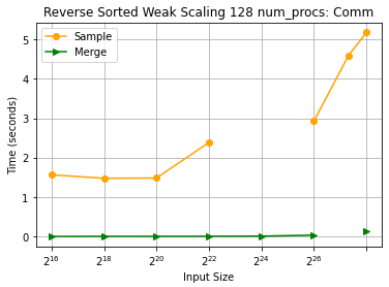
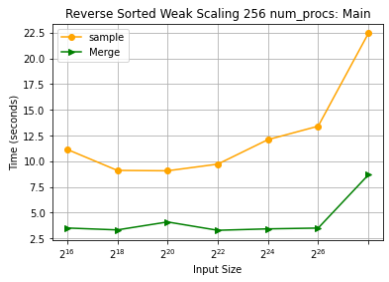
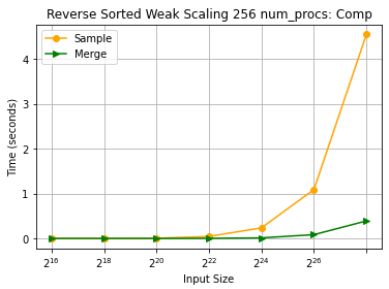
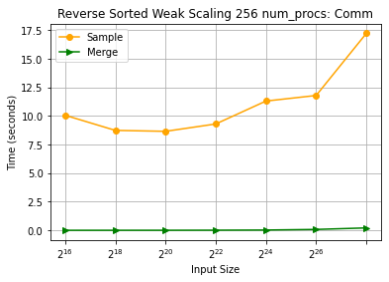
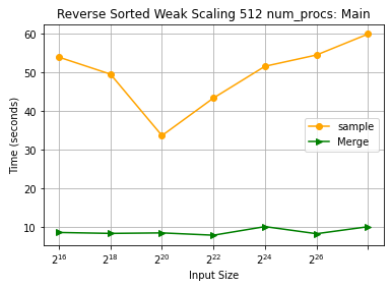
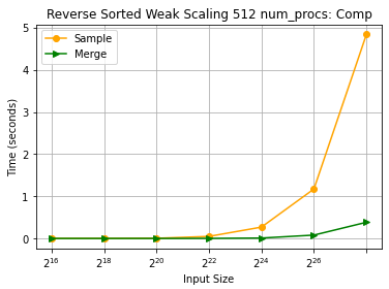
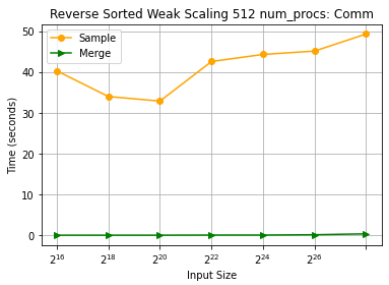
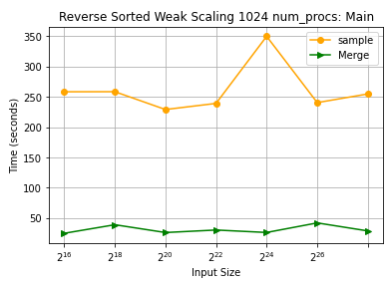
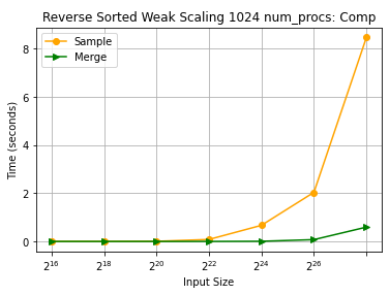
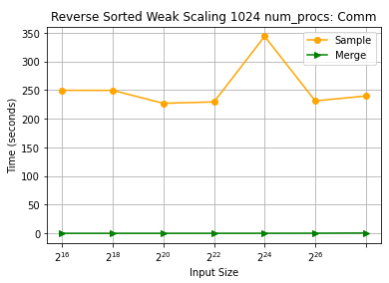
Main	Computation	Communication
		

Main	Computation	Communication
		
		
		
		
		

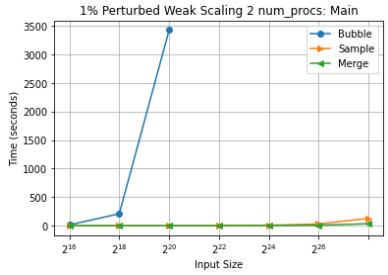
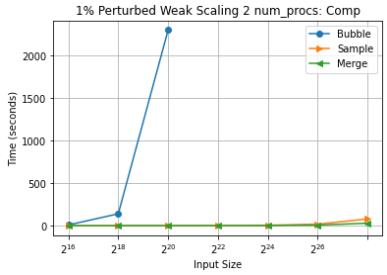
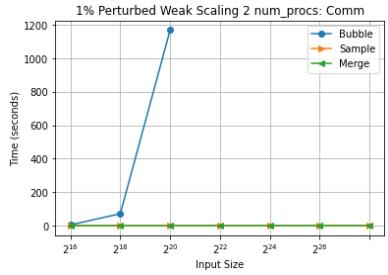
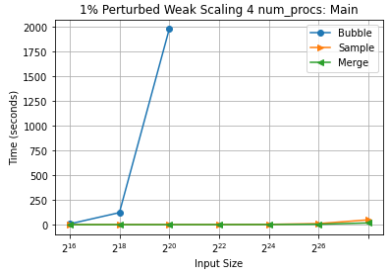
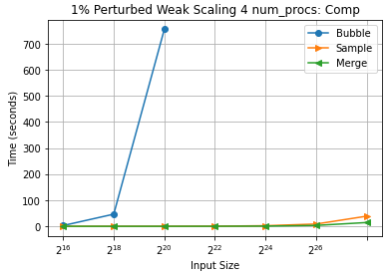
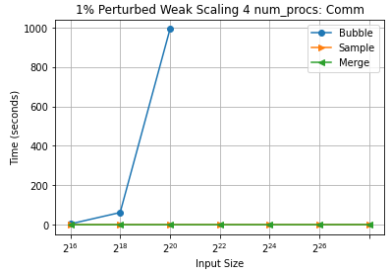
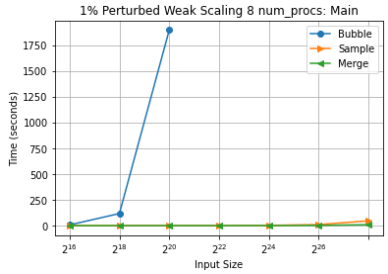
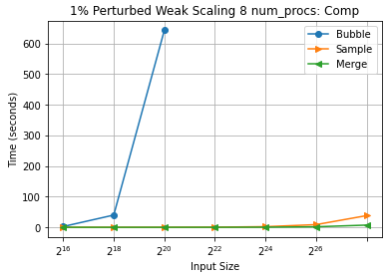
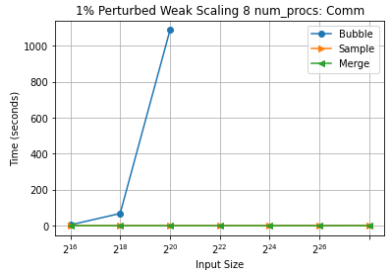
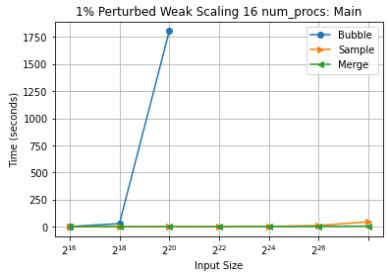
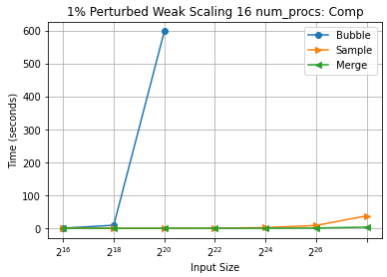
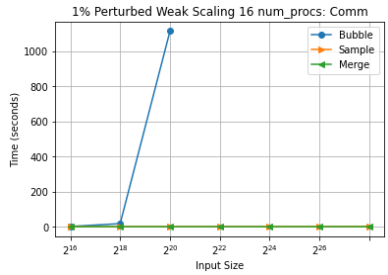
Main	Computation	Communication
		
		
		
		

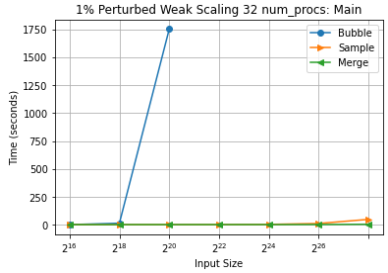
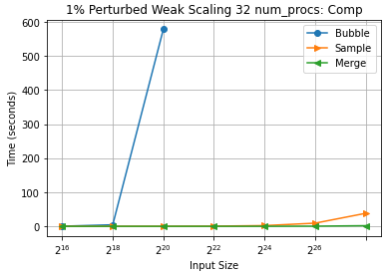
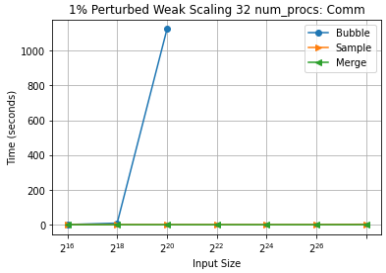
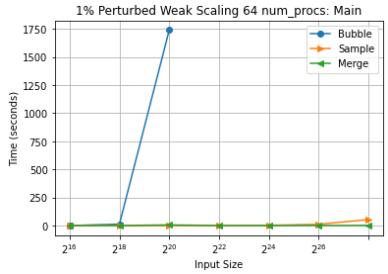
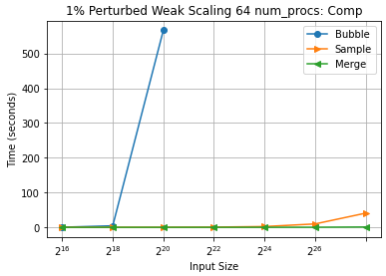
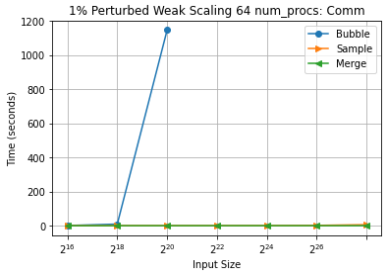
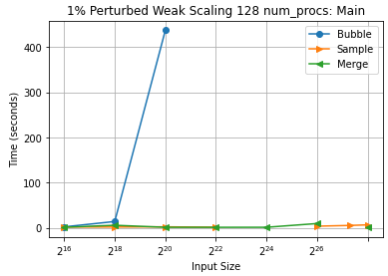
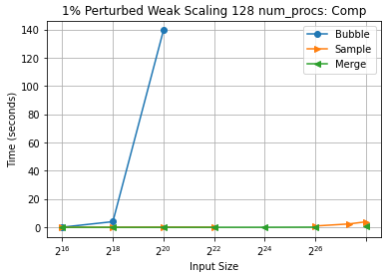
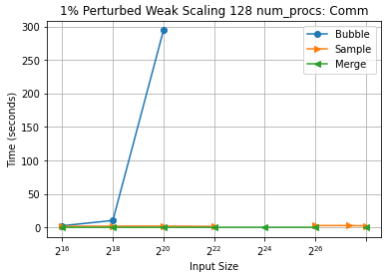
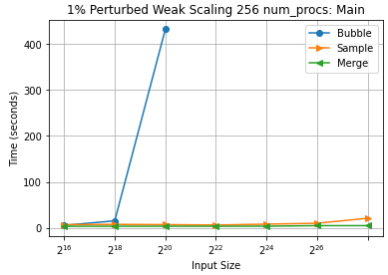
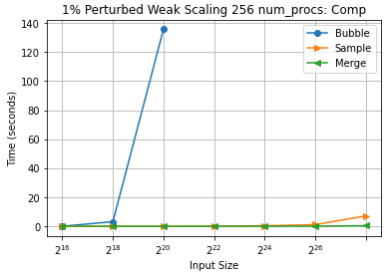
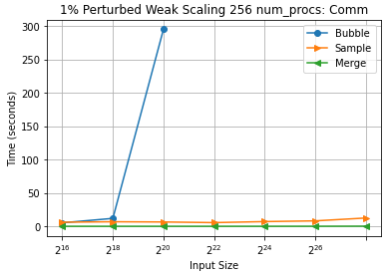
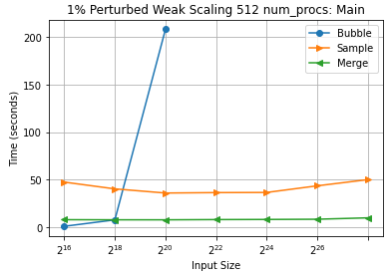
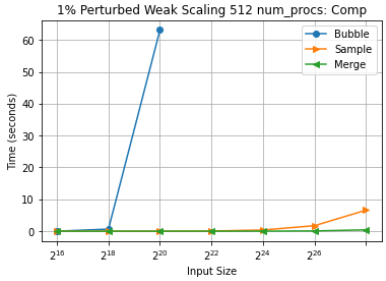
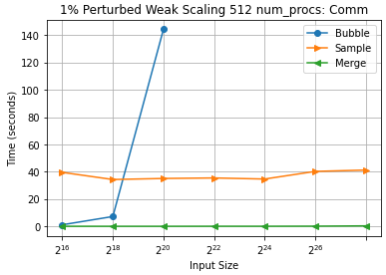
Without Bubble Sort

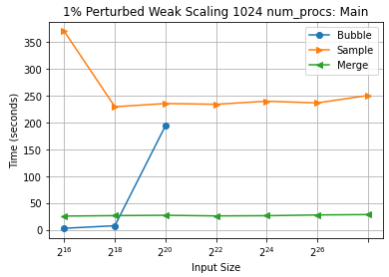
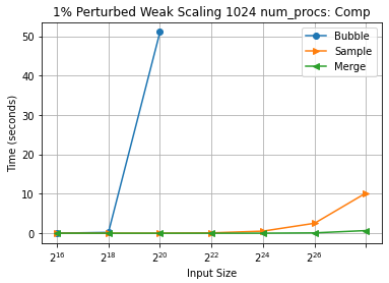
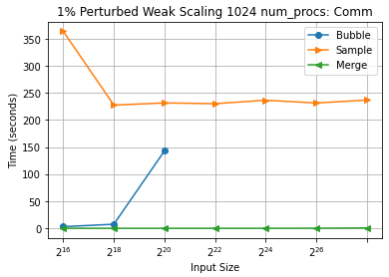
Main	Computation	Communication
		
		
		
		
		

Main	Computation	Communication
		
		
		
		
		

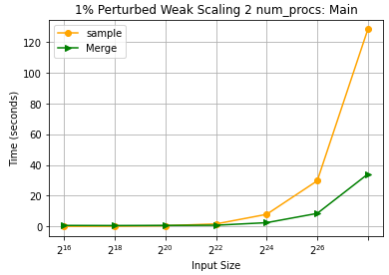
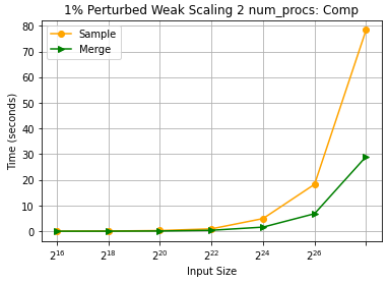
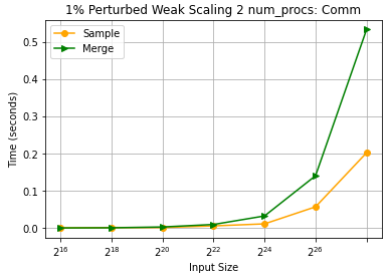
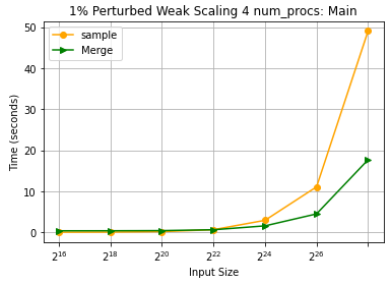
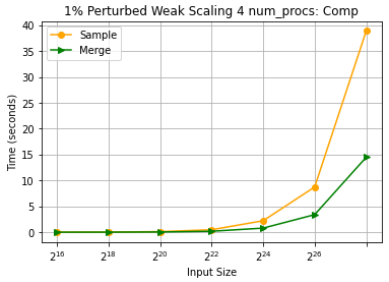
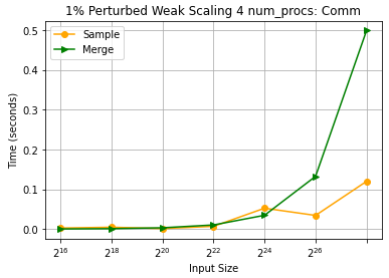
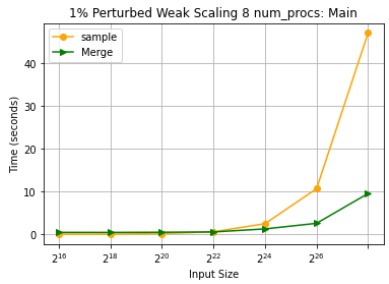
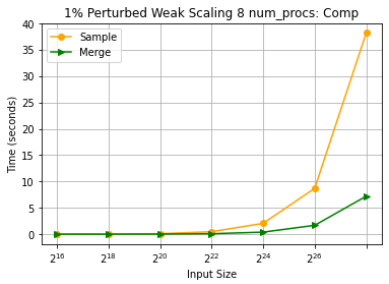
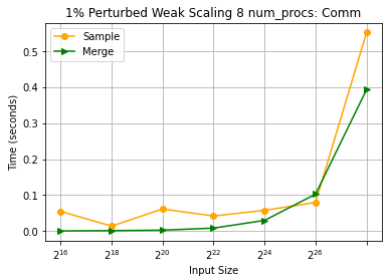
1% Perturbed

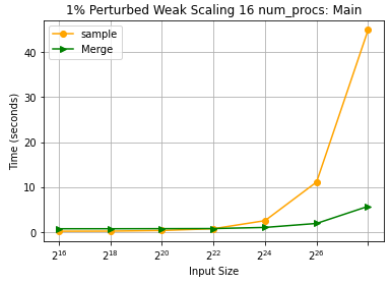
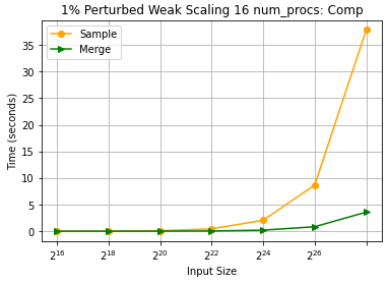
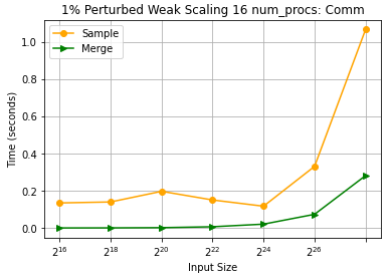
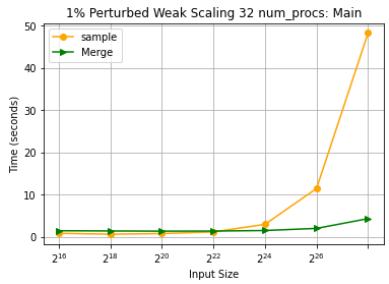
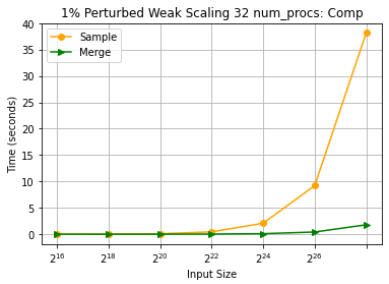
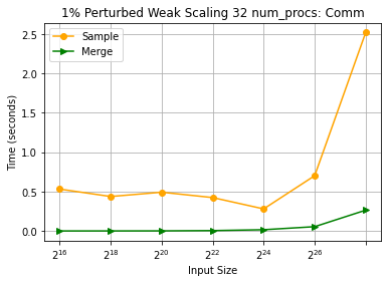
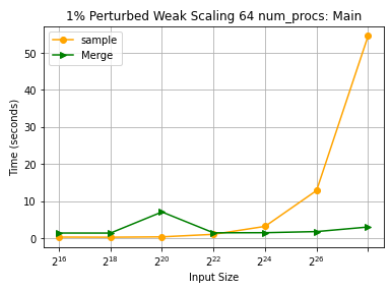
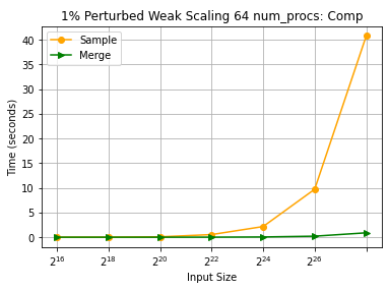
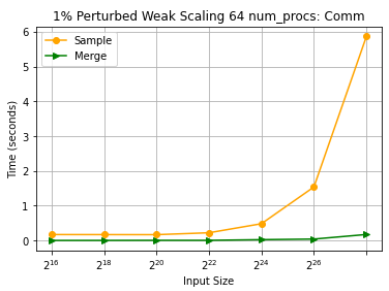
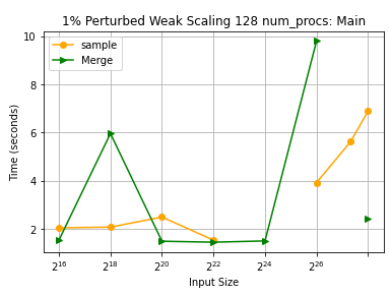
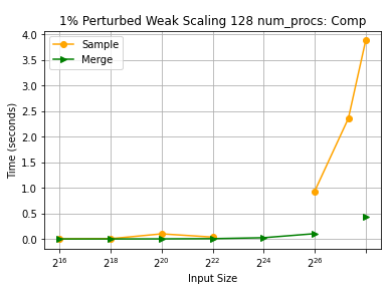
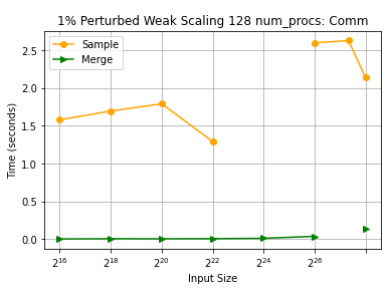
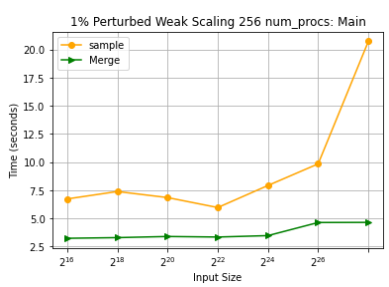
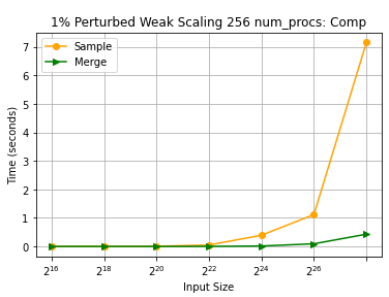
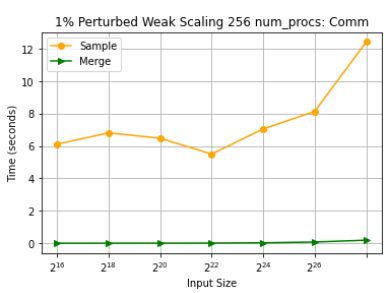
Main	Computation	Communication
		
		
		
		

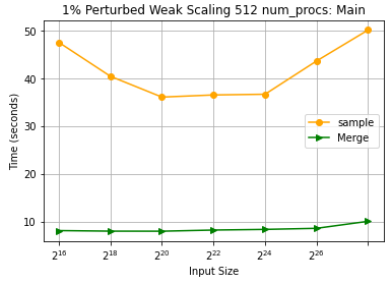
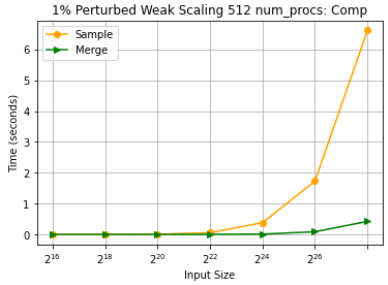
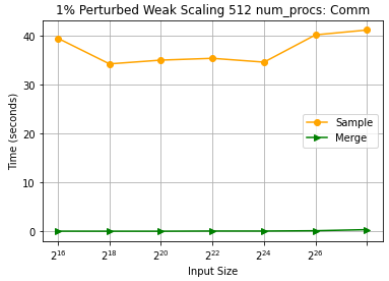
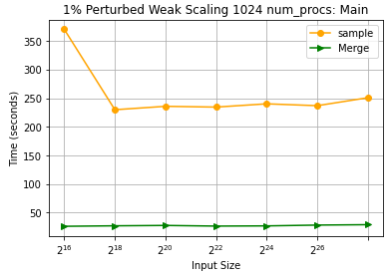
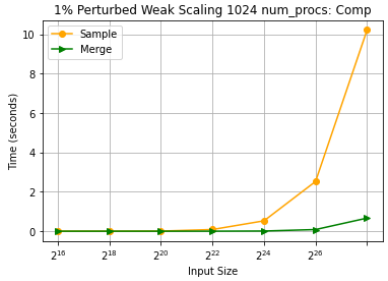
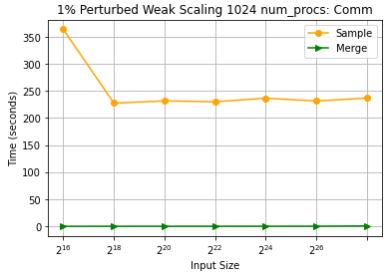
Main	Computation	Communication
		
		
		
		
		

Main	Computation	Communication
		

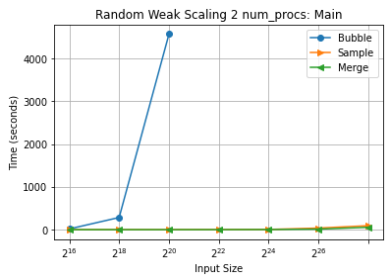
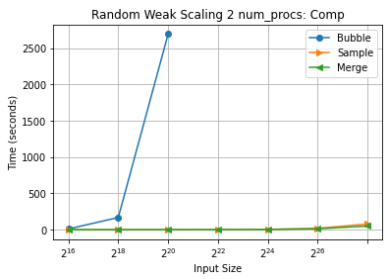
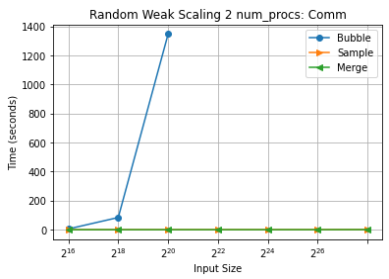
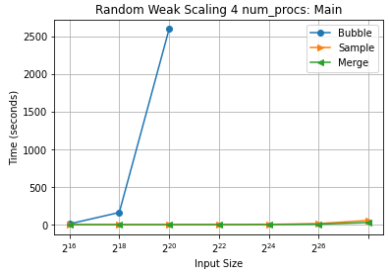
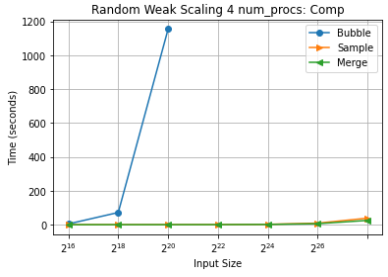
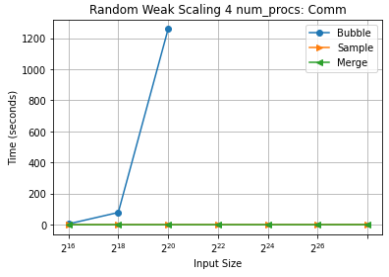
WIthout Bubble Sort

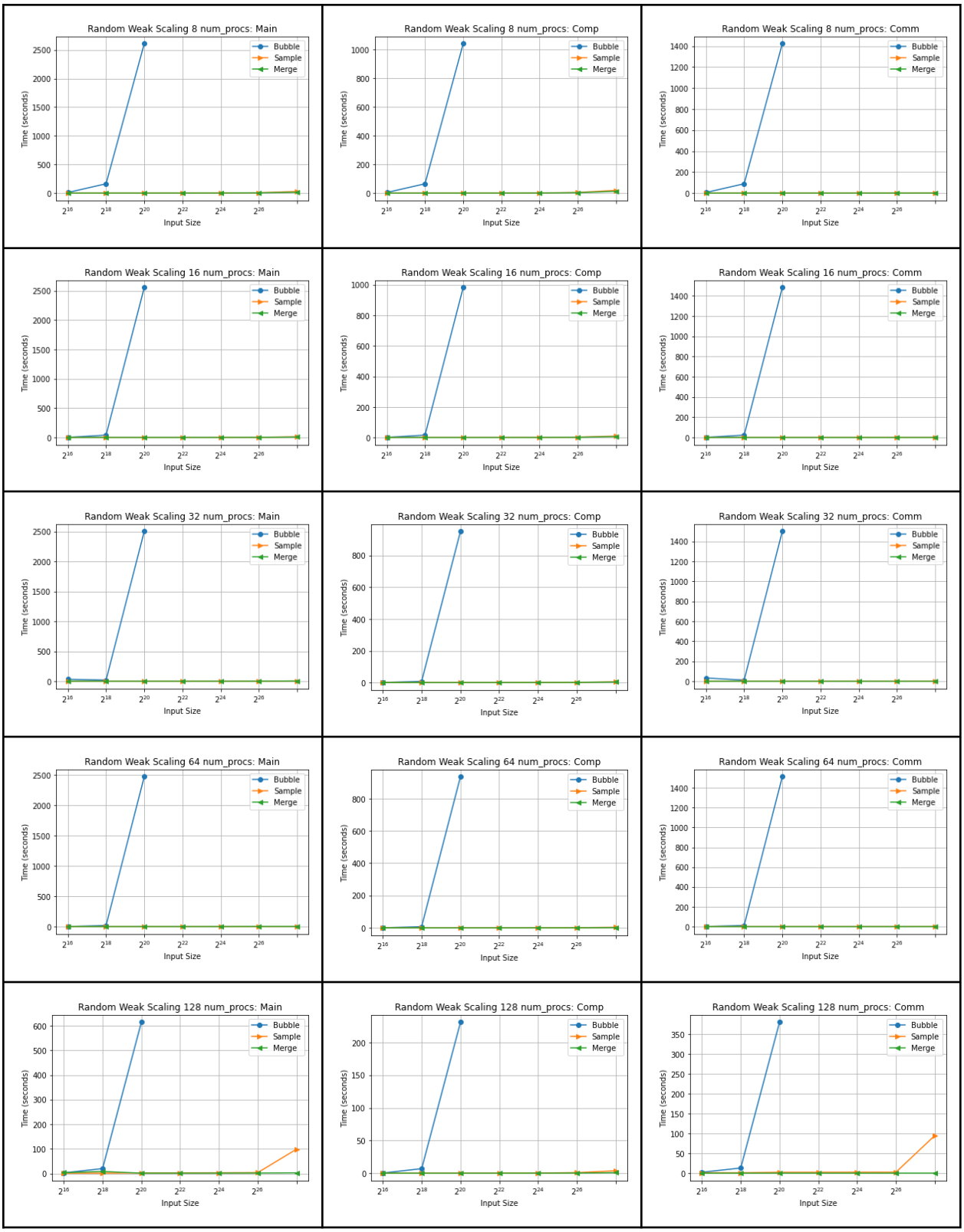
Main	Computation	Communication
		
		
		

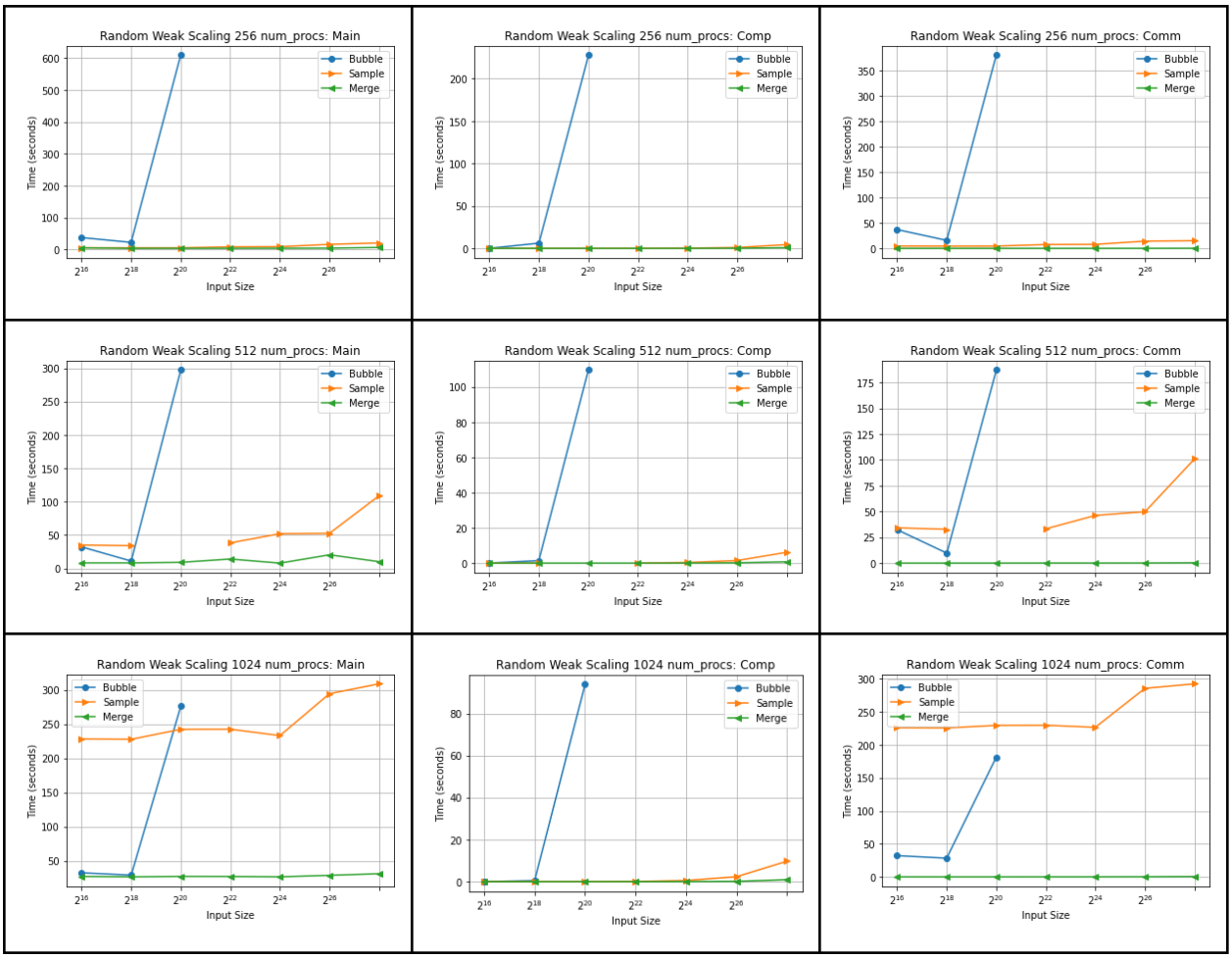
Main	Computation	Communication
		
		
		
		
		

Main	Computation	Communication
		
		

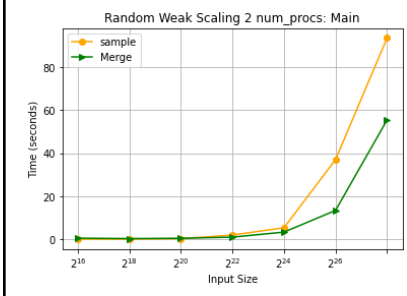
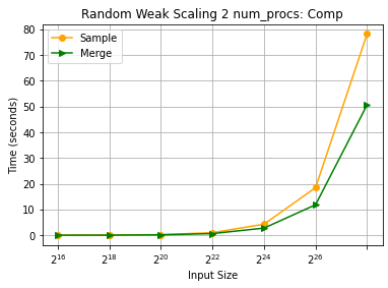
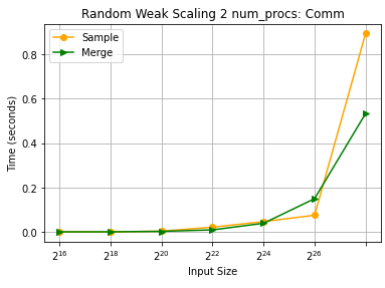
Random

Main	Computation	Communication
		
		

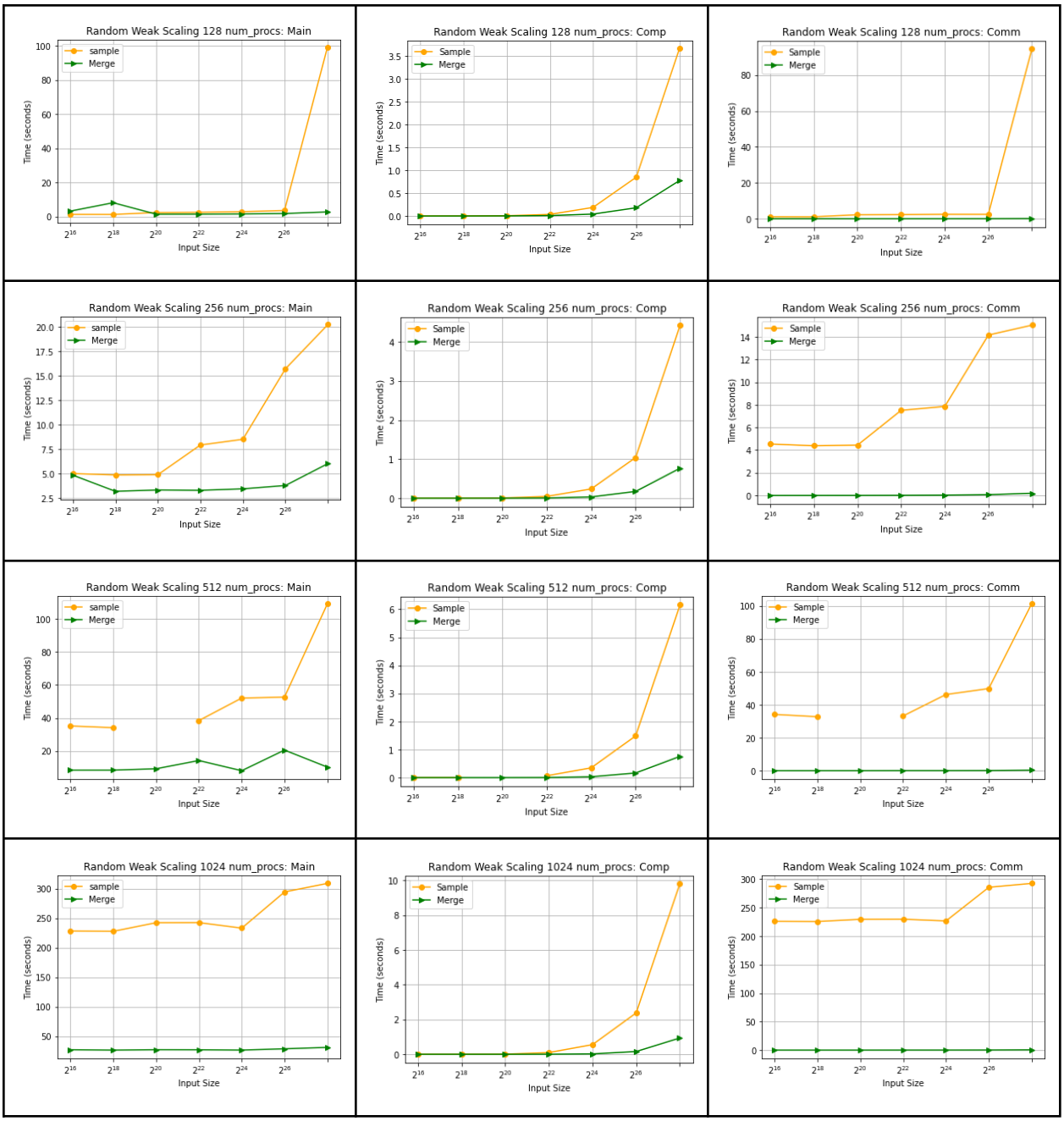

Random Weak Scaling 8 num_procs: Main
Random Weak Scaling 8 num_procs: Comp
Random Weak Scaling 8 num_procs: Comm
Random Weak Scaling 16 num_procs: Main
Random Weak Scaling 16 num_procs: Comp
Random Weak Scaling 16 num_procs: Comm
Random Weak Scaling 32 num_procs: Main
Random Weak Scaling 32 num_procs: Comp
Random Weak Scaling 32 num_procs: Comm
Random Weak Scaling 64 num_procs: Main
Random Weak Scaling 64 num_procs: Comp
Random Weak Scaling 64 num_procs: Comm
Random Weak Scaling 128 num_procs: Main
Random Weak Scaling 128 num_procs: Comp
Random Weak Scaling 128 num_procs: Comm
Bubble
Sample
Merge
Time (seconds)
Input Size



Without Bubble Sort

Main	Computation	Communication
		


Random Weak Scaling 4 num_procs: Main
Random Weak Scaling 4 num_procs: Comp
Random Weak Scaling 4 num_procs: Comm
Random Weak Scaling 8 num_procs: Main
Random Weak Scaling 8 num_procs: Comp
Random Weak Scaling 8 num_procs: Comm
Random Weak Scaling 16 num_procs: Main
Random Weak Scaling 16 num_procs: Comp
Random Weak Scaling 16 num_procs: Comm
Random Weak Scaling 32 num_procs: Main
Random Weak Scaling 32 num_procs: Comp
Random Weak Scaling 32 num_procs: Comm
Random Weak Scaling 64 num_procs: Main
Random Weak Scaling 64 num_procs: Comp
Random Weak Scaling 64 num_procs: Comm
sample
Sample
Merge
Time (seconds)
Input Size

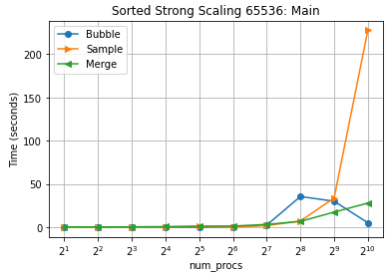
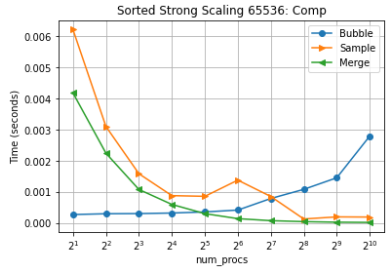
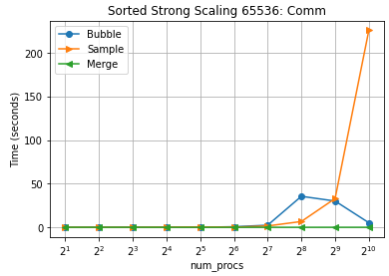
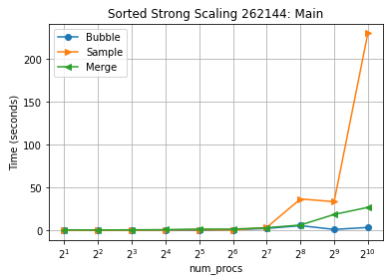
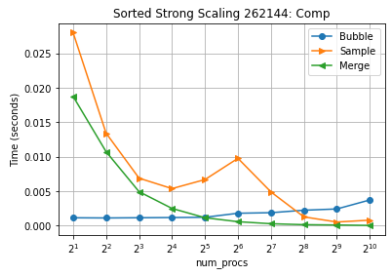
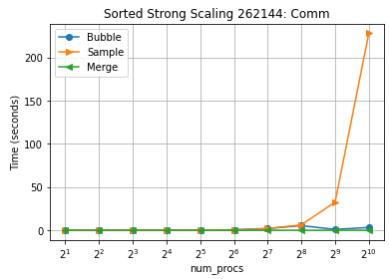
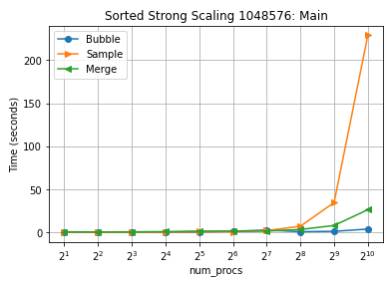
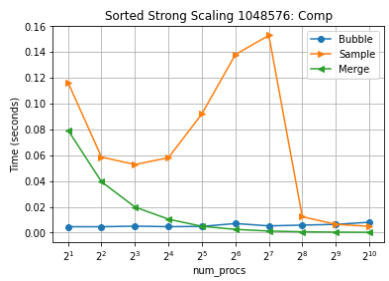
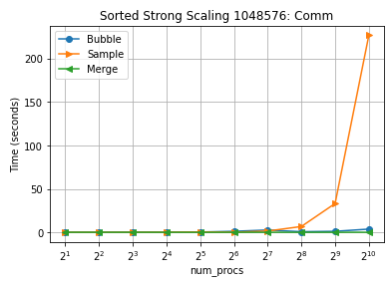


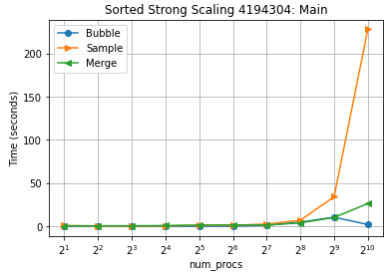
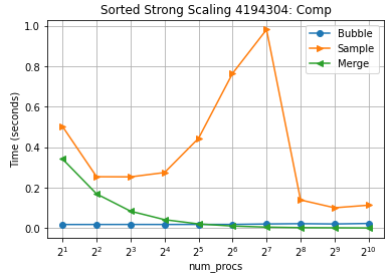
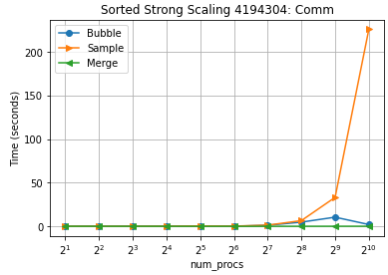
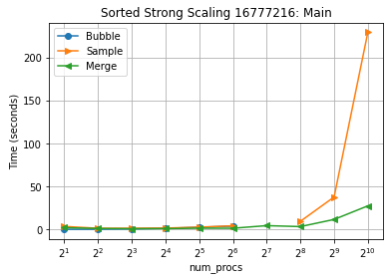
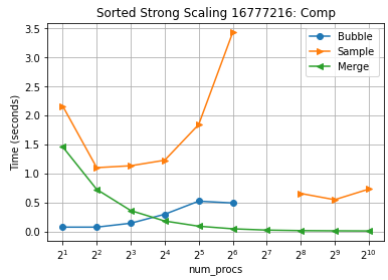
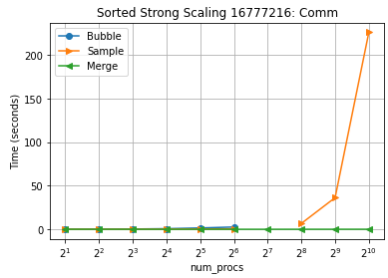
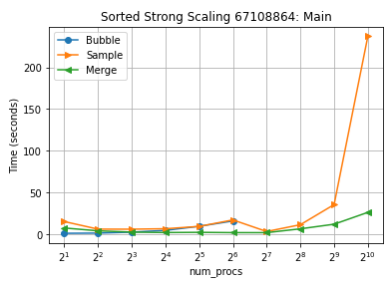
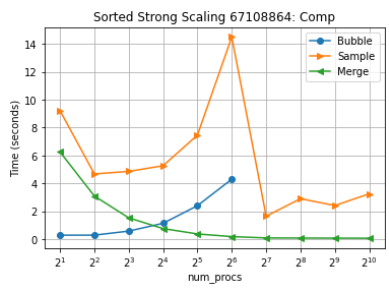
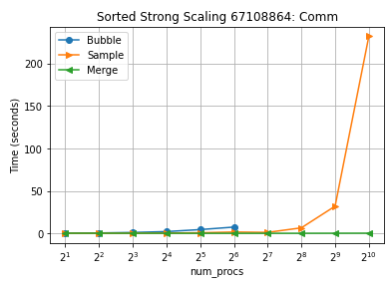
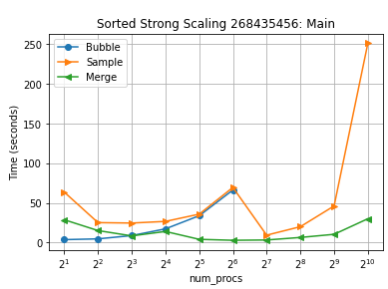
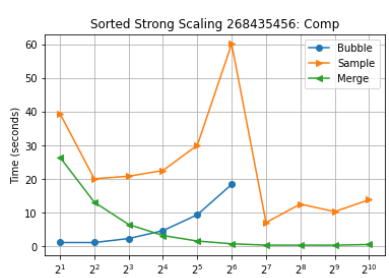
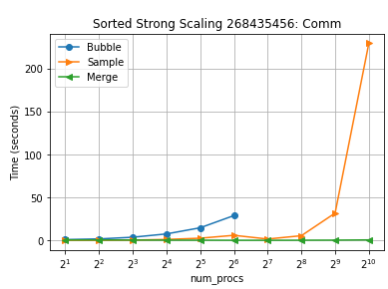
Strong Scaling

For strong scaling it can be seen that all the algorithms do an okay job except when the number of processes gets too high. The computation times continue to decrease as extra processes are added which is expected, however that is not the case for communication. Especially with

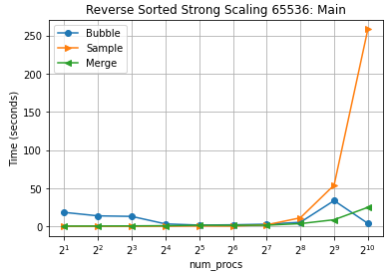
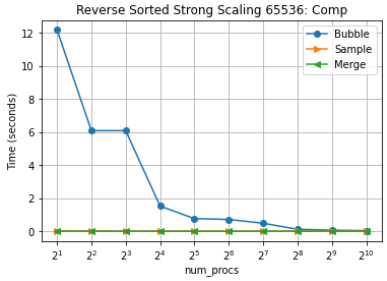
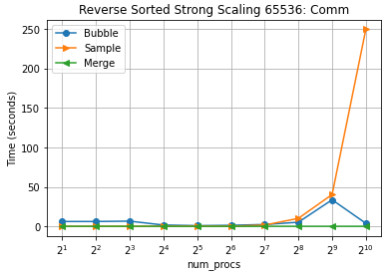
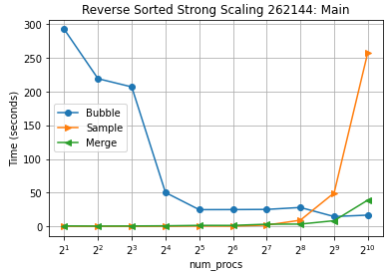
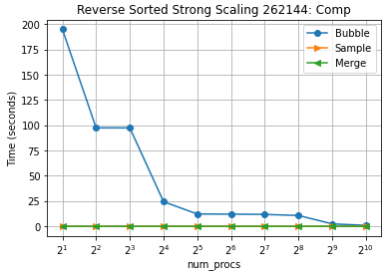
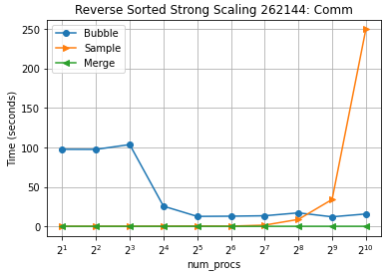
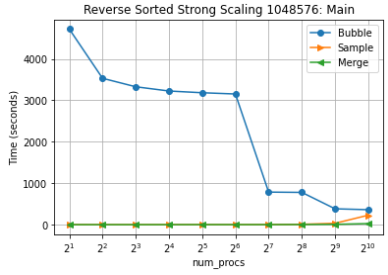
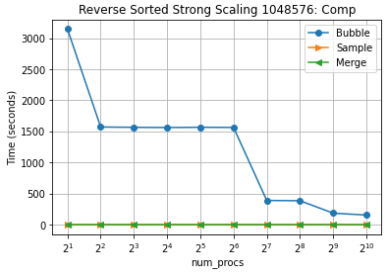
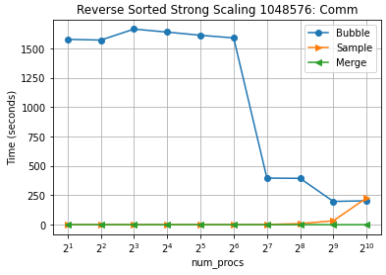
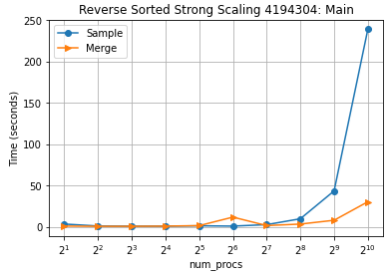
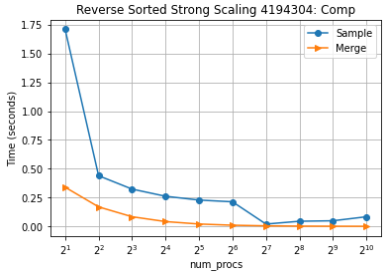
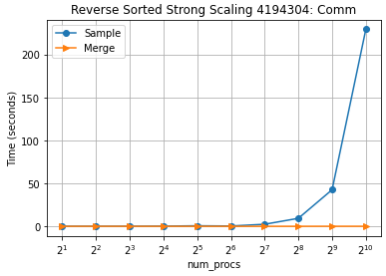
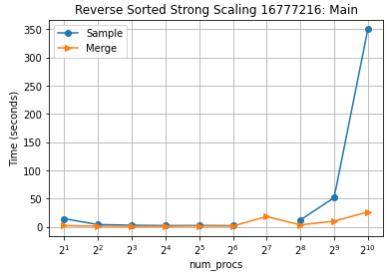
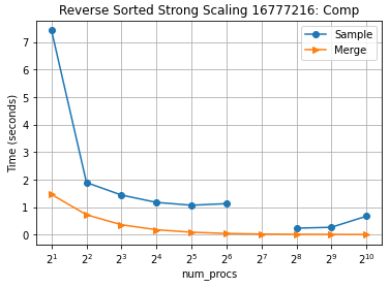
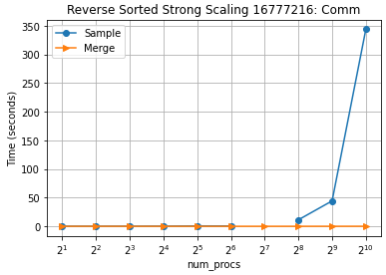
sample sort where the communication overhead begins to grow exponentially with the number of processes. It is also worth noting that while the computation times are benefiting from strong scaling for most of the algorithms the computation time is a magnitude smaller than communication meaning that this increase is quite pointless if it causes the communication time to increase.

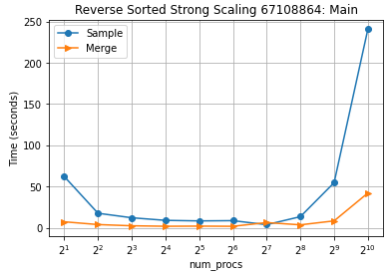
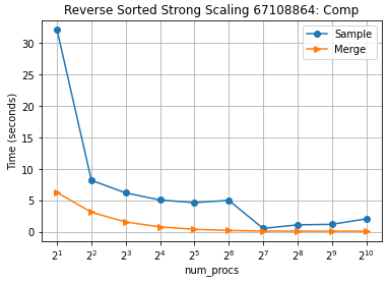
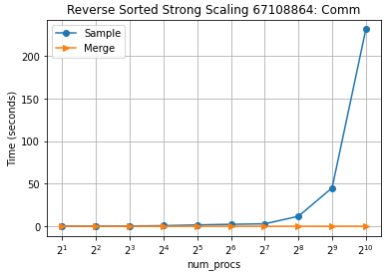
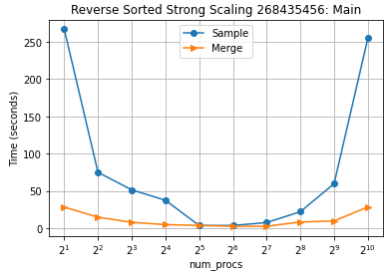
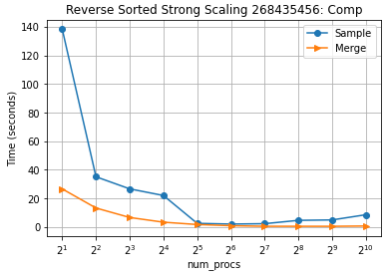
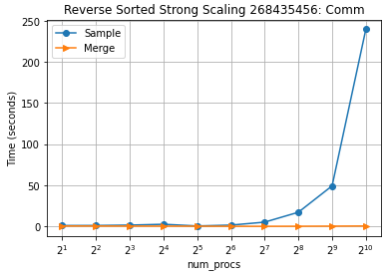
Sorted

Main	Computation	Communication
		
		
		

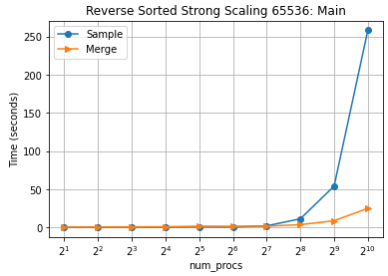
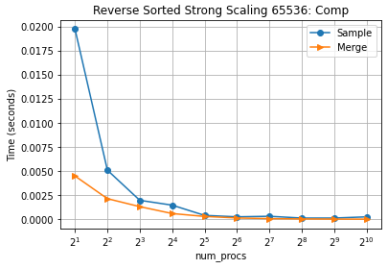
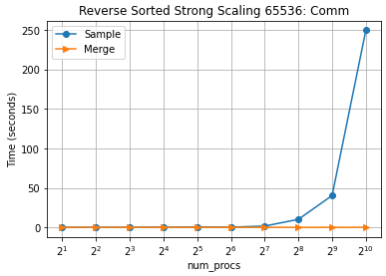
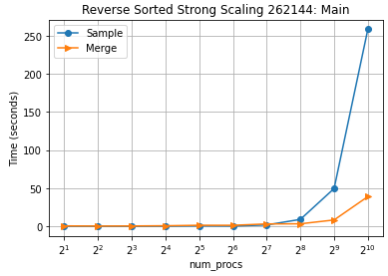
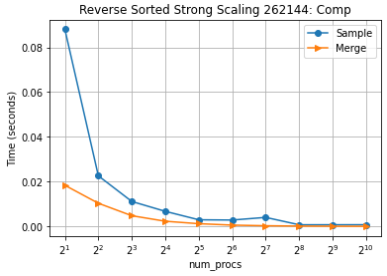
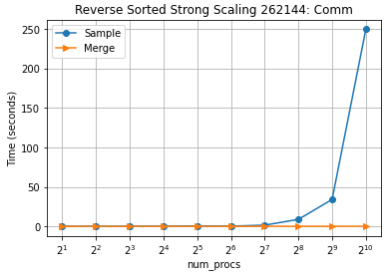
Main	Computation	Communication
		
		
		
		

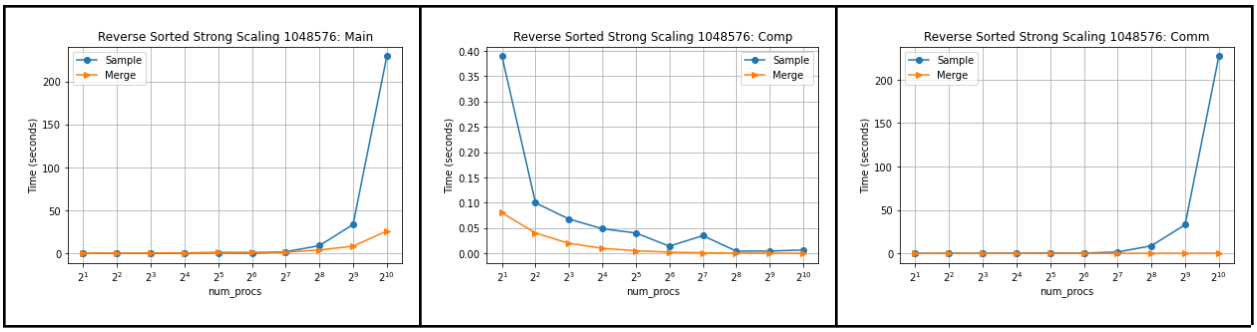
Reverse Sorted

Main	Computation	Communication
		
		
		
		
		

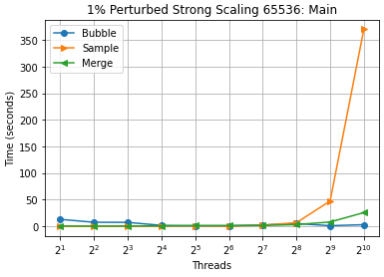
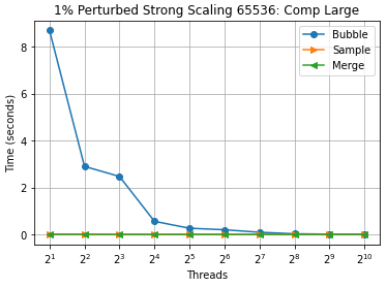
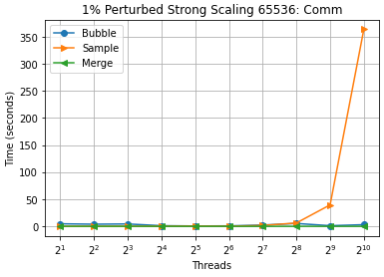
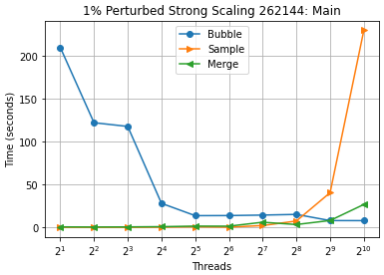
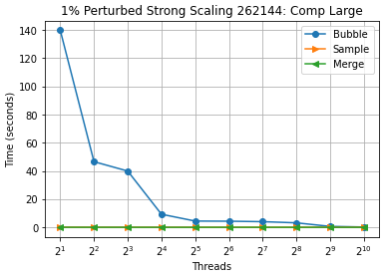
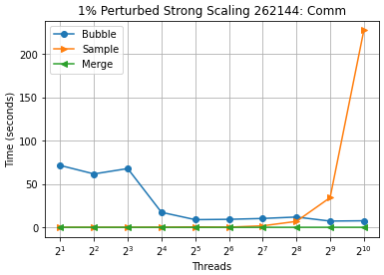
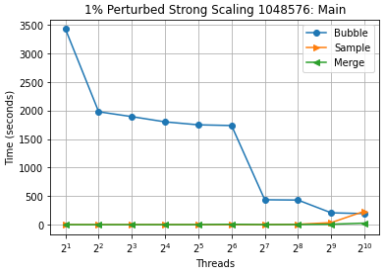
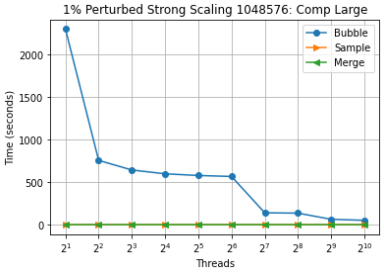
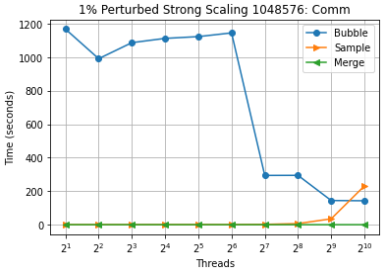
Main	Computation	Communication
		
		

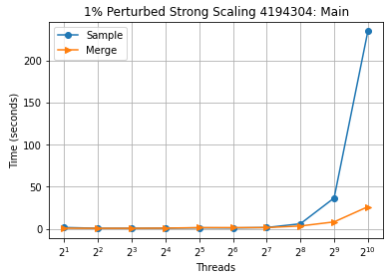
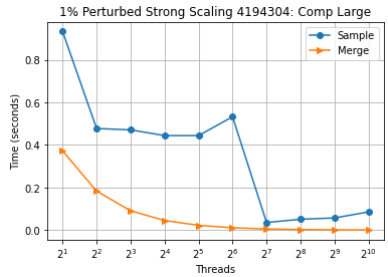
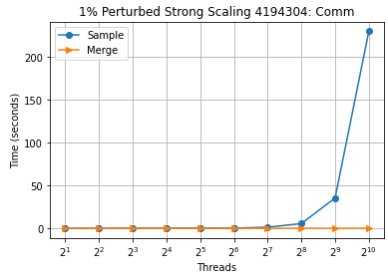
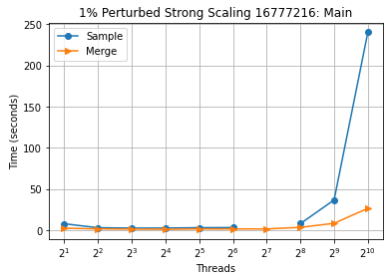
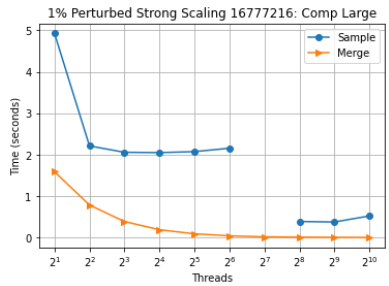
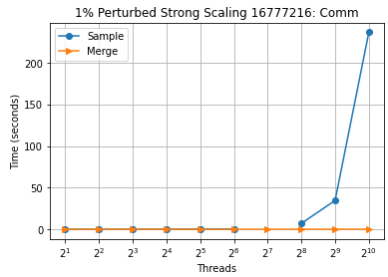
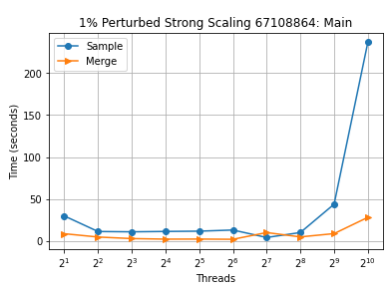
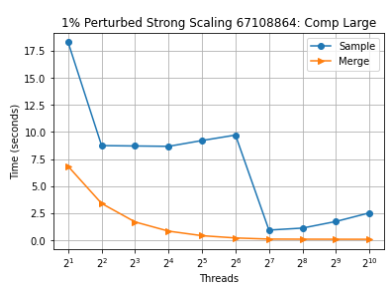
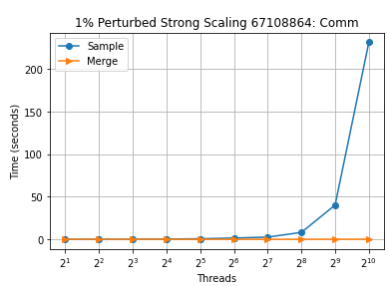
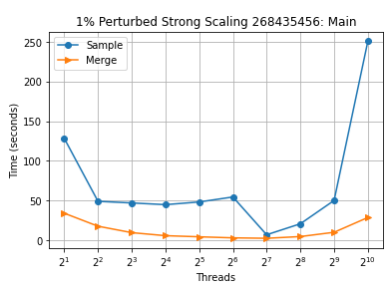
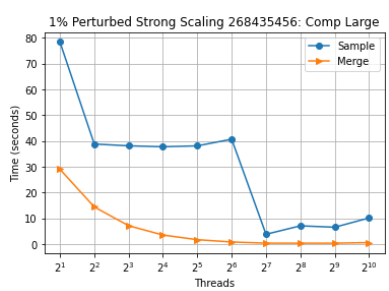
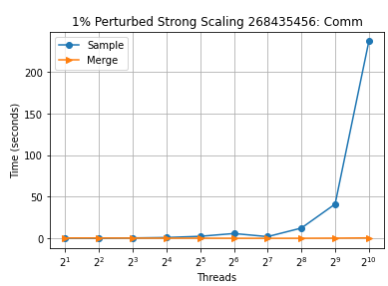
Without Bubble Sort

Main	Computation	Communication
		
		



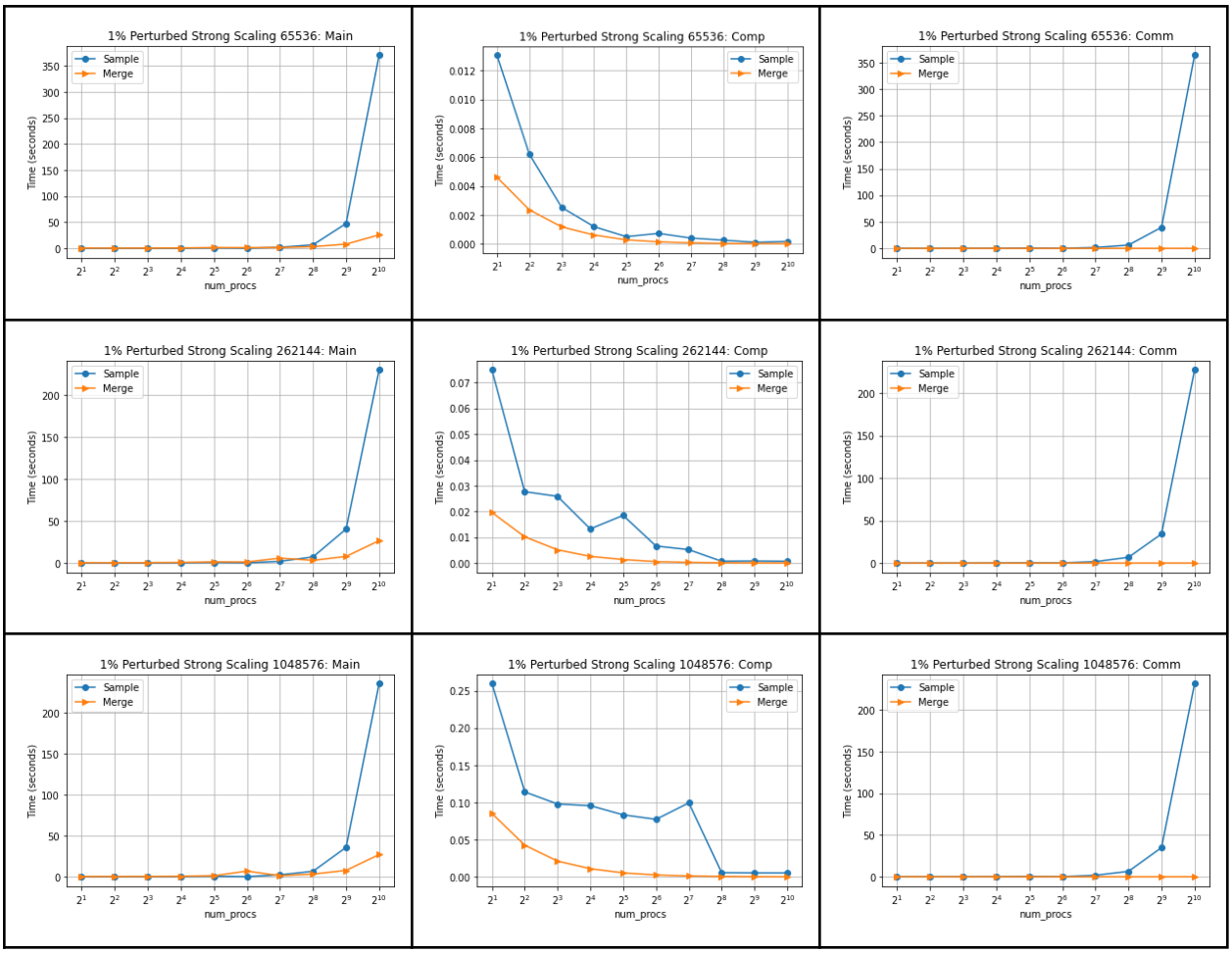
1% Perturbed

Main	Computation	Communication
		
		
		

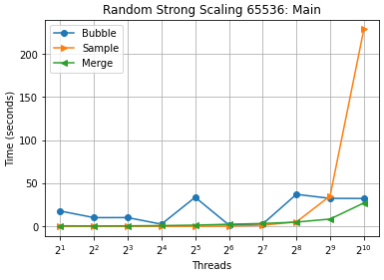
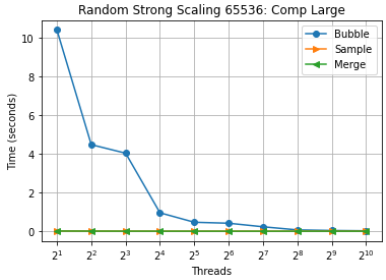
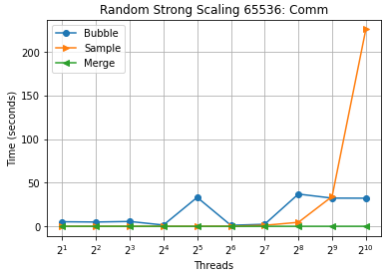
Main	Computation	Communication
		
		
		
		

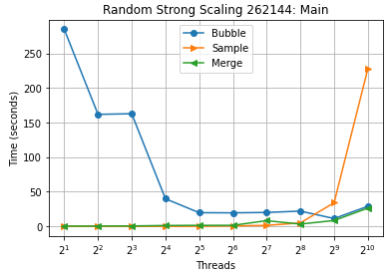
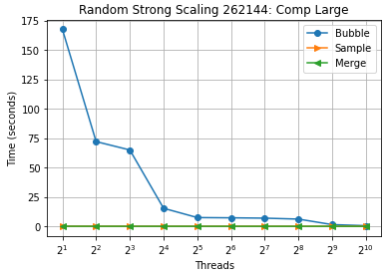
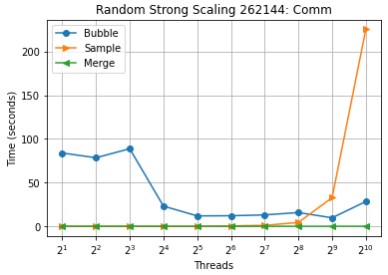
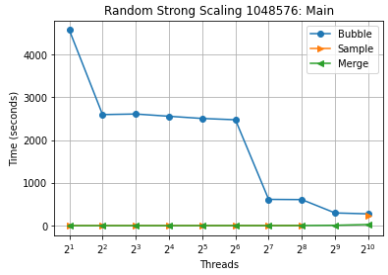
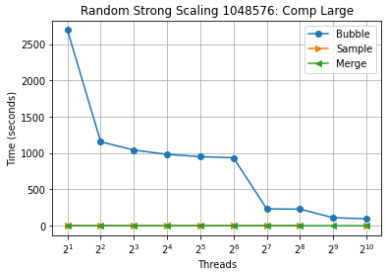
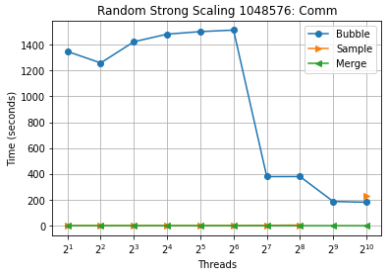
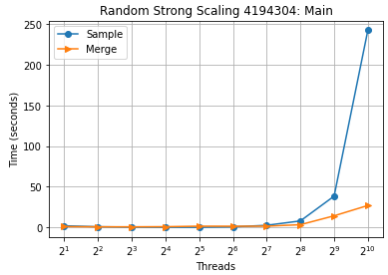
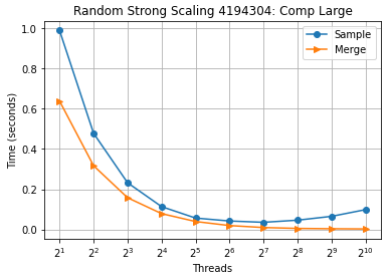
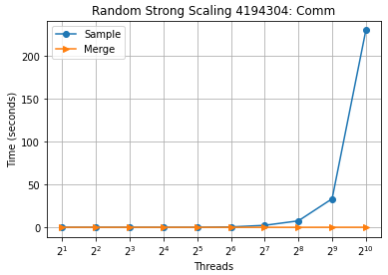
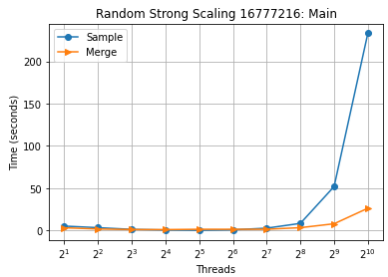
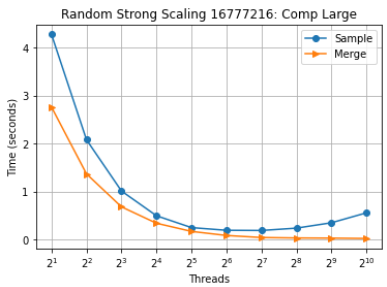
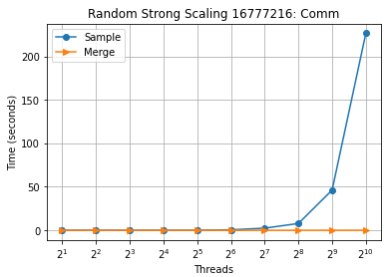
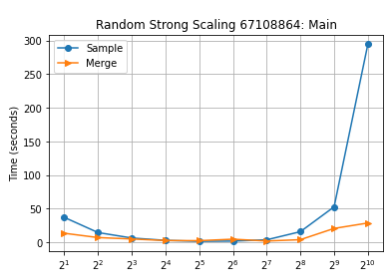
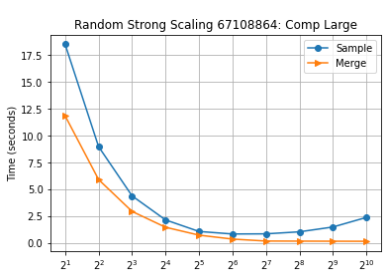
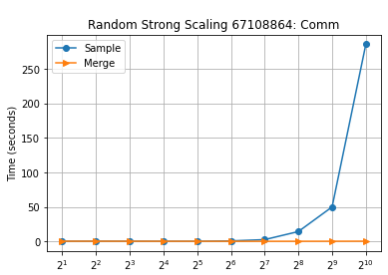
Without Bubble Sort

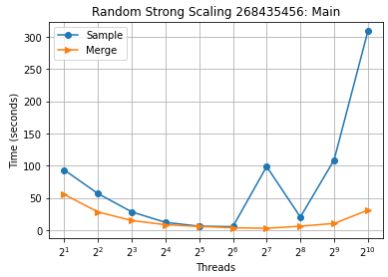
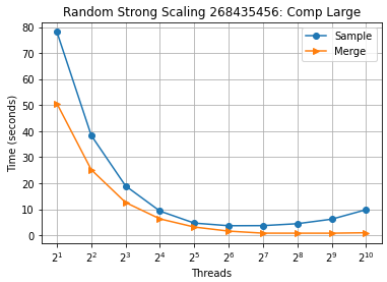
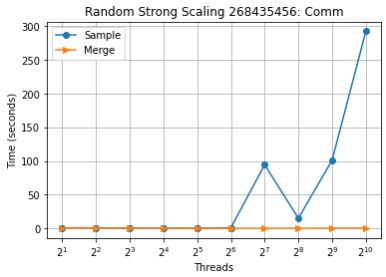
Main	Computation	Communication

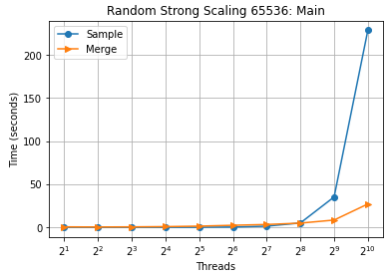
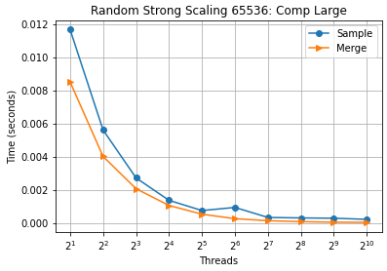
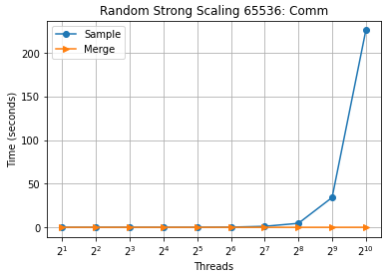
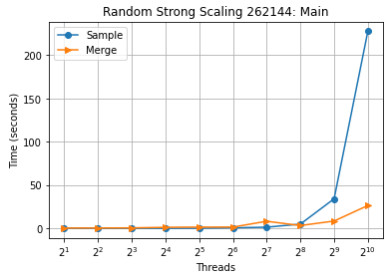
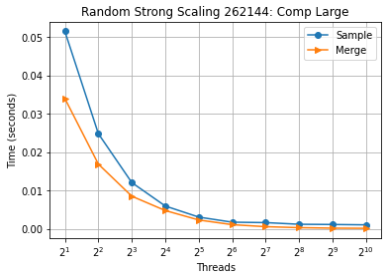
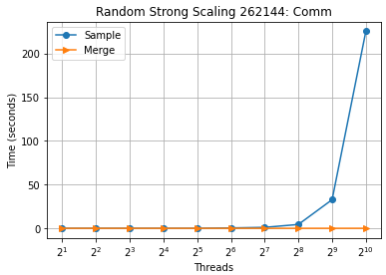
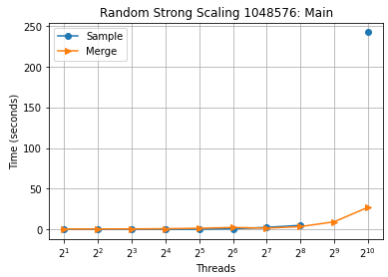
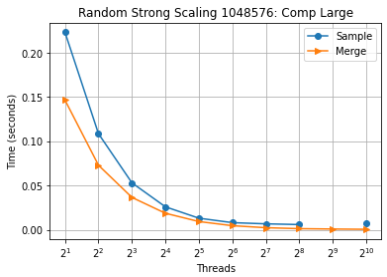
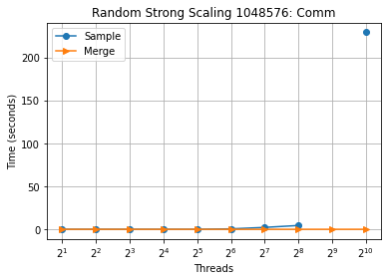


Random

Main	Computation	Communication
		

Main	Computation	Communication
		
		
		
		
		

Main	Computation	Communication
		

Main	Computation	Communication
		
		
		

Weak Scaling Sorted Graphs:

Analyzing the weak scaling graphs above, each of the three MPI sorting algorithms seems to perform relatively similarly and relatively well on this sorted data. As the number of threads increases, time stays relatively the same for merge and bubble sort. However, as the number of threads increases on sample sort, the time of main shoots up starting at about 2^8 threads. This could be because sample sort relies on lots of communication between threads. Each thread sends and receives to the others meaning as you add threads, the cost of communication significantly increases, driving the run time up. The graph with the most variance is the largest input size. This is where we see the run time start higher and decrease as the number of threads increases. This is because adding more threads splits up the work better. If your input size isn't large enough then more threads just create more overhead which is what we see with sample sort's run time shooting up with the final increase of threads.

Weak Scaling Random Graphs:

The weak scaling graphs for random generated data are quite similar to that of the sorted data. However, one big difference to note is the performance of bubble sort on the randomly generated data. As it runs into many out of order elements, it has to make a lot of swaps, driving up the run time. However, this decreases as the number of threads increase, as the swaps are more efficiently divided between threads.

1% Perturbed Weak Scaling:

The observations for these graphs are almost exactly the same as the Randomly sorted data above. The only difference to note is that the run times are slightly lower for bubble sort on the 1% perturbed data than on the random data. This is because not as many elements are out of place. Merge sort performed the best.

Reverse Sorted Weak Scaling:

Observations to note on these graphs are also mainly the same as the previous sections. Some things to note are that for bubble sort, this algorithm's run time was the highest. This is because each element is as far from its intended location as it can get, meaning it needs the most swaps to get to its sorted position. Additionally, for the largest input size it can be seen that the run time dramatically decreases, bottoms out, then increases again. This displays how at first adding threads increases efficiency and lightens the load but adding to many then drives up the run time due to communication overhead.

Strong Scaling Sorted:

For the graphs above, from 2-64 threads the runtime for each starting algorithm starts low and remains low until the 2^24 input size, at which it rockets up. This makes sense because as the input size goes up, you would ideally have more threads to split the work. A lower number of threads then results in the shooting up of the runtime at the end. As the number of threads increases the lines begin to become more level than sloped. For 512 and 1024 threads, they're than is needed so run time stays consistent, for sample sort being extremely high. For the 256 graphs we see that run time goes down and then up again for input sizes. This is because it's too may threads for a small input size, causing high run time that then come back down a little as the input size fits it more.

Strong Scaling Random:

When comparing the graphs above, it appears that while merge and sample sort have fairly constant runtimes for the same number of threads regardless of input size, bubble sort has a sharp increase in runtime from the input size of 2^18 to 2^20; these trends can be observed in all of the graphs, meaning that the number of threads does not have a significant effect on them. The last two graphs (for 512 and 1024 threads) do show slightly different trends in sample sort; the runtimes across input sizes for sample sort appear higher with these numbers of threads, suggesting that the overhead costs of additional parallelization outweigh any speedups. The larger numbers of threads also show slight upticks for runtime with larger input sizes (especially when increasing from 2^26 to 2^28): this makes sense since these are exponential increases in input size, which would be expected to have longer run times.

1% Perturbed Strong Scaling

From 2-256 threads, the graphs above are very similar. Sample and merge sort have a relatively low run time (with sample sort taking a little longer), while bubble sort's run time shoots up for each as the input size increases. This makes sense because although not many elements are out of order it takes lots of swapping to get them into the right place, increasing workload by adding even a small amount of out-of-order elements. As we get into a larger number of theads we see that sample sort has a higher run time (as more communication is required). Merge sort seems to perform well regardless.

Reverse Sorted Strong Scaling:

From looking at the graphs above, similar trends to the other input types can be observed, with bubble sort having sharp increases in runtime with increases in input size, while merge and sample sort have relatively constant, lower runtimes across input sizes for a fixed number of threads. Again, the main variations in runtimes for sample and merge sort were seen in the final two graphs (for 512 and 1024 threads), where there was a sizable increase in runtime for these

algorithms in comparison to the lower numbers of threads; this change was especially apparent in the sample sort with 1024 threads. It is likely that at these points, the costs of adding more threads and communicating between them outweigh the benefits of further parallelization.

weak scaling

Comparing a change in the number of threads running a given input size, it's no surprise that over larger input sizes increasing threads produces faster runtimes. The threshold appears to be around 2^24 when a decrease in runtimes begins to appear as thread sizes increase, and this only gets more prevalent as the input size increases. Before 2^24 inputs the overhead to add threads does not speed up the runtime until 1024 threads, and only on some input sizes. Even then 1024 threads does decrease runtime, it is never less than running the input on lower thread sizes, supporting the idea that the overhead creates a negative impact. Across different input types over the same input sizes, they generally have similar shapes, whether they have a positive relationship that represents an s-curve in the case of sizes below 2^22 and a quadratic curve (in the best case) for larger input sizes. It's interesting to note that Sorted Weak Scaling of 2^26 inputs doesn't follow the trend and instead has a positive relationship as opposed to the negative almost-quadratic relationship the other graphs exhibit. Additionally, in the case of the 2^24 input size, the graphs are all over the place, this seems to be the line across the board where things start changing, and potentially the biggest showing of how the algorithm compares on different input types.

strong scaling

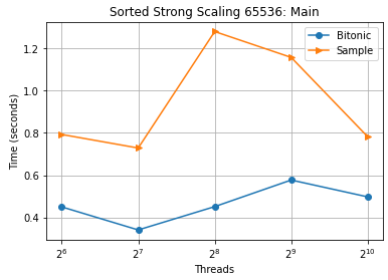
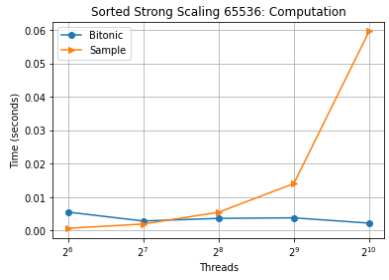
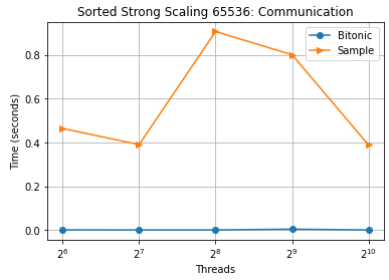
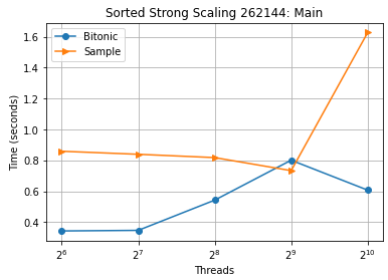
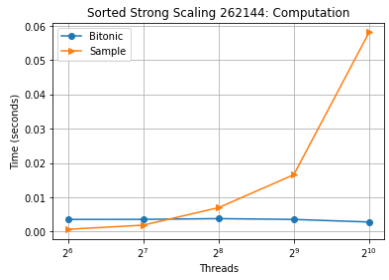
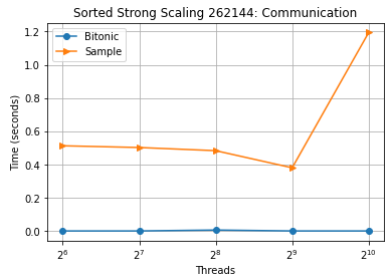
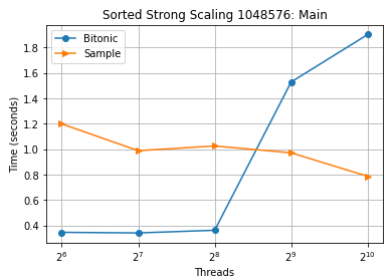
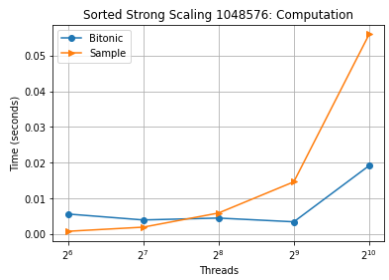
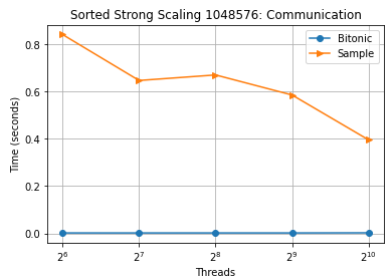
Displaying the data with changing input sizes gives a consistent picture across different input types on the increase in runtime with larger input sizes, which doesn't change regardless of how many threads the data is run on. Obviously the time scale changes for different thread numbers and input types, however the overall trend is the same. All graphs exhibit a similar positive quadratic curve. On the larger thread sizes of 512 and 1024 a peak around input size 2^22 indicates the least efficient combination between threads and input sizes. It's interesting to note that the peaks are fairly consistent in the lag they cause in runtime, however on random data they're about a second slower than the other input types, I would've guessed sorted input would be that way because it begins as one large bitonic step and wouldn't need to be sorted all the way.

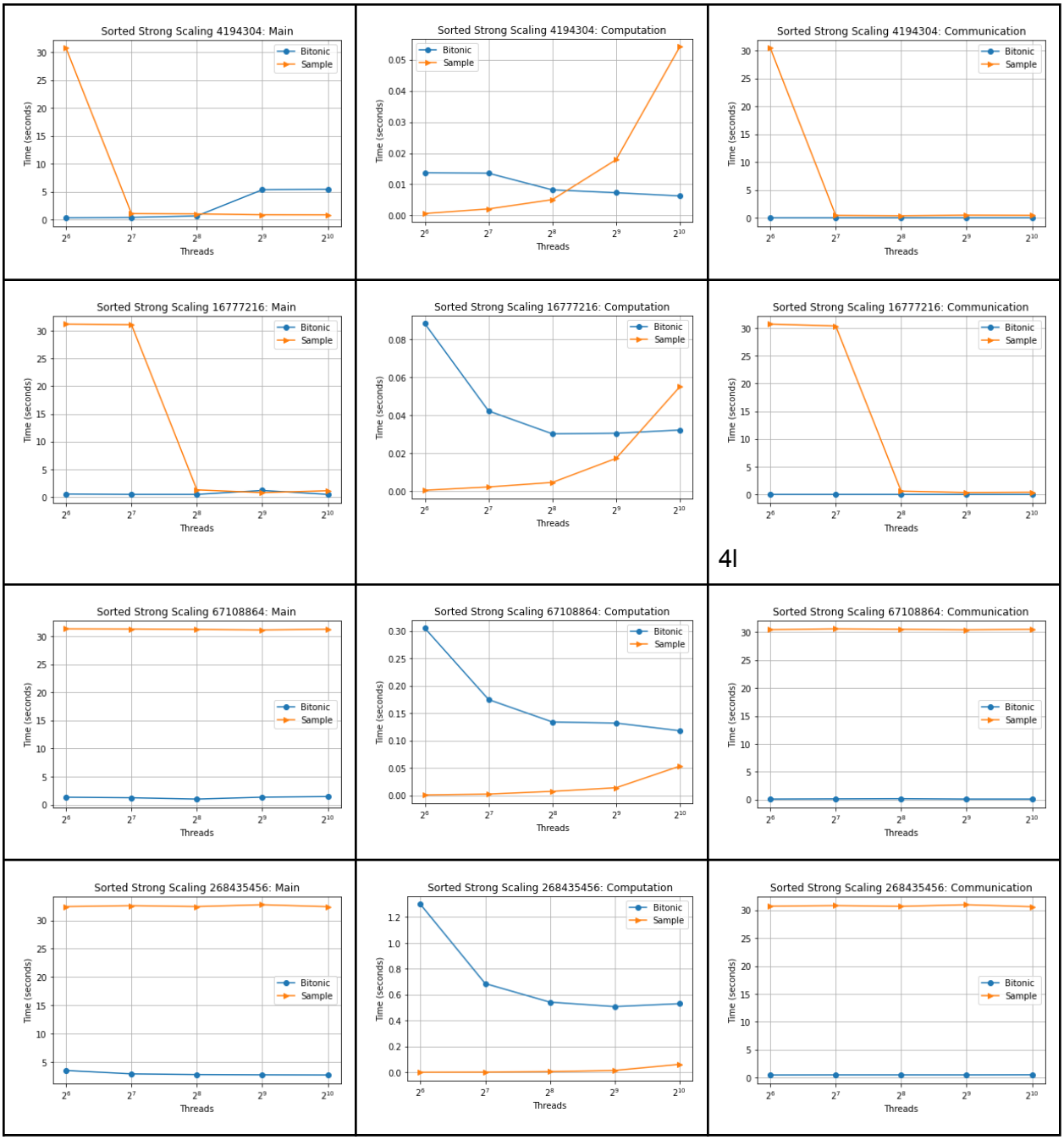
CUDA Comparison Graphs

Strong Scaling

The bitonic and sample sort algorithms scaled well, bitonic scaled well with an almost constant computation and communication time. Sample sort generally increased in computation time on more threads due to the overhead of separating the data more. Since the main function includes data generation and completion checks, it makes sense that the runtime increases with greater input sizes and threads due to the time it takes to integrate the data.

Sorted

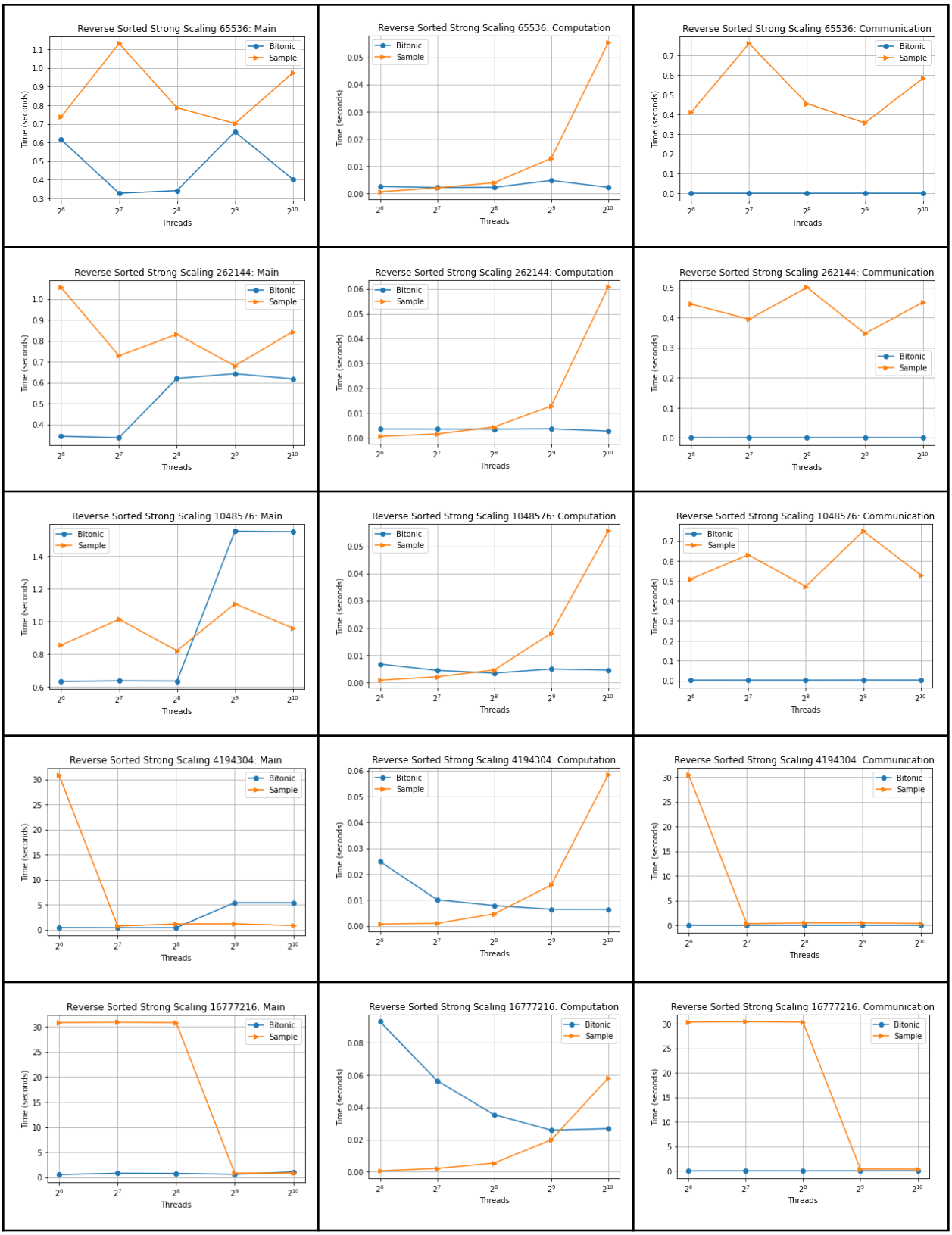
Main	Computation	Communication
		
		
		

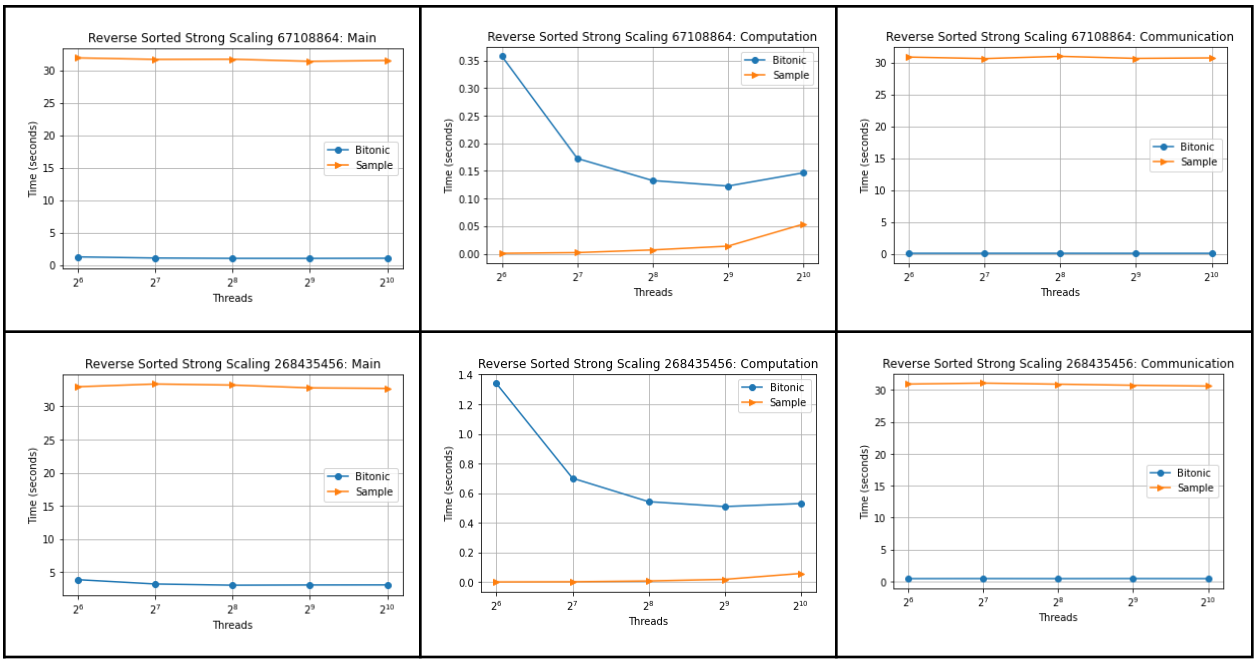


4l

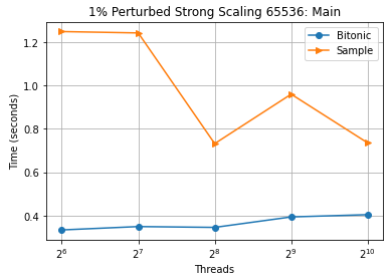
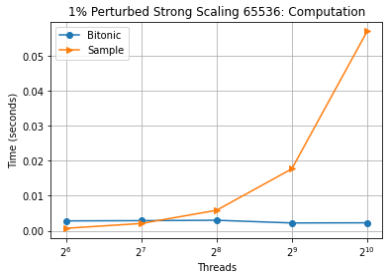
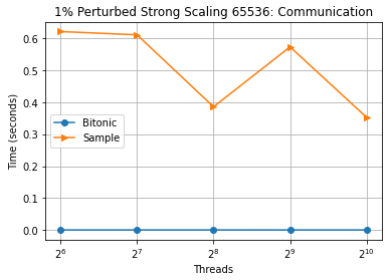
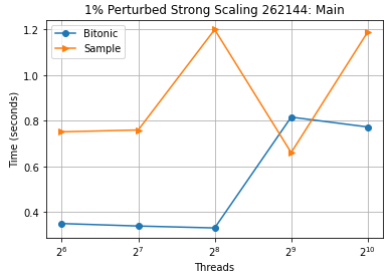
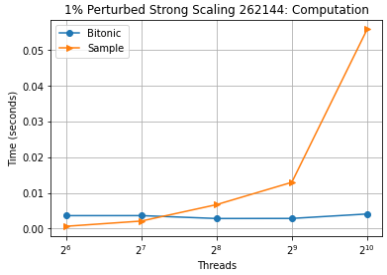
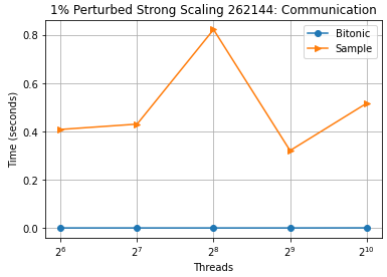
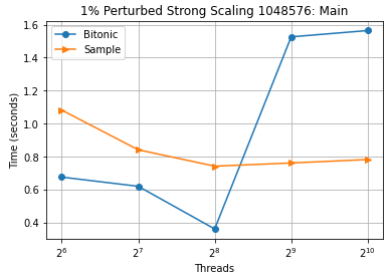
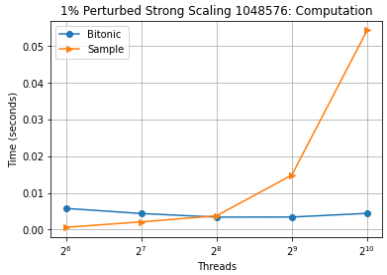
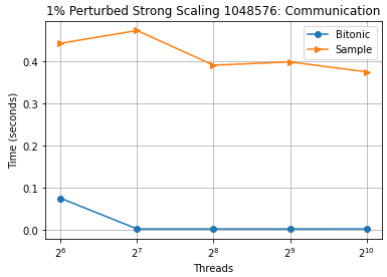
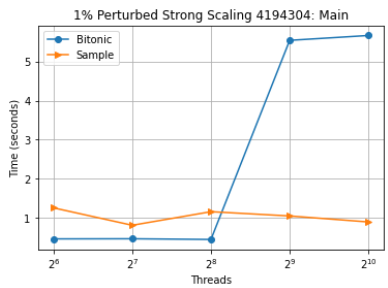
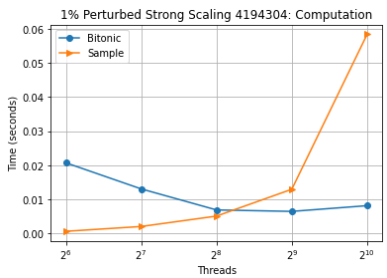
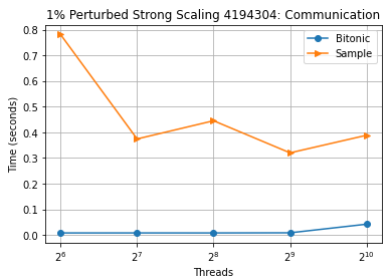
Reverse Sorted

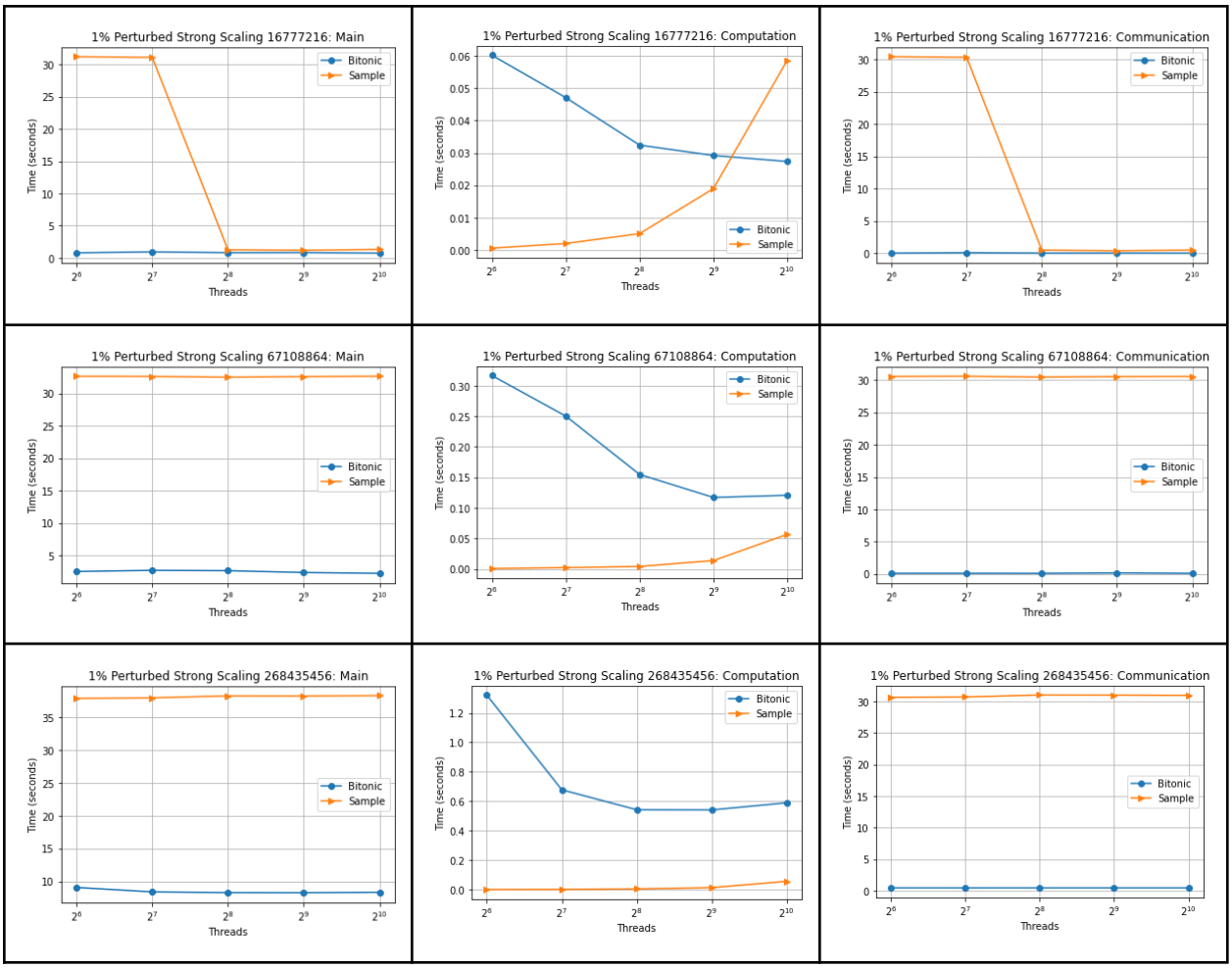
Main	Computation	Communication


Reverse Sorted Strong Scaling 65536: Main
Reverse Sorted Strong Scaling 65536: Computation
Reverse Sorted Strong Scaling 65536: Communication
Reverse Sorted Strong Scaling 262144: Main
Reverse Sorted Strong Scaling 262144: Computation
Reverse Sorted Strong Scaling 262144: Communication
Reverse Sorted Strong Scaling 1048576: Main
Reverse Sorted Strong Scaling 1048576: Computation
Reverse Sorted Strong Scaling 1048576: Communication
Reverse Sorted Strong Scaling 4194304: Main
Reverse Sorted Strong Scaling 4194304: Computation
Reverse Sorted Strong Scaling 4194304: Communication
Reverse Sorted Strong Scaling 16777216: Main
Reverse Sorted Strong Scaling 16777216: Computation
Reverse Sorted Strong Scaling 16777216: Communication
Bitonic
Sample
Time (seconds)
Threads

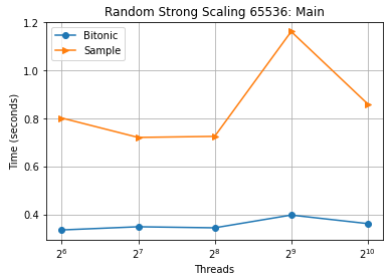
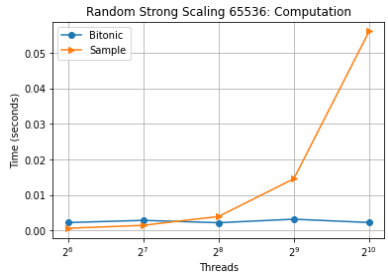
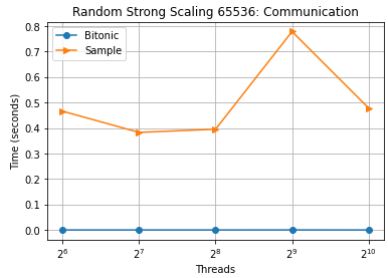
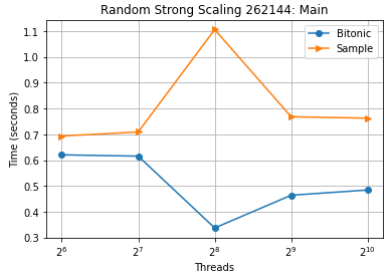
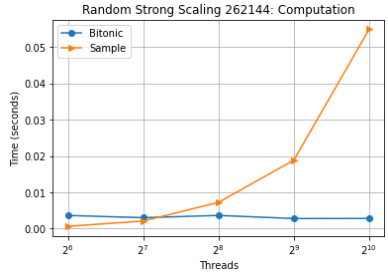
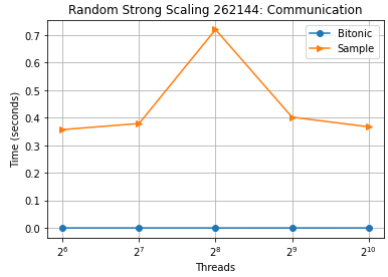
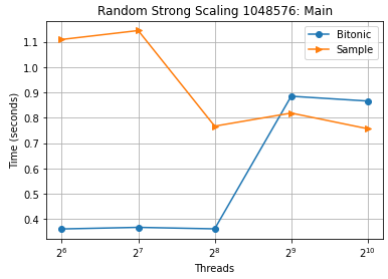
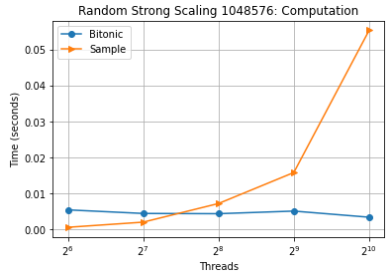
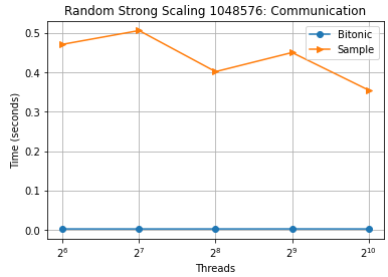
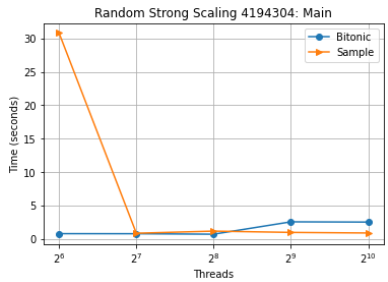
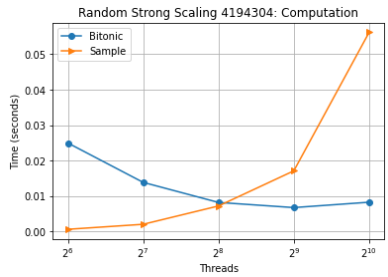
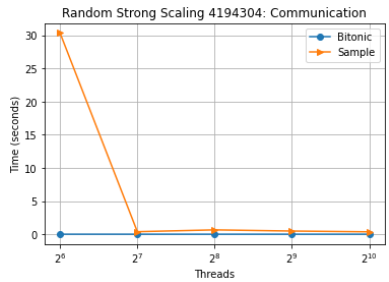

Reverse Sorted Strong Scaling 67108864: Main
Reverse Sorted Strong Scaling 67108864: Computation
Reverse Sorted Strong Scaling 67108864: Communication
Reverse Sorted Strong Scaling 268435456: Main
Reverse Sorted Strong Scaling 268435456: Computation
Reverse Sorted Strong Scaling 268435456: Communication
Time (seconds)
Threads
Bitonic
Sample

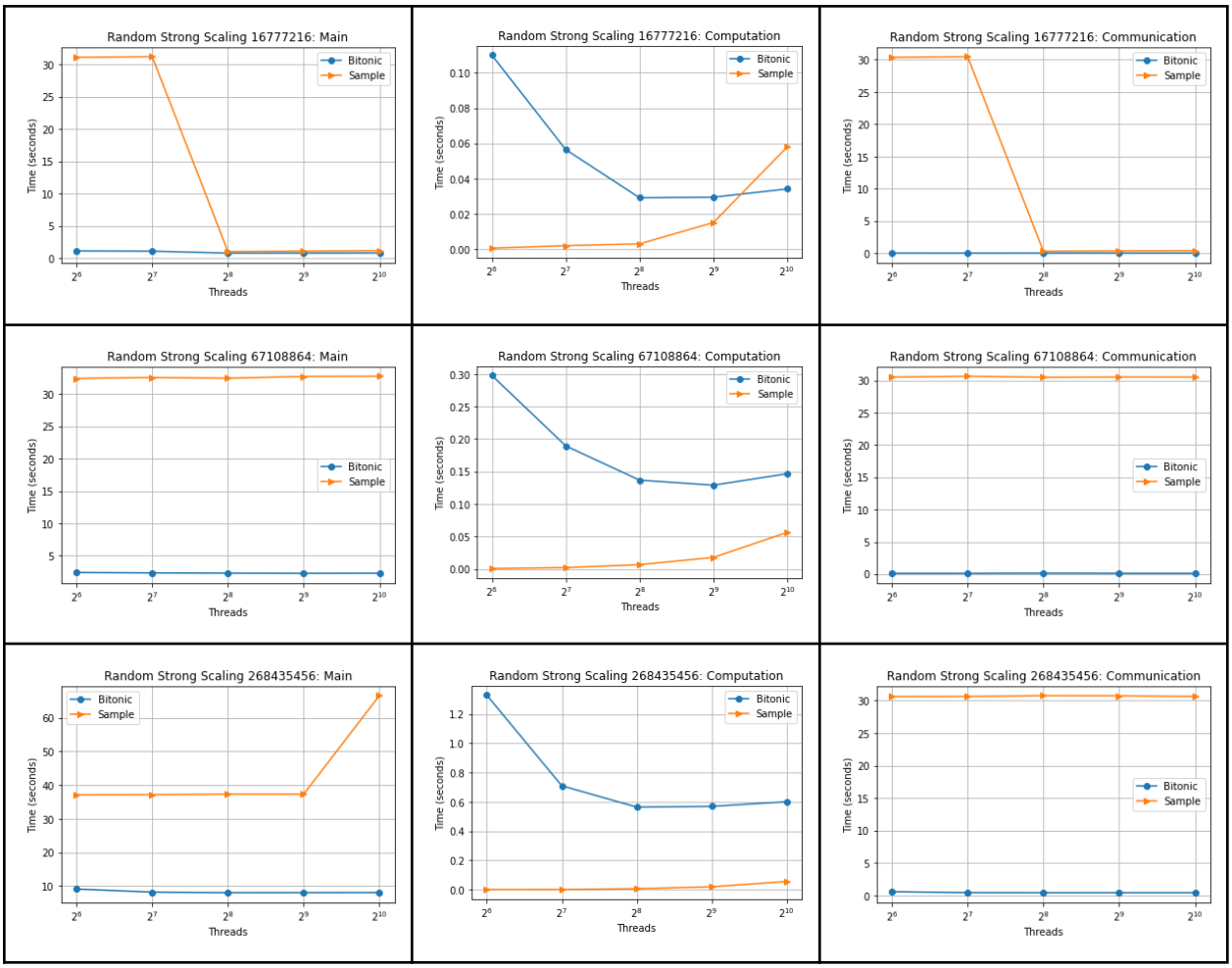
1% Perturbed

Main	Computation	Communication
		
		
		
		


1% Perturbed Strong Scaling 16777216: Main
1% Perturbed Strong Scaling 16777216: Computation
1% Perturbed Strong Scaling 16777216: Communication
1% Perturbed Strong Scaling 67108864: Main
1% Perturbed Strong Scaling 67108864: Computation
1% Perturbed Strong Scaling 67108864: Communication
1% Perturbed Strong Scaling 268435456: Main
1% Perturbed Strong Scaling 268435456: Computation
1% Perturbed Strong Scaling 268435456: Communication
Time (seconds)
Threads
Bitonic
Sample

Random

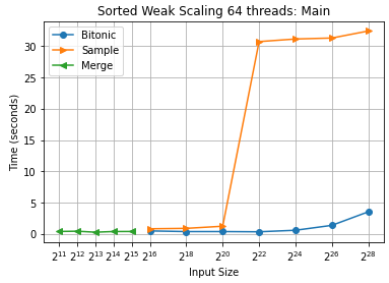
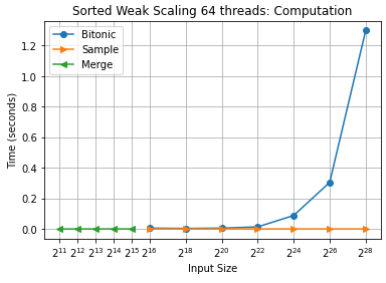
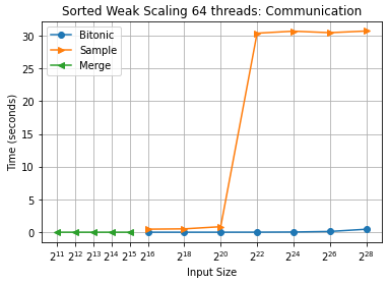
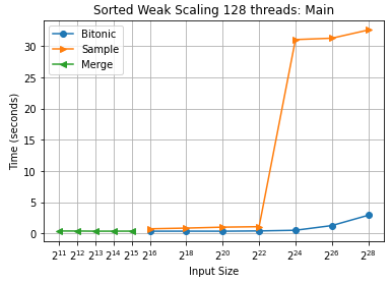
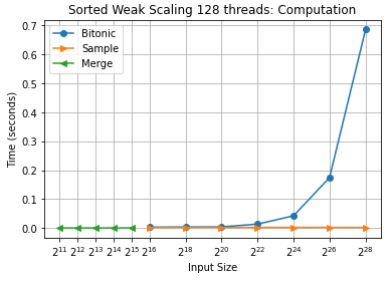
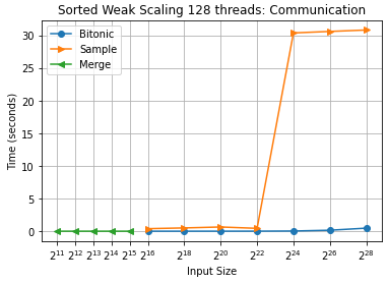
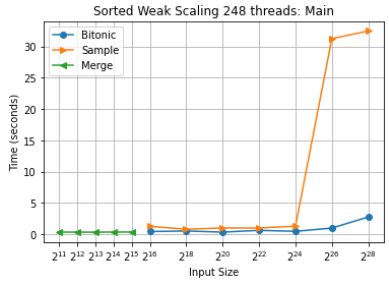
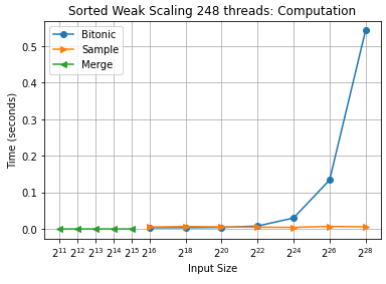
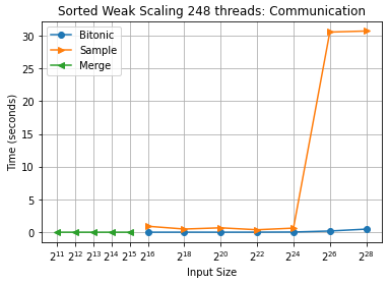
Main	Computation	Communication
		
		
		
		

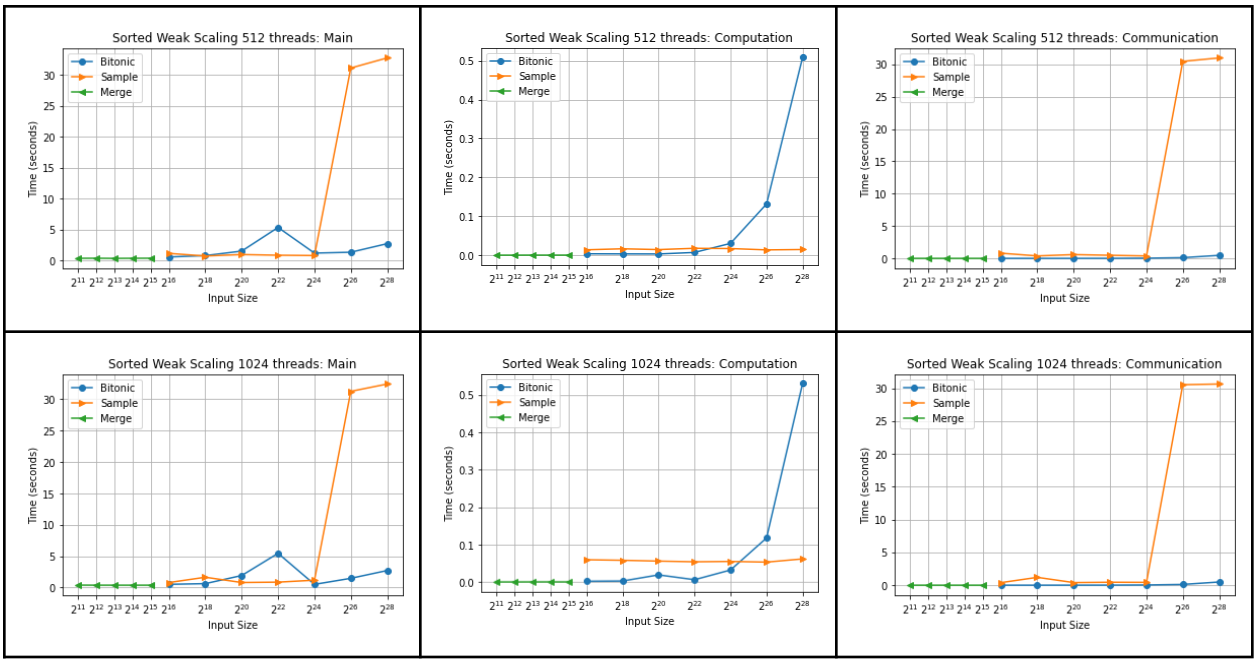

Random Strong Scaling 16777216: Main
Random Strong Scaling 16777216: Computation
Random Strong Scaling 16777216: Communication
Random Strong Scaling 67108864: Main
Random Strong Scaling 67108864: Computation
Random Strong Scaling 67108864: Communication
Random Strong Scaling 268435456: Main
Random Strong Scaling 268435456: Computation
Random Strong Scaling 268435456: Communication
Bitonic
Sample
Time (seconds)
Threads

Weak Scaling

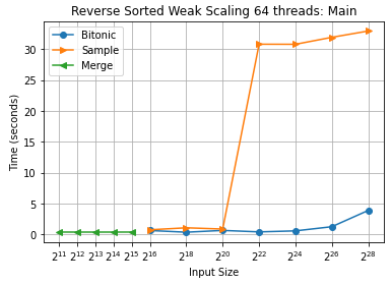
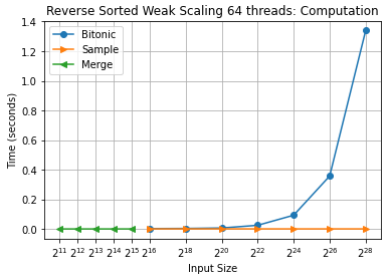
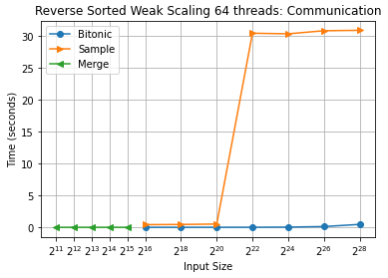
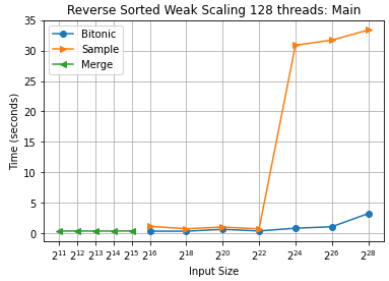
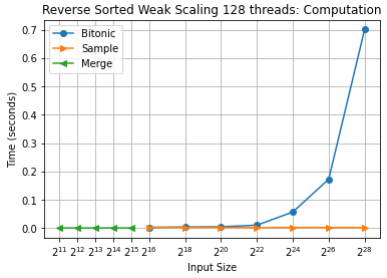
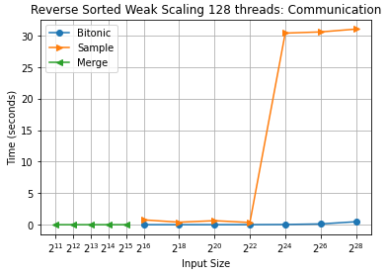
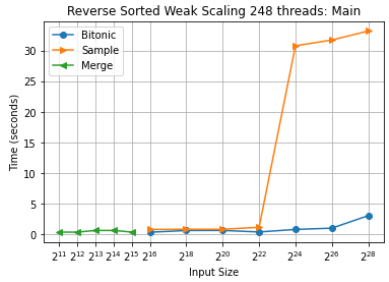
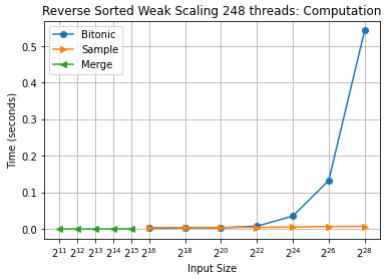
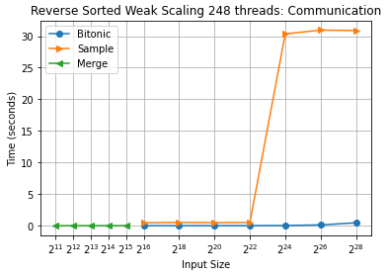
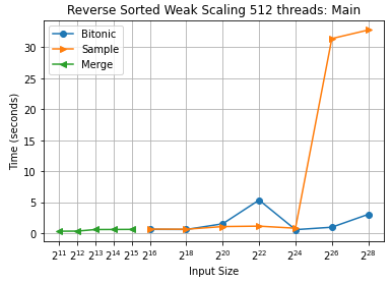
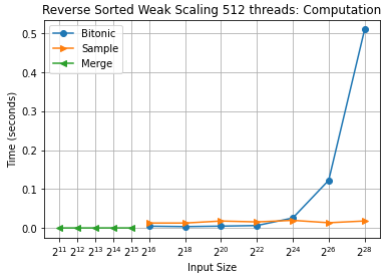
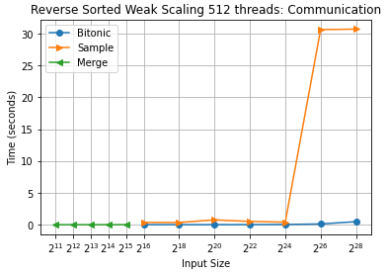
All of the graphs show good weak scaling with generally positive slopes which is expected as the input size increases on the same amount of threads. For Bitonic sort it scaled in the computation while communication stayed relatively stable. Sample sort was the opposite where it stayed stable through the computation (which was really efficient) but skyrocketed up in communication between threads at a given input size. 1% perturbed had the best weak scaling performance overall, with runtimes for sample sort staying low until the last one or two input sizes.

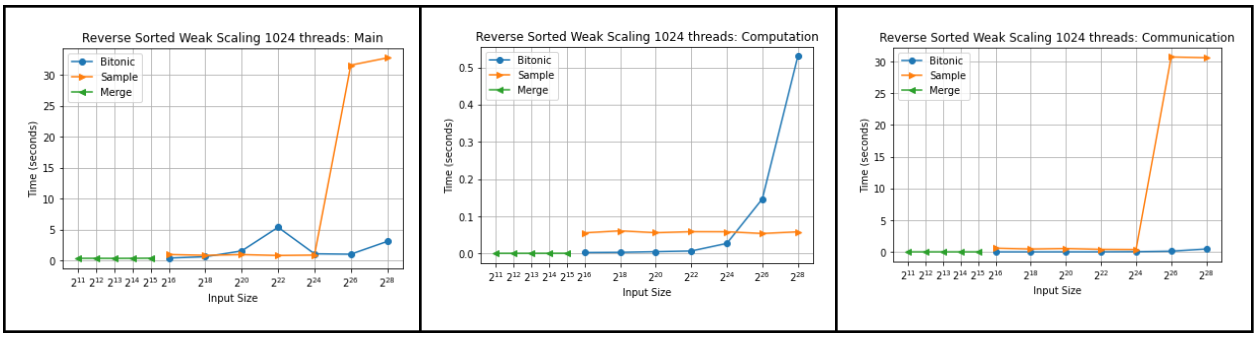
Sorted

Main	Computation	Communication
		
		
		

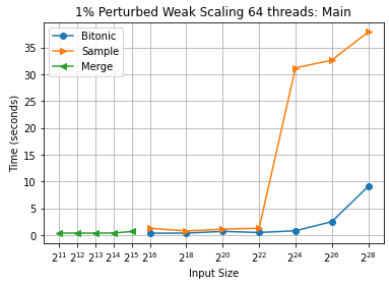
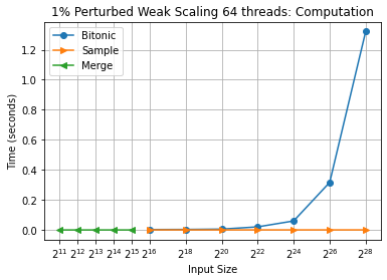
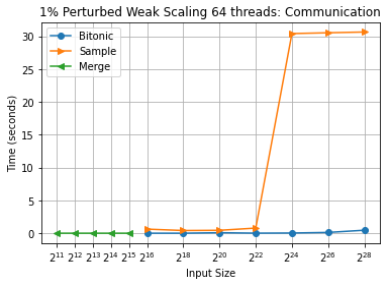
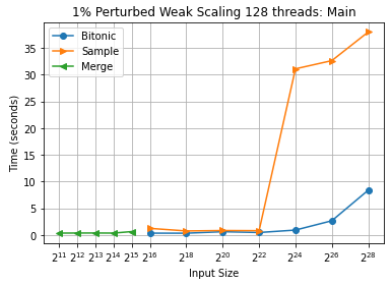
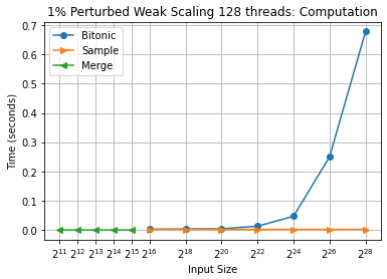
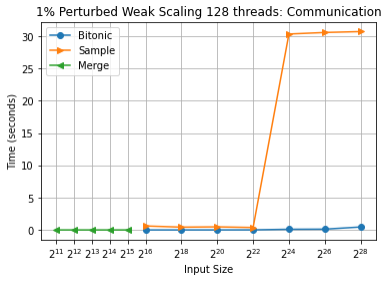
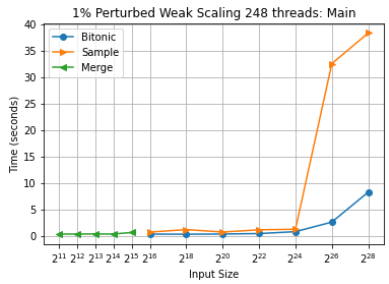
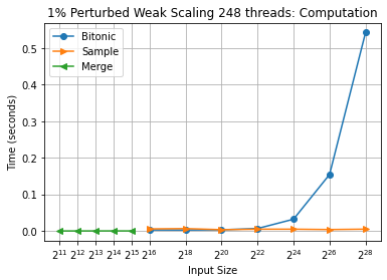
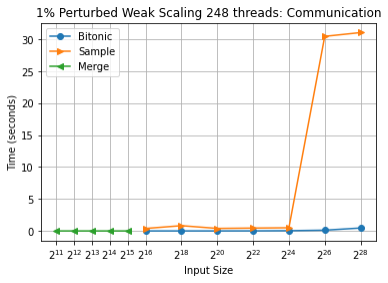
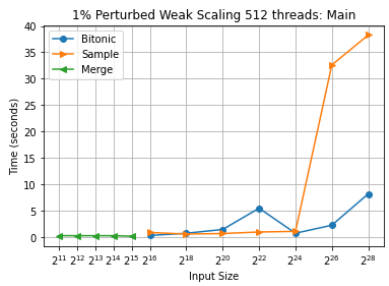
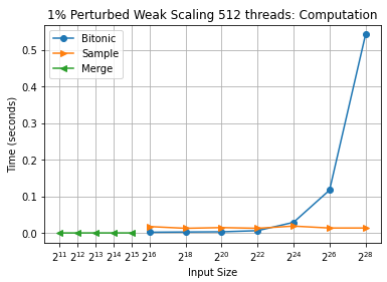
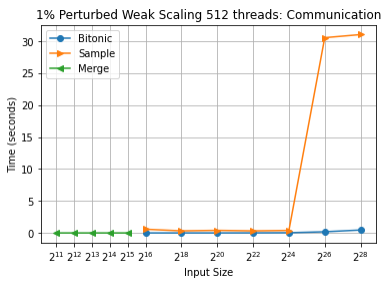

Sorted Weak Scaling 512 threads: Main
Sorted Weak Scaling 512 threads: Computation
Sorted Weak Scaling 512 threads: Communication
Sorted Weak Scaling 1024 threads: Main
Sorted Weak Scaling 1024 threads: Computation
Sorted Weak Scaling 1024 threads: Communication
Bitonic
Sample
Merge
Time (seconds)
Input Size

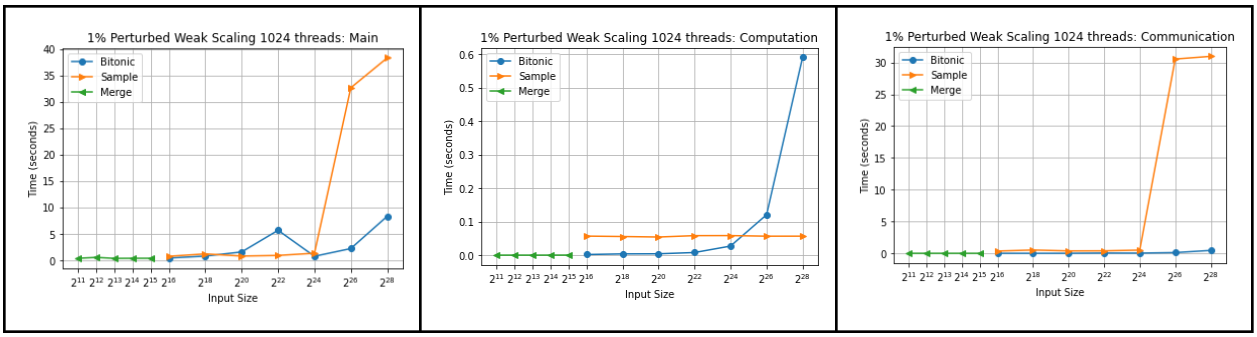
Reverse Sorted

Main	Computation	Communication
		
		
		
		

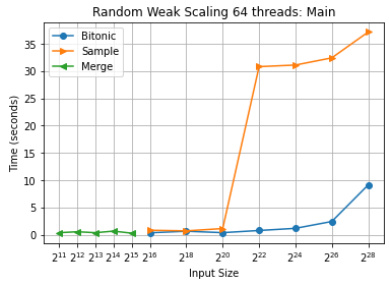
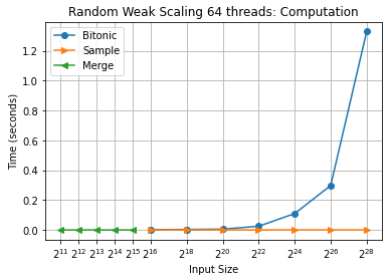
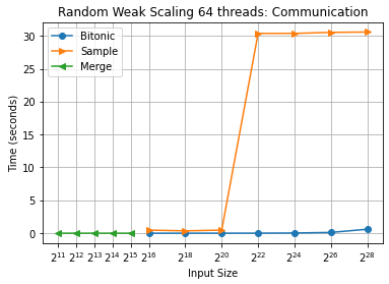
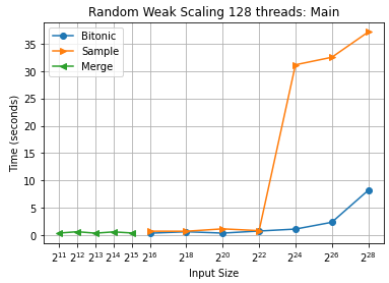
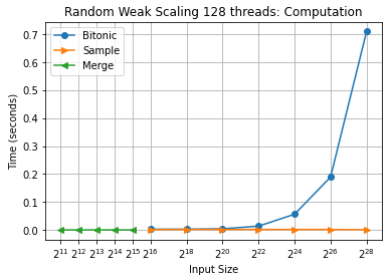
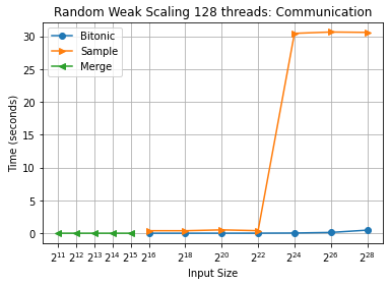
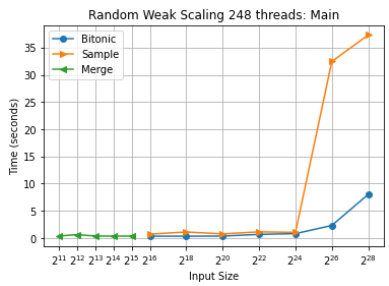
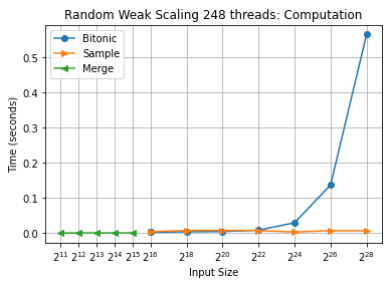
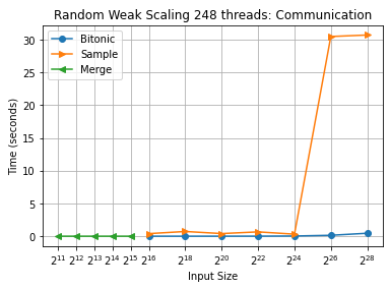
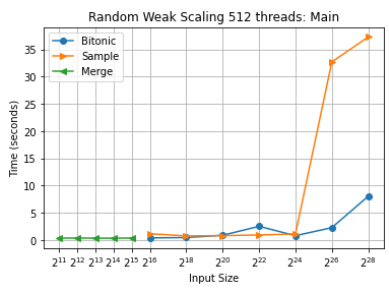
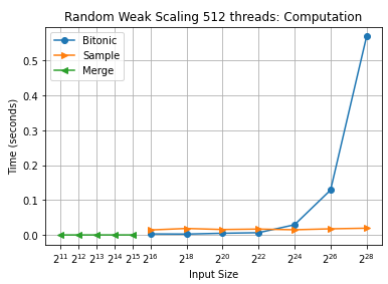
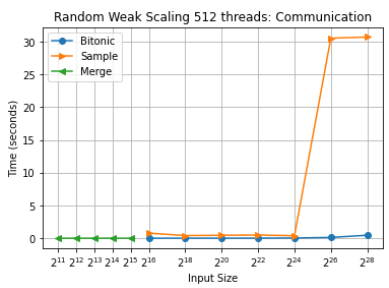

Reverse Sorted Weak Scaling 1024 threads: Main
Bitonic
Sample
Merge
Time (seconds)
Input Size
Reverse Sorted Weak Scaling 1024 threads: Computation
Bitonic
Sample
Merge
Time (seconds)
Input Size
Reverse Sorted Weak Scaling 1024 threads: Communication
Bitonic
Sample
Merge
Time (seconds)
Input Size

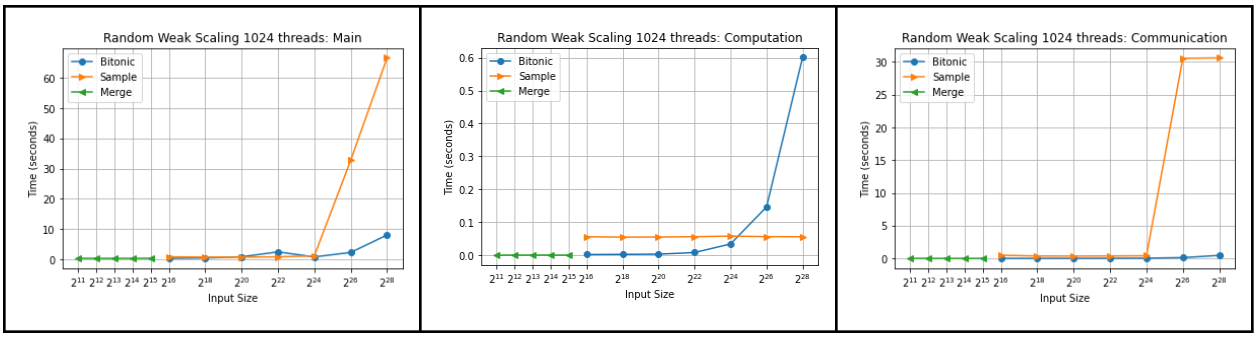
1% Perturbed

Main	Computation	Communication
		
		
		
		


1% Perturbed Weak Scaling 1024 threads: Main
Bitonic
Sample
Merge
Time (seconds)
Input Size
1% Perturbed Weak Scaling 1024 threads: Computation
Bitonic
Sample
Merge
Time (seconds)
Input Size
1% Perturbed Weak Scaling 1024 threads: Communication
Bitonic
Sample
Merge
Time (seconds)
Input Size

Random

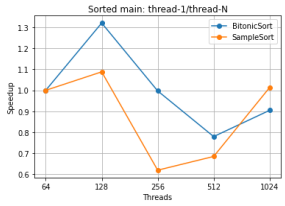
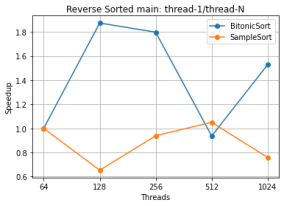
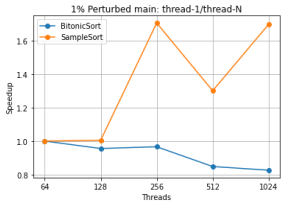
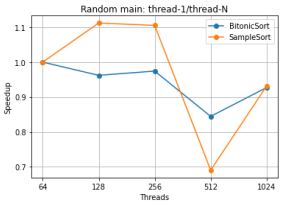
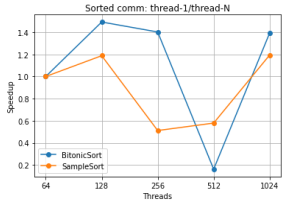
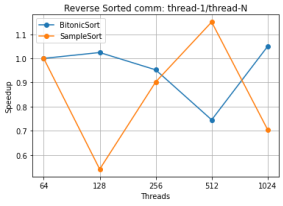
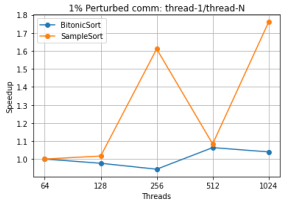
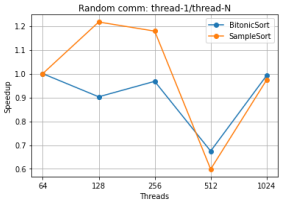
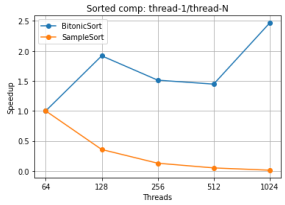
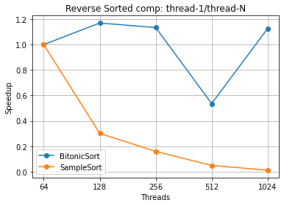
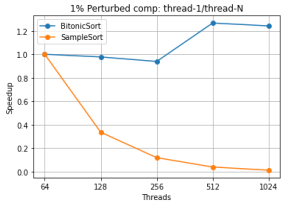
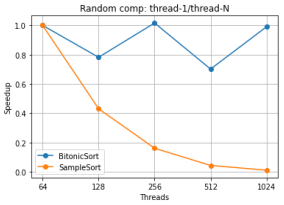
Main	Computation	Communication
		
		
		
		



Speedup Graphs

Cuda

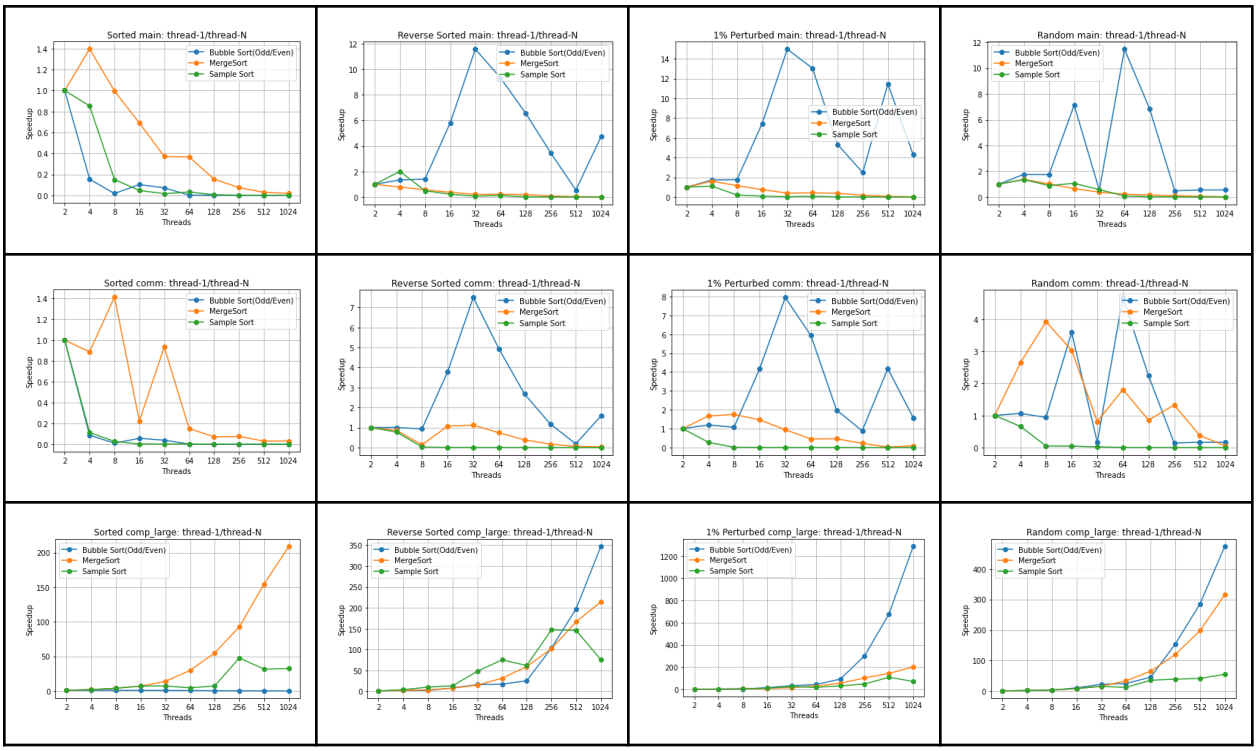
The graphs below show a comparison between the speedups for the parallelized CUDA implementations of bitonic sort and sample sort. A CUDA implementation of merge sort was also created, but it is not included in the following comparison graphs, as it was tested on lower input sizes, so direct comparisons cannot be made. When looking at the graphs below, it appears that for most input types and thread sizes, bitonic sort had greater speedups than sample sort. However, there are some exceptions to this trend, with sample sort having greater speedups than bitonic sort for most thread sizes for 1% perturbed and random input types. Interestingly, while speedups consistently trend downward with increased thread sizes for the sample sort on the comm graphs (due to increased communication overhead with more parallelization), this is not the case with bitonic sort, which has some communication speedups greater than 1.

Sorted	Reverse Sorted	1% Perturbed	Random
			
			
			

MPI

Through all input types, the main function of merge and sample sort has little to no speedup. The slopes for these implementations are flat, even a little negative. This indicates that the overall performance did not show large improvements as more threads were added. Bubble sort is an exception to this. It shows a high variance in speedup, with some high spikes and some low dips. For each implementation, these overall performance trends are caused by commu nication overhead it seems. Looking at the the communication graphs for each implementation, it can be seen that the shape mirrors the “main” graph. This is because each implementation’s communication times are a lot higher than computation, resulting in a minimal ability to see the effects of computation on the “main” graphs. This is further supported by the computation graphs on the bottom row. These actually look pretty optimal. As more threads are added, speedup increases, meaning the algorithm is parallelizing efficiently. Bubble sort appears to have the largest increase in speedup for this section, however, when taking the disproportionately large run time for bubble sort into account, it still makes it less desirable than other implementations. One other thing to note before finishing is that bubble sort looks very different on sorted data. It has virtually no speedup, but actually performs very well in terms of run time.

Sorted	Reverse Sorted	1% Perturbed	Random


Sorted main: thread-1/thread-N
Reverse Sorted main: thread-1/thread-N
1% Perturbed main: thread-1/thread-N
Random main: thread-1/thread-N
Sorted comm: thread-1/thread-N
Reverse Sorted comm: thread-1/thread-N
1% Perturbed comm: thread-1/thread-N
Random comm: thread-1/thread-N
Sorted comp_large: thread-1/thread-N
Reverse Sorted comp_large: thread-1/thread-N
1% Perturbed comp_large: thread-1/thread-N
Random comp_large: thread-1/thread-N
Bubble Sort(Odd/Even)
MergeSort
Sample Sort
Speedup
Threads

Individual Algorithms

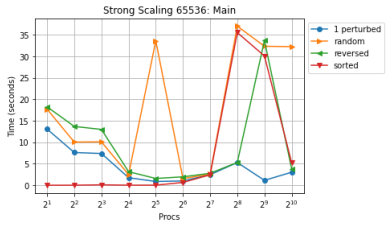
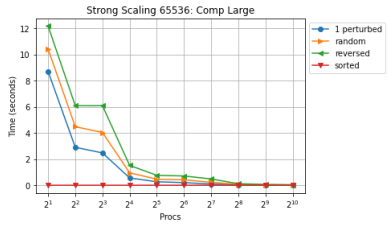
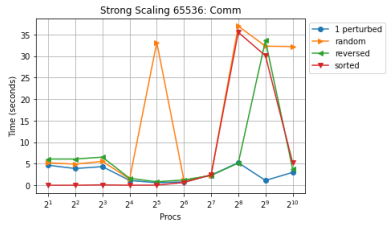
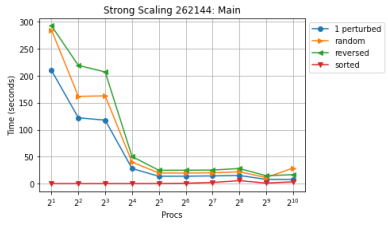
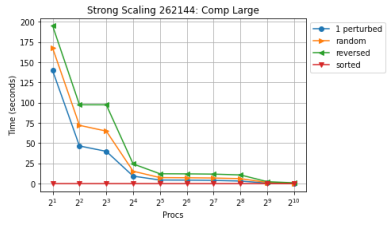
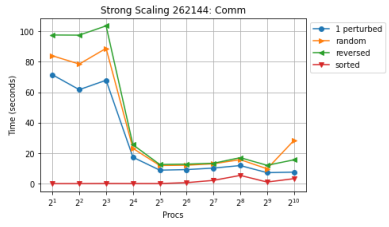
MPI

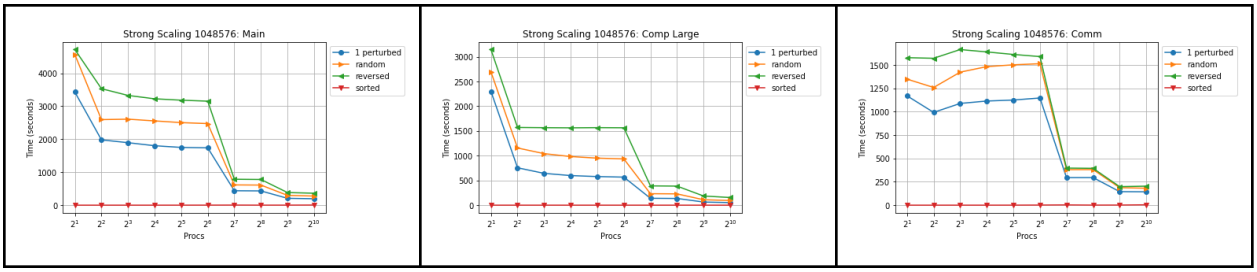
Bubble Sort

Strong Scaling

Bubble sort was implemented using an odd even sort where each process performs sequential bubble sort on its segment of data, then the data is passed around between processes to continue sorting it from start to finish

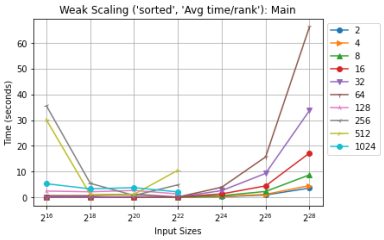
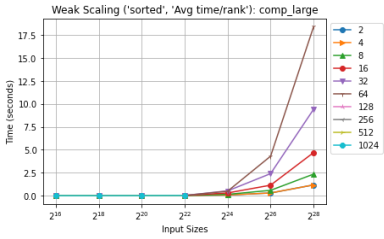
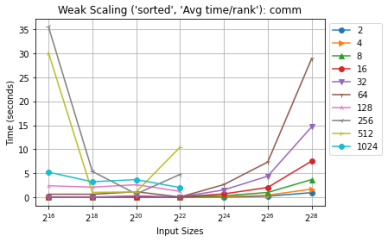
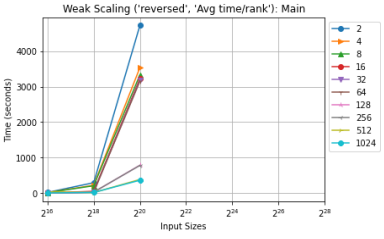
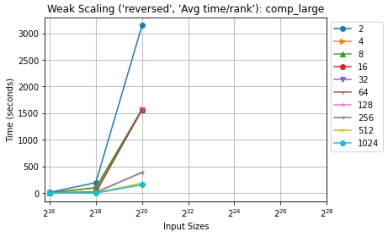
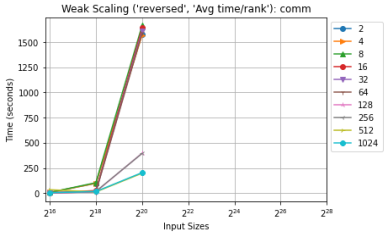
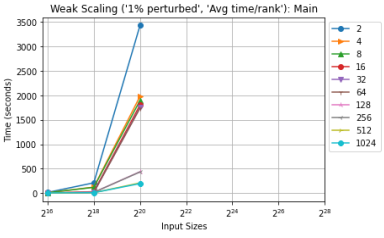
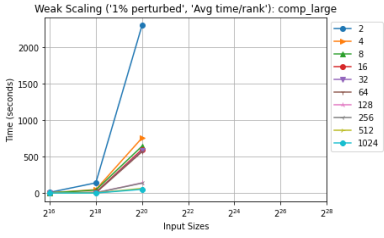
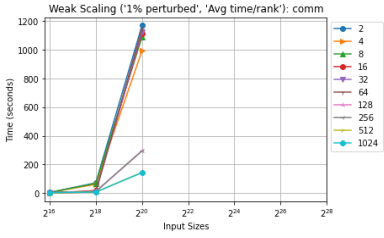
Bubble sort Strong scaled really well in computation as it was given extra processes to run on the overall runtime went down substantially. It was originally expected that it would not Strong scale well for communication, but after seeing the results and thinking about it a little deeper this does make sense. At a very low number of processes each process has to pass large amounts of data whenever there is a communication step, this leads to a long communication. As can be seen in the graphs, increasing the number of processes eventually finds a point where the communication cost is now bottlenecked by needing to communicate and not the time it takes to transfer so many data points.

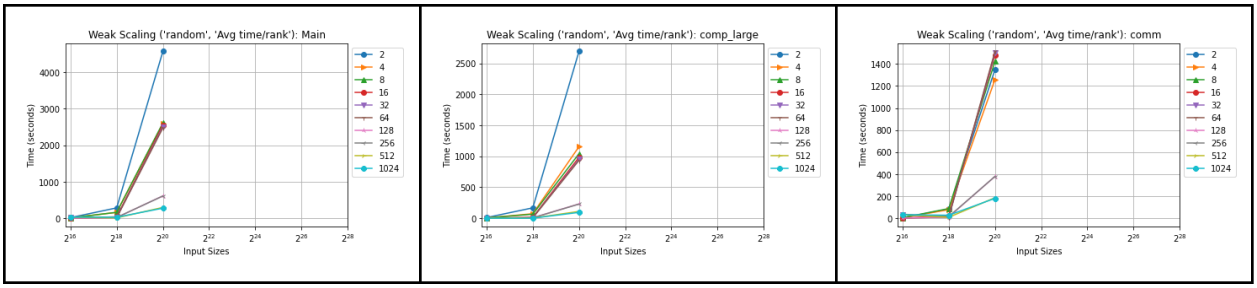
Main	Computation	Communication
		
		



Weak Scaling

As can be seen, bubble sort weak scaled quite poorly. I believe this is due to the inherent sequential algorithm being used to perform this sort. As the amount of data increases the algorithm can simply not keep up by any means since the O(n^2) runtime it uses sequentially.

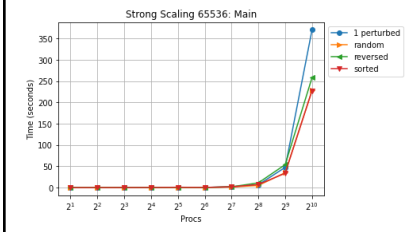
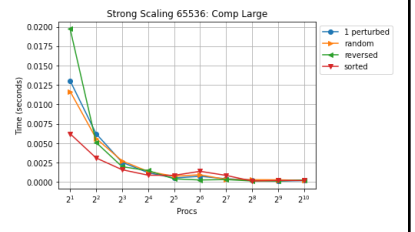
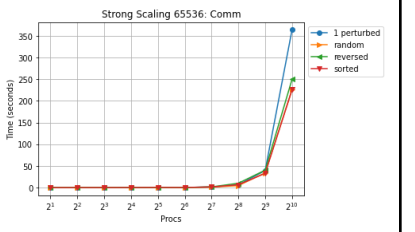
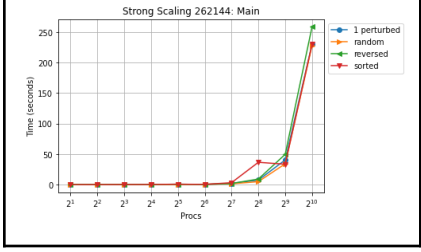
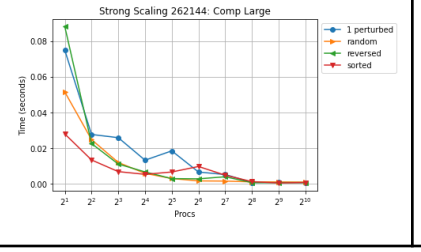
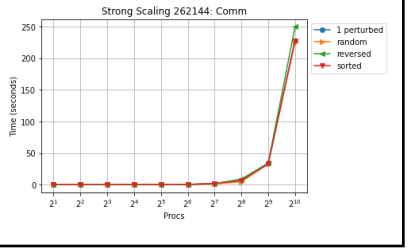
Main	Computation	Communication
		
		
		

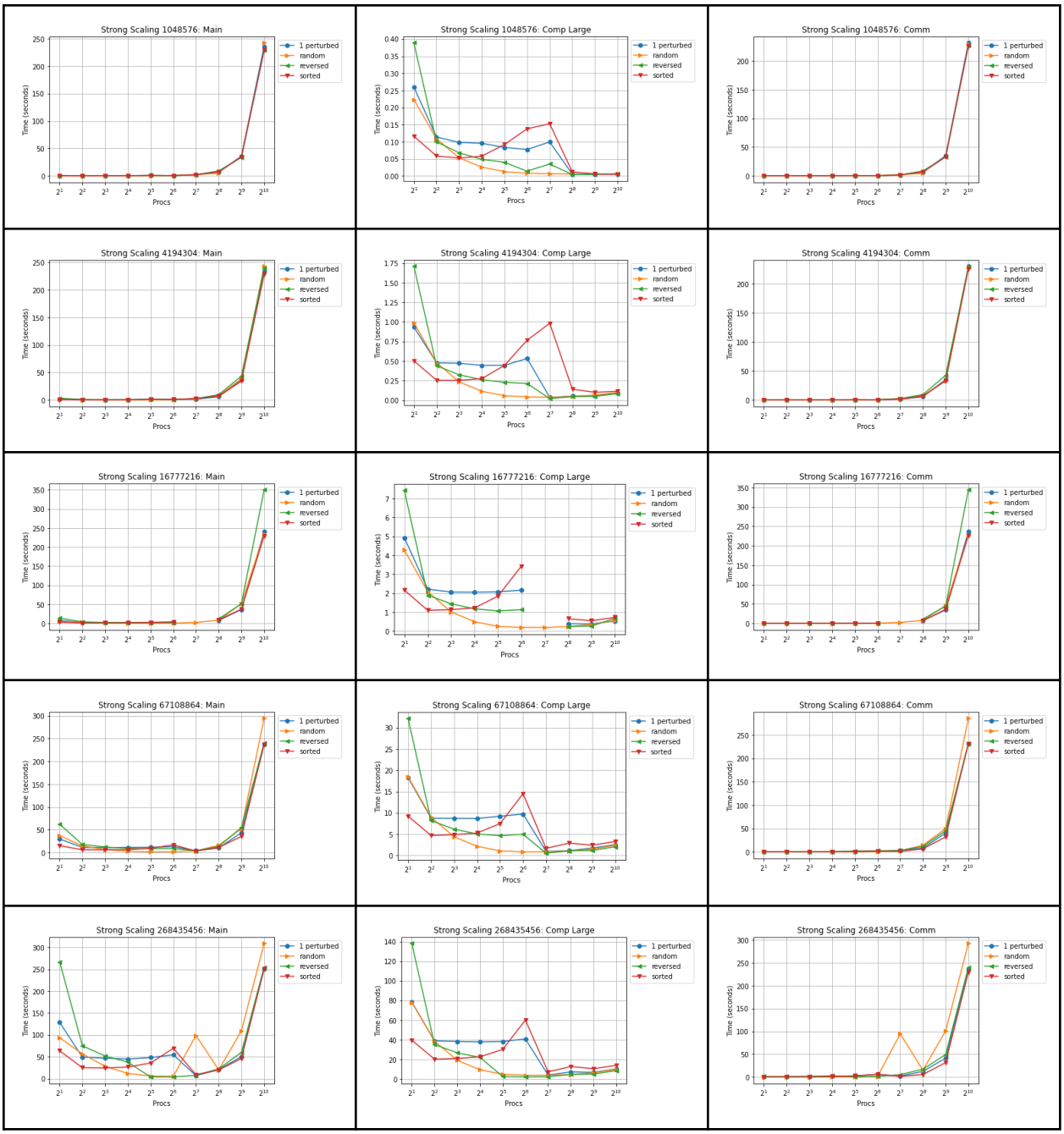


Sample Sort

Strong Scaling

The first thing to note is that regardless of the input type, sample sort performed very similarly on the data, so the trends will be discussed primarily as a whole. Through all input sizes and input types the graphs make it clear that there is a very large overhead in communication which drives the shape of the “main” graphs. The communication graphs look similar regardless of the input size. Staying mostly flat until 2^8 processes, at which the run time skyrockets. This could be because I used more nodes, which can sometimes create a larger communication overhead as the nodes have to communicate farther. The computation graphs also look similar, with a downward slope, performing better as processes are added. This is ideal scaling, but does not outweigh the extremely high communication times. All this being said, in the main graphs, you cannot see the benefit of adding threads at all until the largest two input sizes. Then, computation parallelization benefits carry more weight and there is a slightly negative slope at the beginning of those graphs.

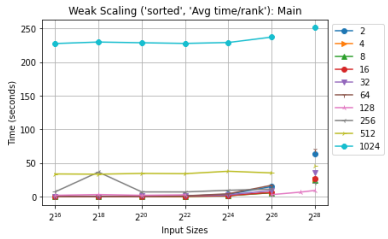
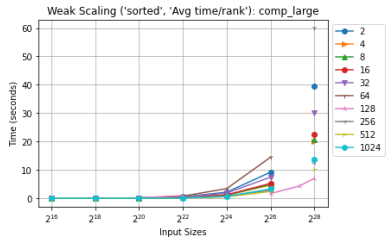
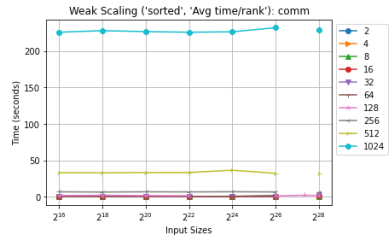
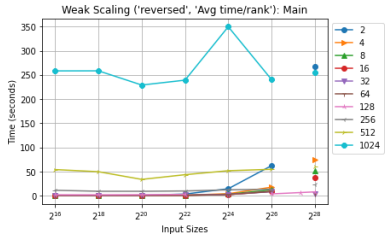
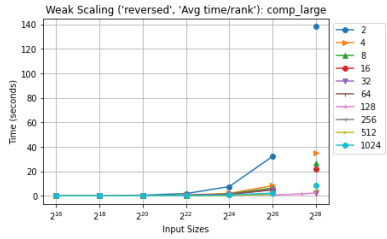
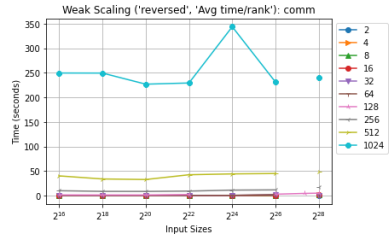
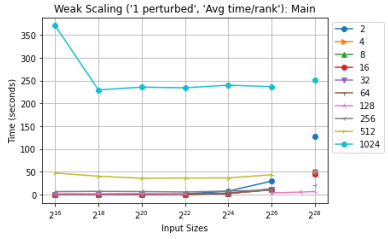
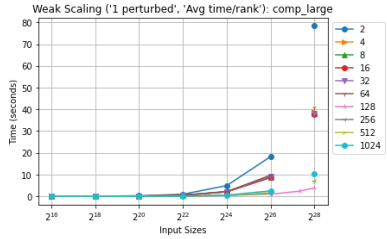
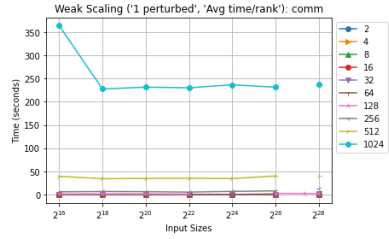
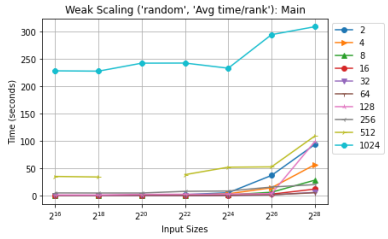
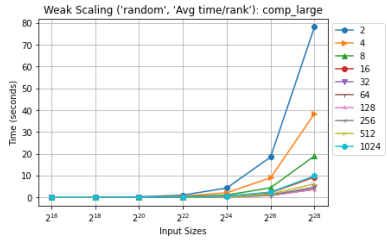
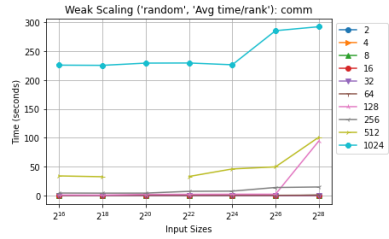
Main	Computation	Communication
		
		



Weak Scaling

In the weak scaling graphs we see a few main trends. In the main and communication graphs that communication overhead is still evident, as runtime is dramatically higher for a higher number of threads (512 & 1024) to be precise. For the ost part, these graphs tend to stay pretty flat, meaning the time stays in the same range for each input size when the same number of processes are used. Booking at the computation graphs we can see that time gets higher as the input size increases, which is to be expected. Additionally, we see that a higher number of threads results in a lower run time as the input size grows. Some takeaways from these graphs

are that the sweet spot for each input type seems to be about 128 processes. Additionally, communication is not implemented in a way that supports lots of threads.

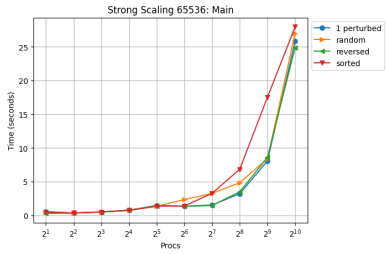
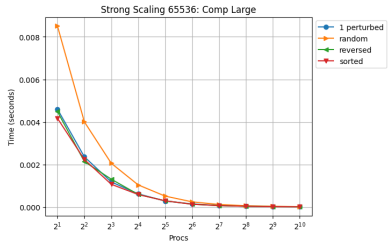
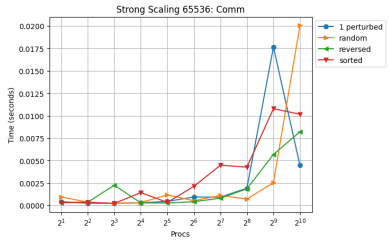
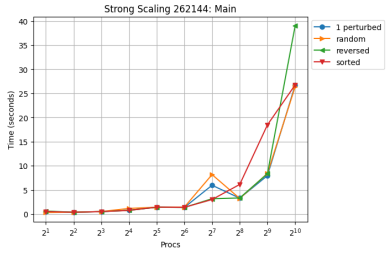
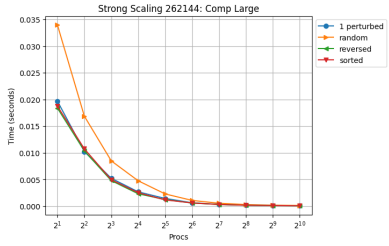
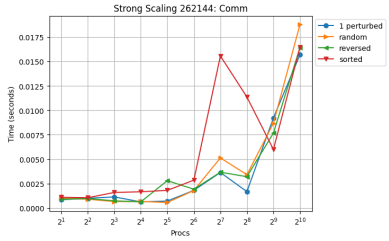
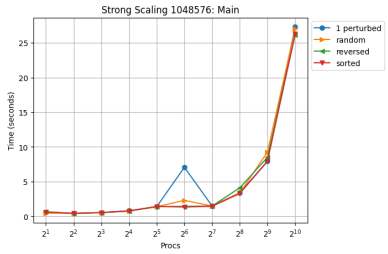
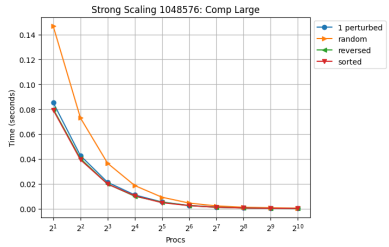
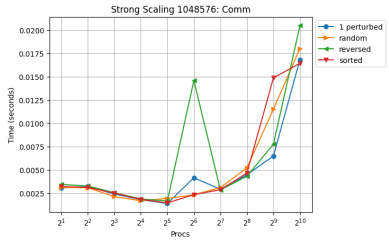
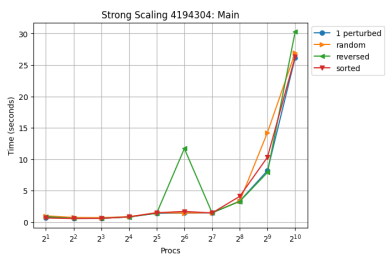
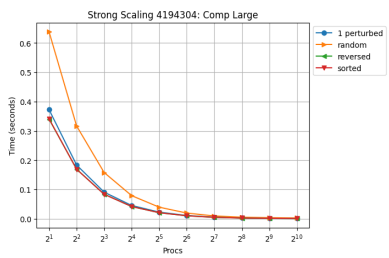
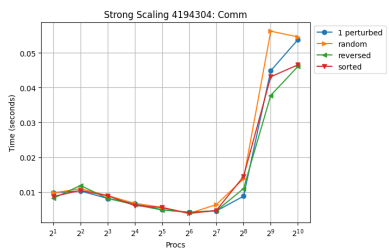
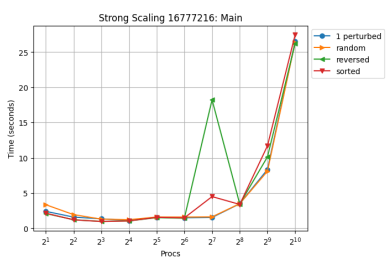
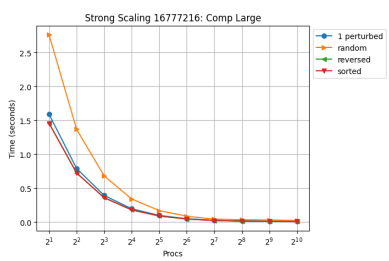
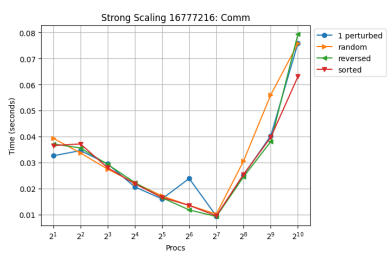
Main	Computation	Communication
		
		
		
		

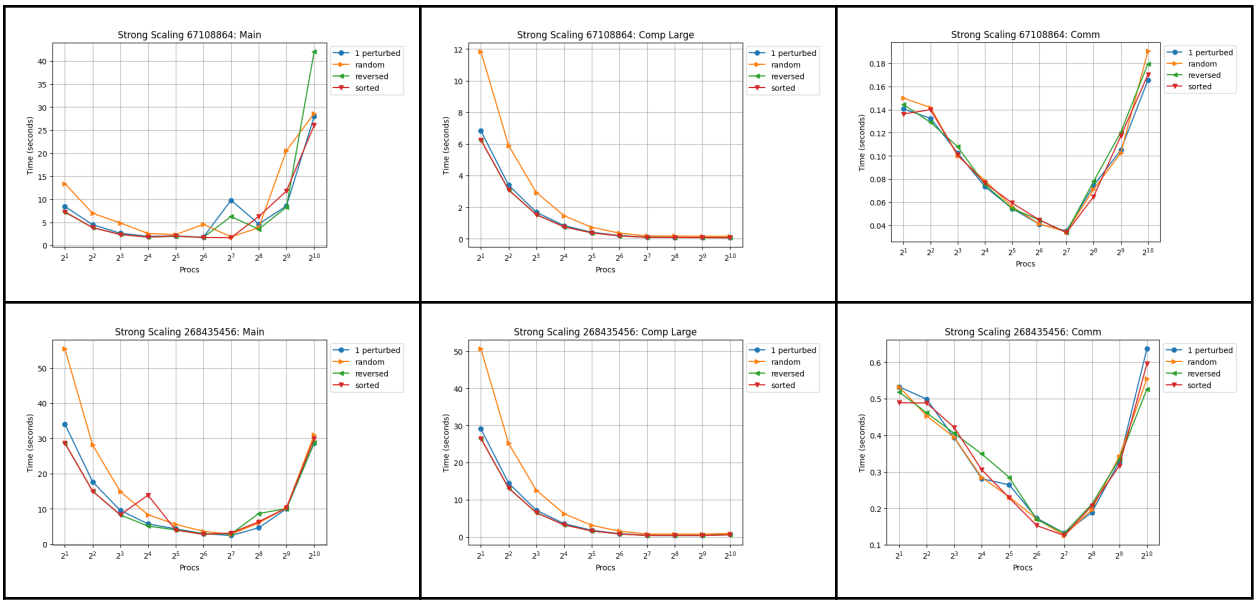
Merge Sort

Strong Scaling

In general, when looking at the plots below, a repeated trend of constant then increasing runtimes for the main and communication regions and constant then decreasing runtimes for the computation region can be observed. It appears that the increased overhead for communication and other non-computational tasks as more processes are added outweighs the impacts of quicker computation on overall runtime once there are roughly 2^7 or 2^8 processes. For the two

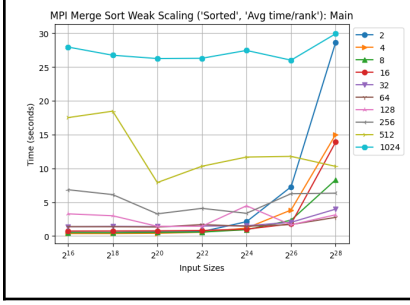
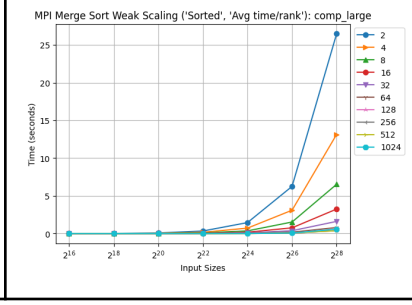
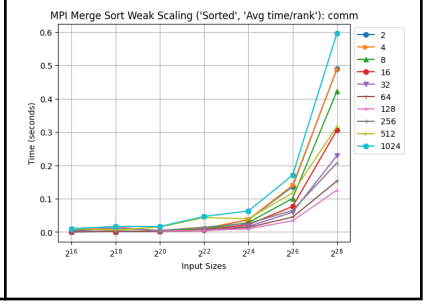
largest input sizes tested, the main plot shows an initially decreasing, then increasing curve; from this, it can be gathered that for the largest input sizes there is more of a clear benefit of adding more parallelization until the point that the additional overhead outweighs this benefit.

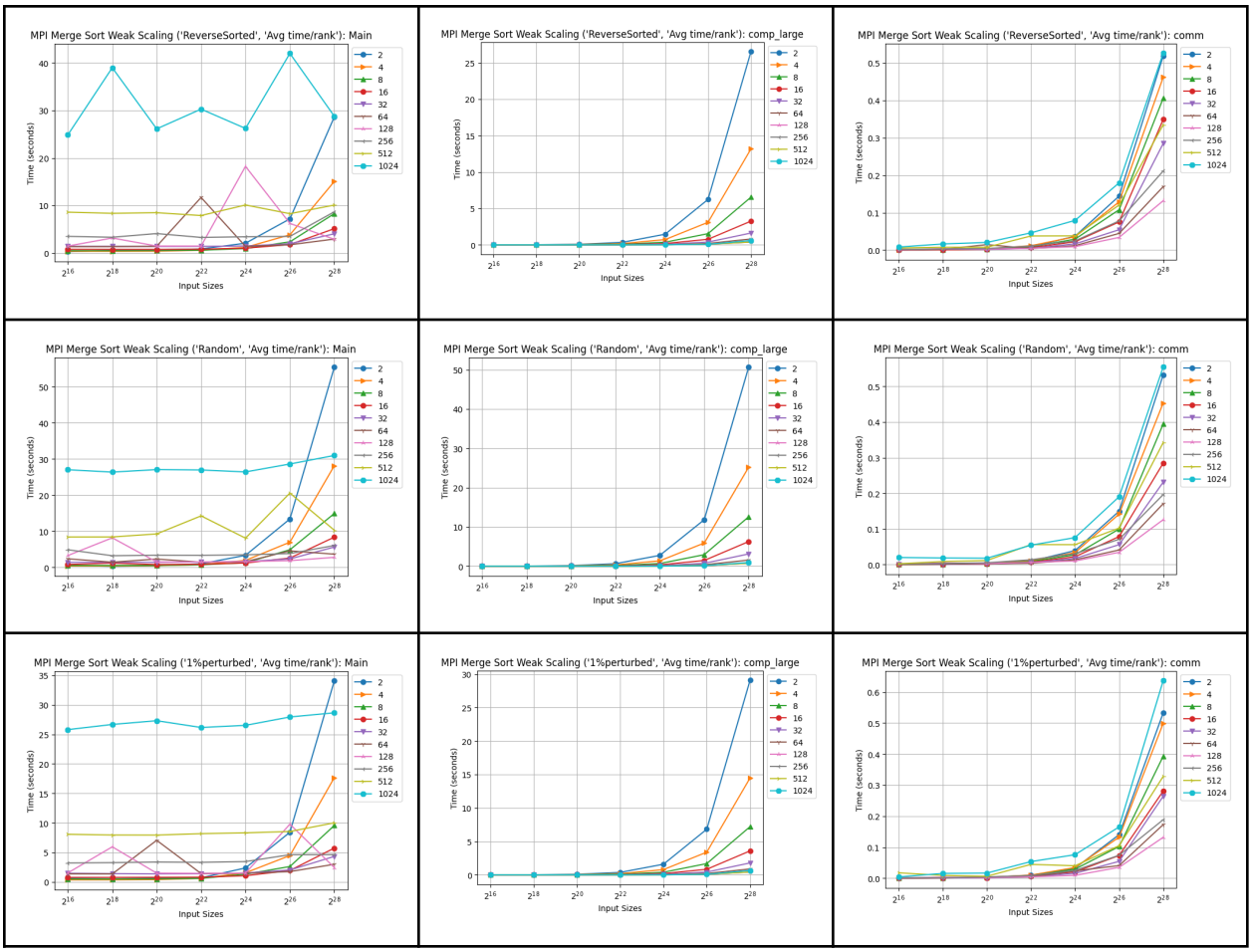
Main	Computation	Communication
		
		
		
		
		



Weak Scaling

As expected, the graphs below show that there are significant increases in runtime accompanying exponential increases in input sizes, especially visible for the largest sizes. However, before this point, the lines representing the runtimes are fairly flat, meaning that the added parallelization likely helped. For the computation graphs, less processes generally meant longer runtimes, while for the communication graphs, the opposite was true, as expected. Across the main graphs below, the largest and smallest numbers of processes generally performed worse than those in the middle; this is likely where the benefits and disadvantages of parallelization are better balanced.

Main	Computation	Communication
		

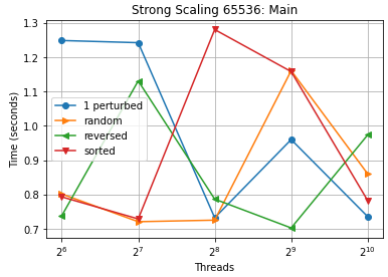
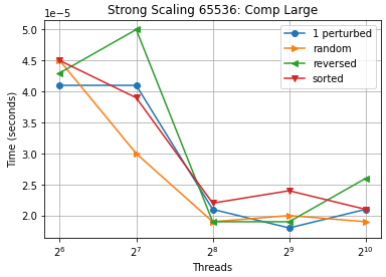
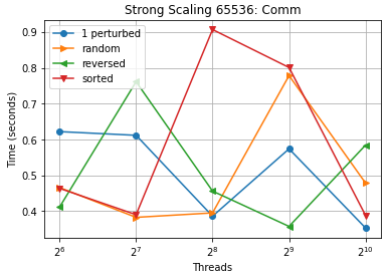
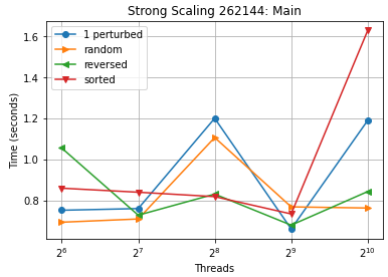
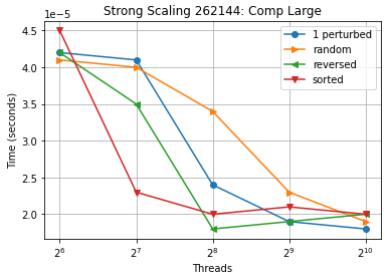
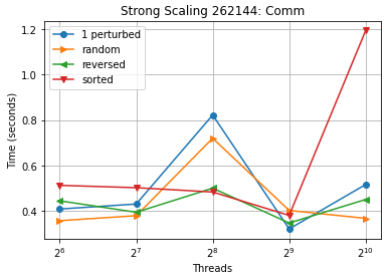
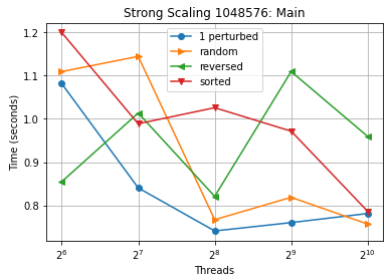
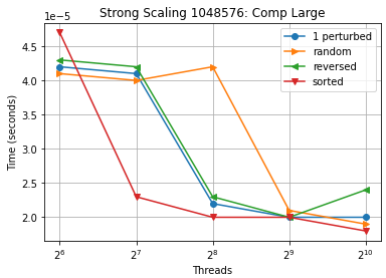
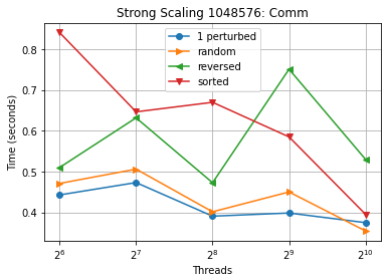
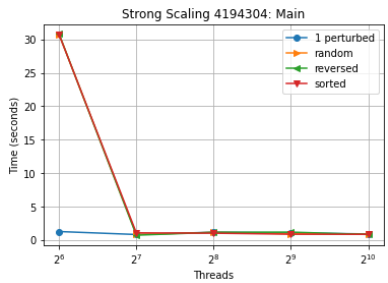
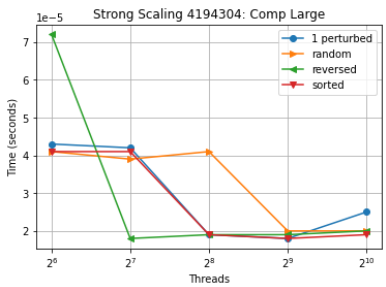
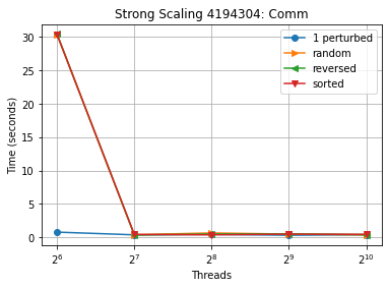
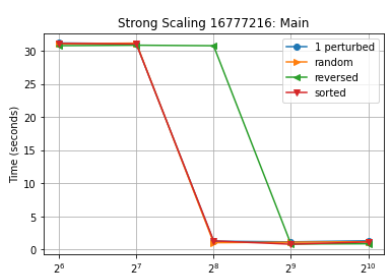
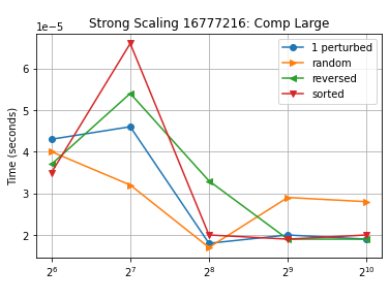
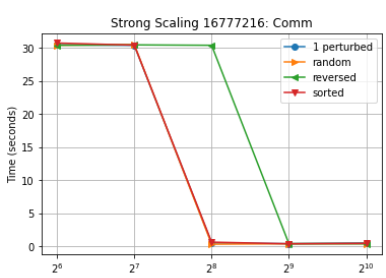


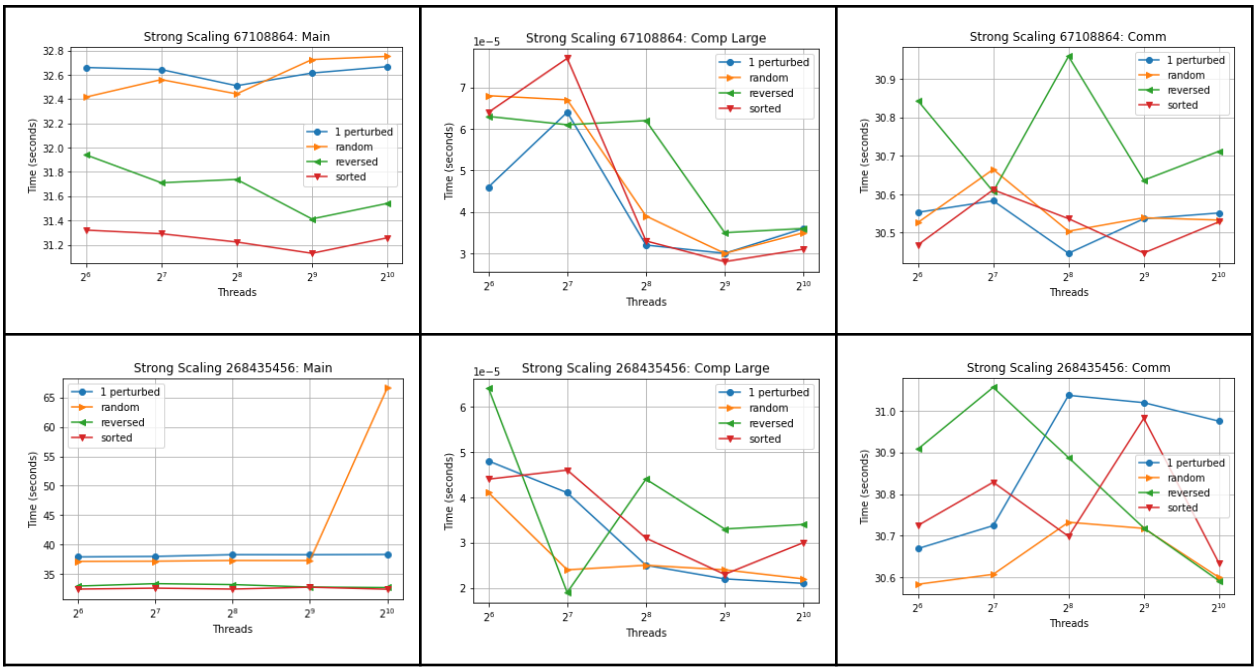
CUDA

Sample Sort

Strong Scaling

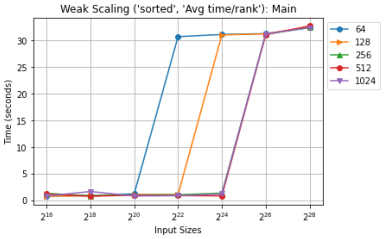
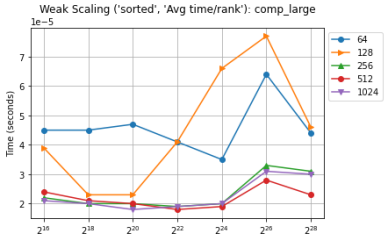
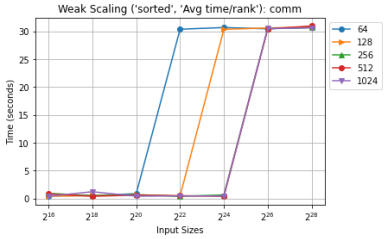
For this implementation, we see slightly inconsistent results. For the first 5 input sizes, the results are driven by the communication graphs so I will mostly be talking about just the computation and communication for those. Firstly, regardless of the input size, as more processes are added, run time goes down for computation. On the other hand, the trends in communication are a lot less consistent. For the middle input sizes,run time dramatically decreases at about 2^8 threads. However, for most other sizes, the lines are either flat or highly variable. The only consistent trend is that these communication times are super high, which diminishes the benefits of parallelizing the process.

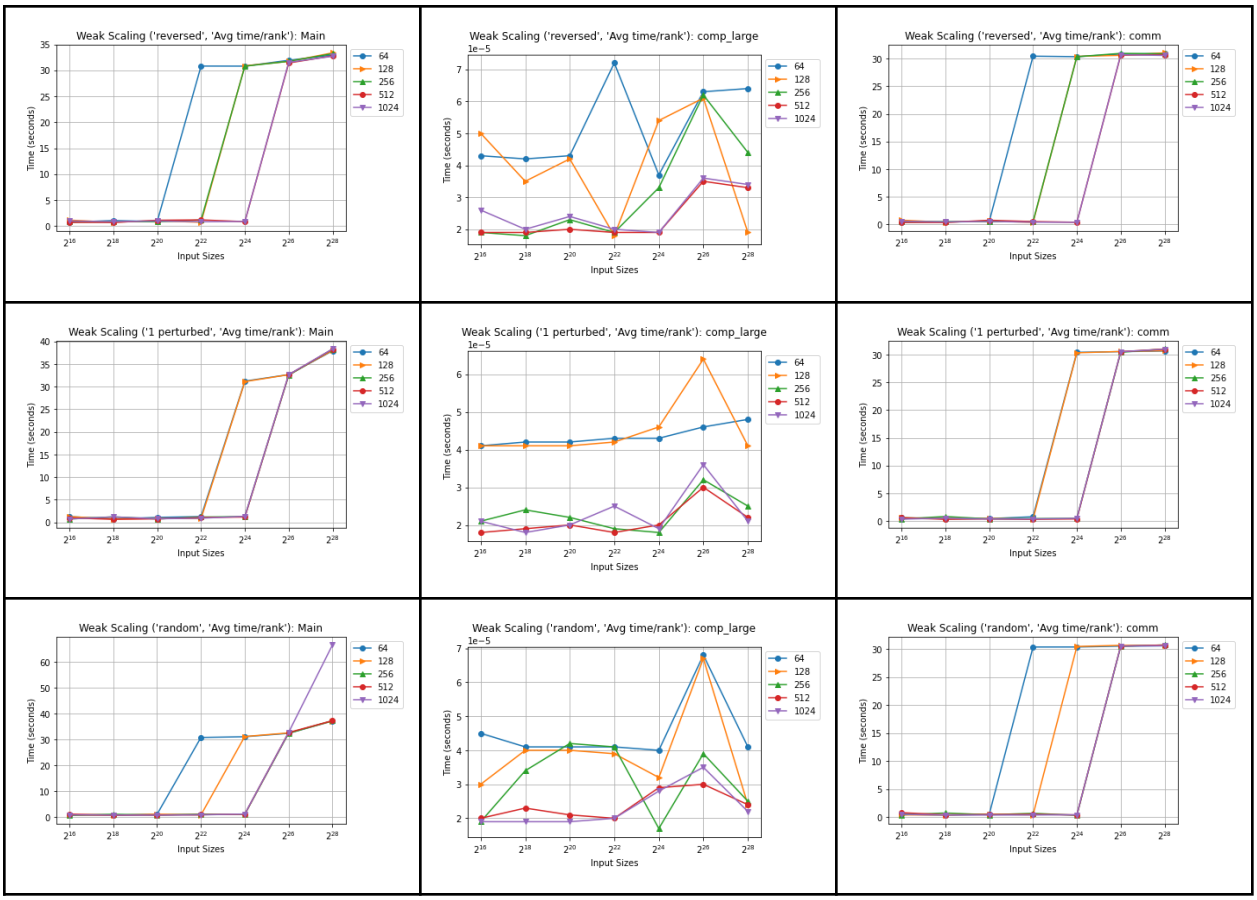
Main	Computation	Communication
		
		
		
		
		



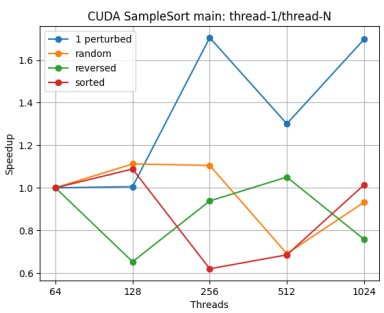
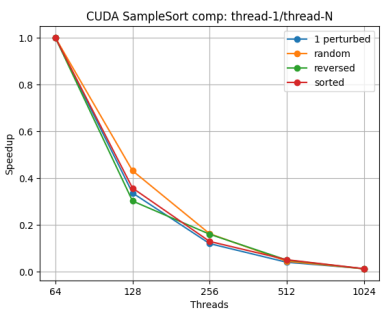
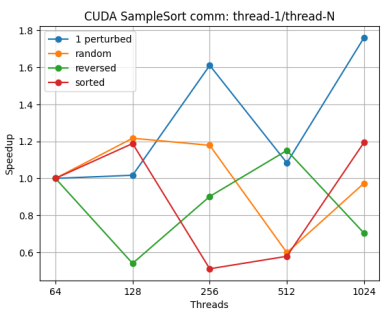
Weak Scaling

The weak scaling trends are consistent across all input types. Again, communication times are high and drive the overall shape of the main graph. For communication, there's a dramatic increase in runtime as a certain input size is hit. Overall, the more threads used, the higher input size that can be handled. For computation, more threads also meant better performance, which is definitely a good trend. This pretty easily translates to the overall conclusion that more threads is better, with the ideal amount being 512 or 1024 (very similar results for both). However, the extreme jump in communication indicates a slight inefficiency in memory usage, as the communication section is comprised of memcpy.

Main	Computation	Communication
		



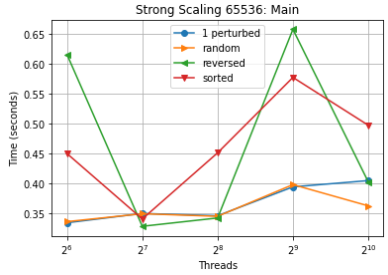
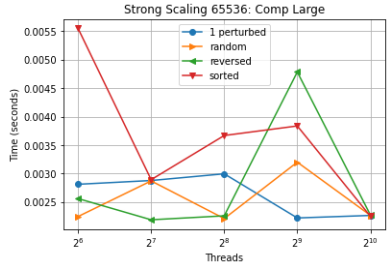
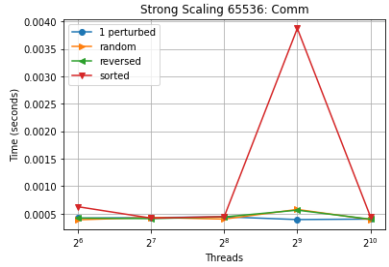
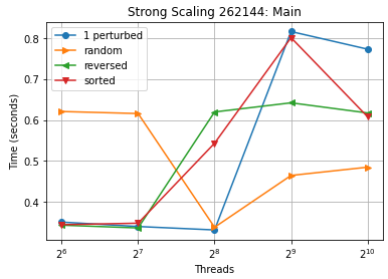
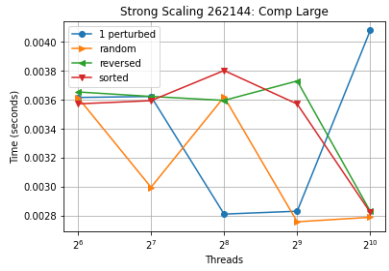
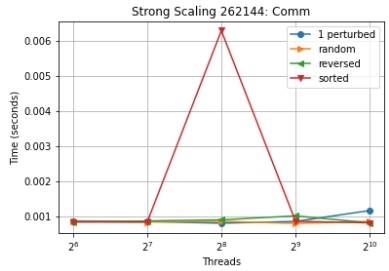
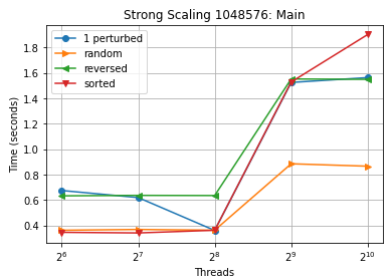
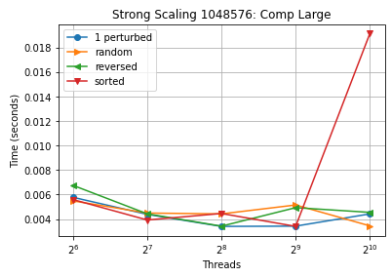
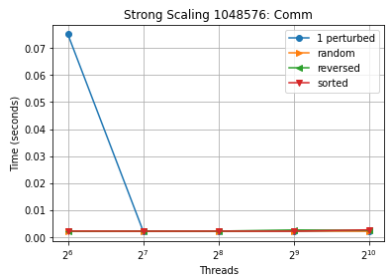
Speedup

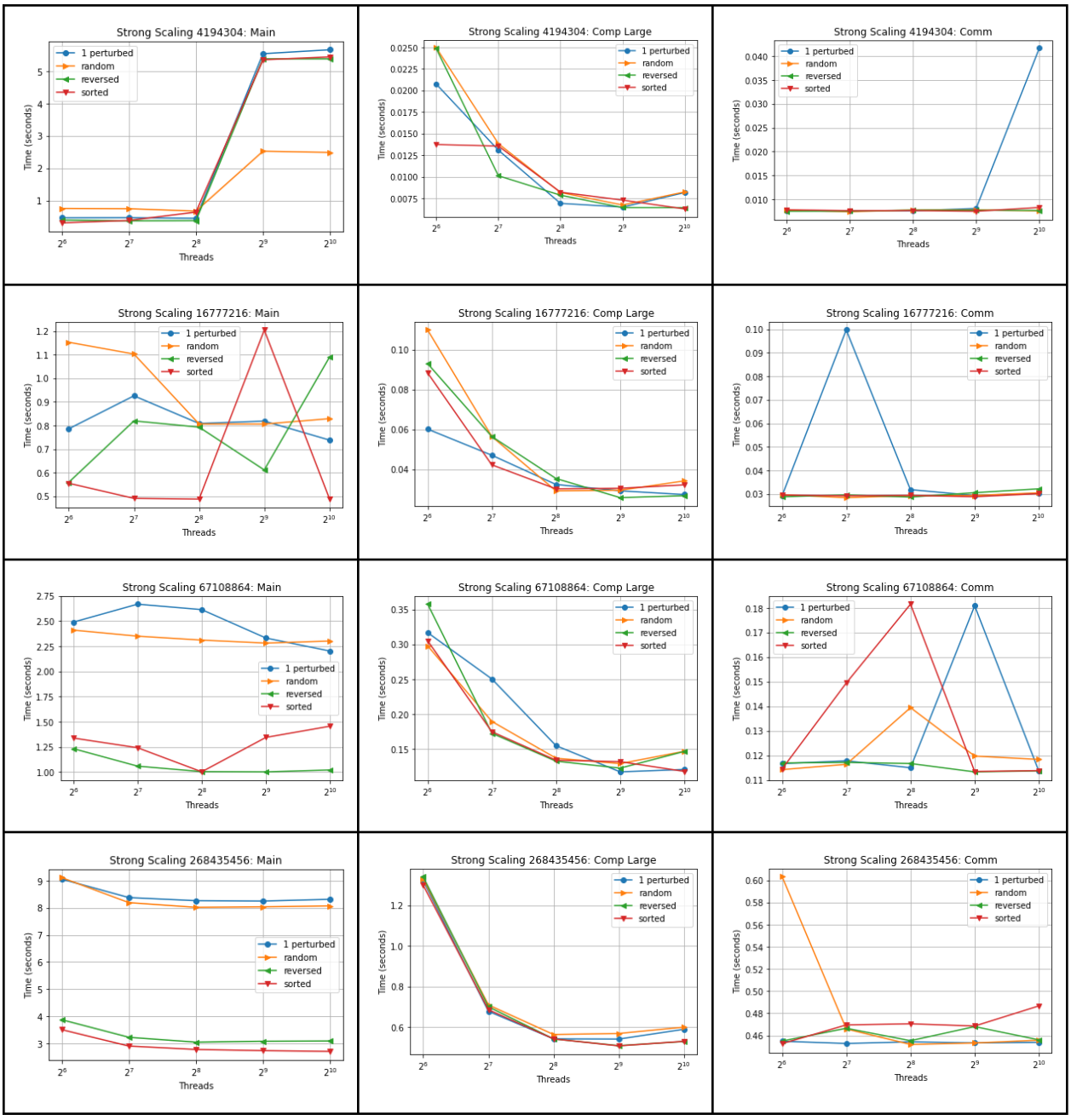
Main	Computation	Communication
		

Bitonic Sort

Strong Scaling

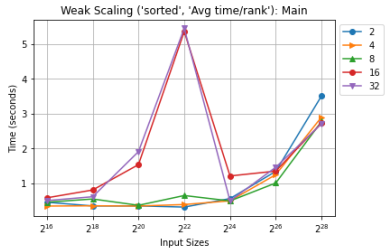
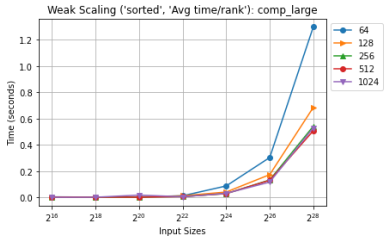
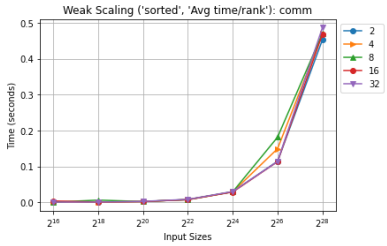
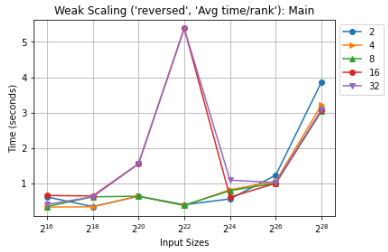
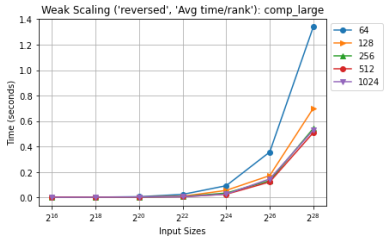
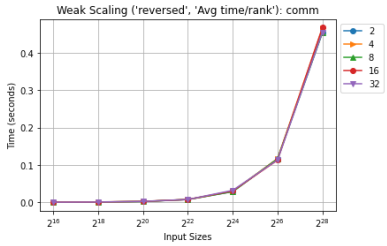
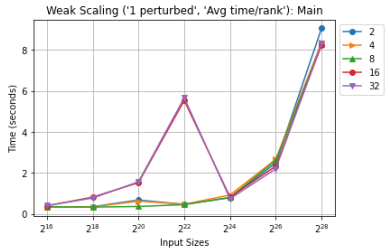
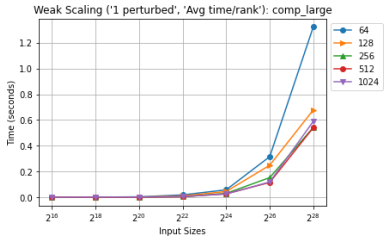
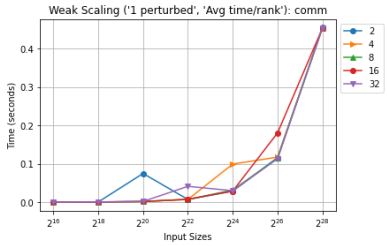
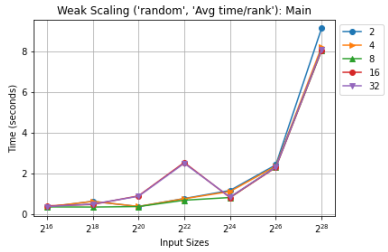
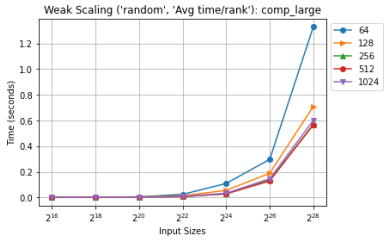
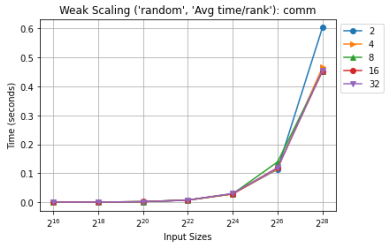
Bitonic sort had relatively good strong scaling, especially as input sizes increased. The computation time decreased with more threads while communication times stayed generally stable. In general sorted and reverse sorted data performed better, as random and 1% perturbed had spikes at some point in the computation or communication, which may have been specific to these runs, so it'd be interesting to do multiple runs and compare them to see how different node placements affect the runtimes.

Main	Computation	Communication
		
		
		



Weak Scaling

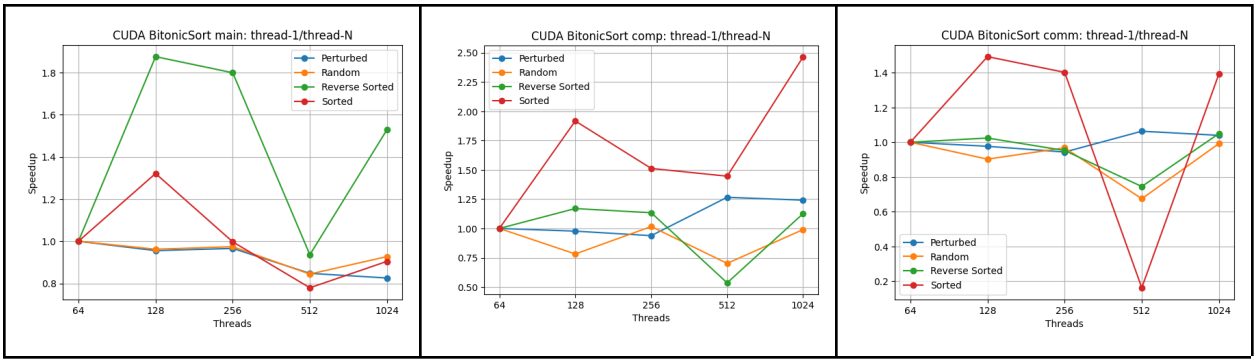
Bitonic sort weak scaling was really nice - expected increases in run time in the computation and communication and all sorting algorithms perform similarly with 1% perturbed having the most variation. For each main calculation there is a large spike at 2^22 input size, which I suppose is due to data generation or collection because the computation and communication graphs do not show evidence of this happening.

Main	Computation	Communication
		
		
		
		

Speedup

Looking at general trends in speedup, bitonic sort does have some speed up from 64 threads to the larger sizes of threads to run on for the sorted and perturbed data, while reverse sorted and random tend to have a slow down. Computation shows the greatest speedup, which does indicate an efficient algorithm in that sense that the sorts themselves are quicker with more threads. Communication is a hold up and generally has little to no impact on speed up, so for further improvement I would focus my efforts there to create more efficient communication.

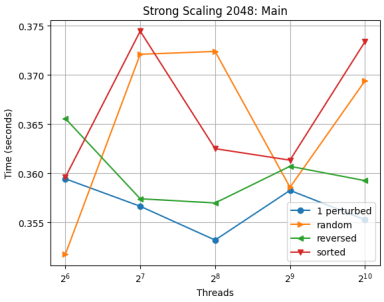
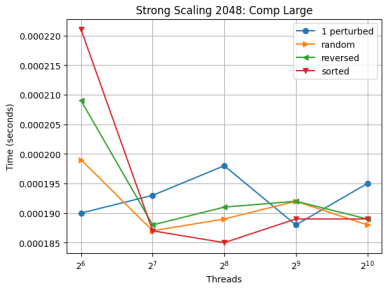
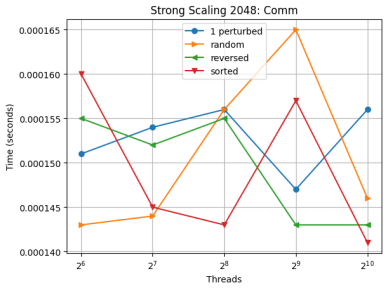
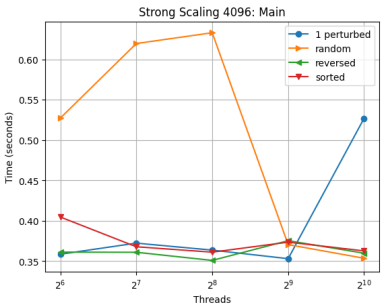
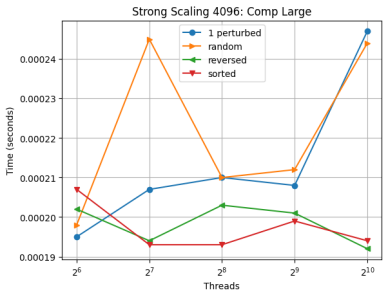
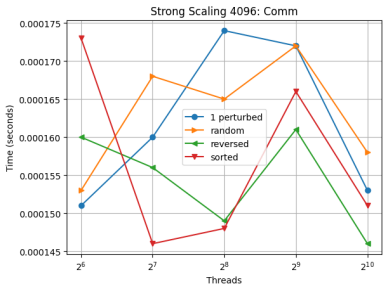
Main	Computation	Communication

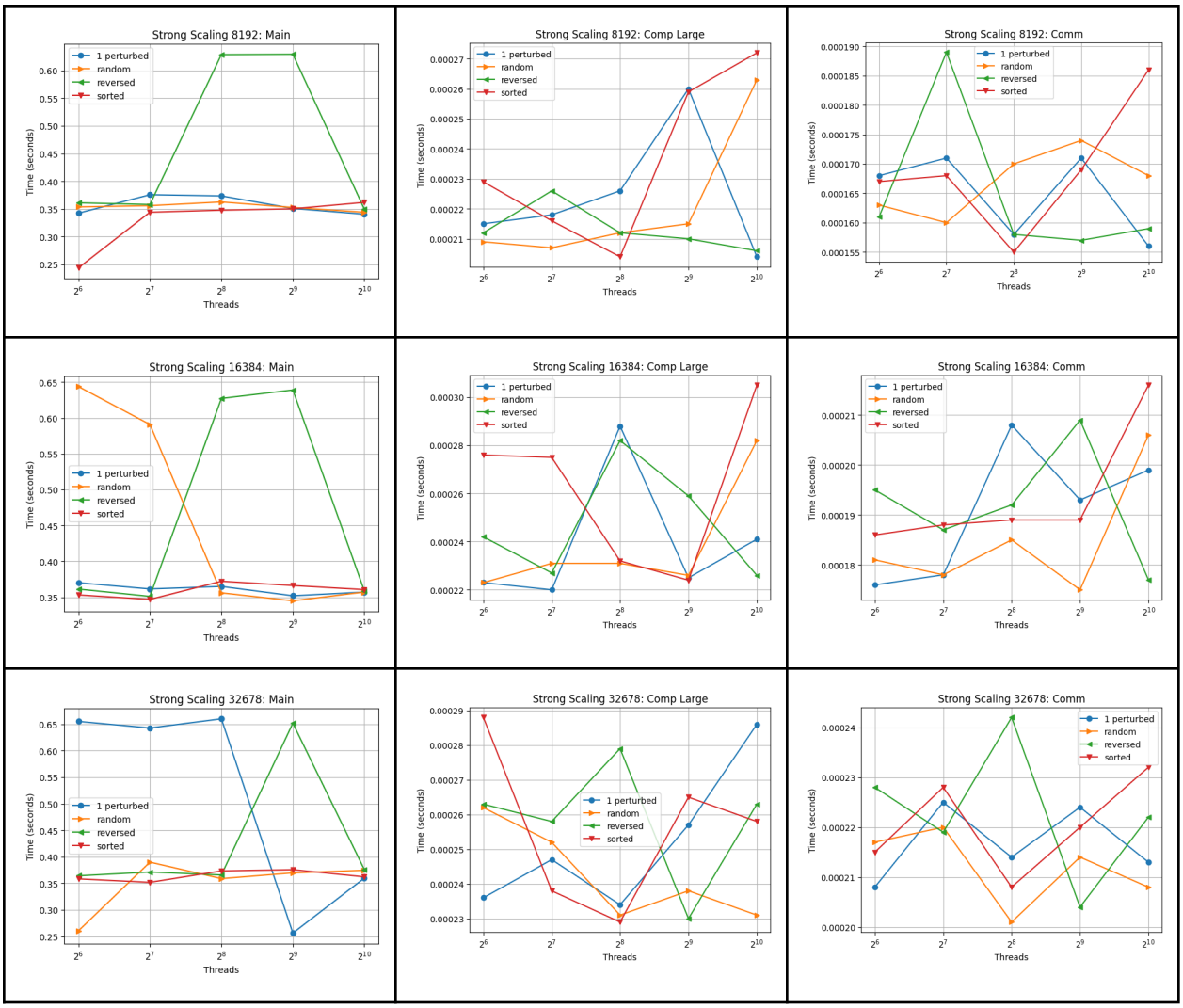


Merge Sort

Strong Scaling

Several obstacles were faced in the CUDA implementation for merge sort, specifically with running the program on larger input sizes. Since these issues were not resolved in time, the testing and graphs generated below are based on smaller input sizes (array lengths of 2^11 to 2^15). In general, it is difficult to see clear, repeated trends across the plots below; this is likely because the test input sizes were too small to be able to observe a clear benefit of parallelization.

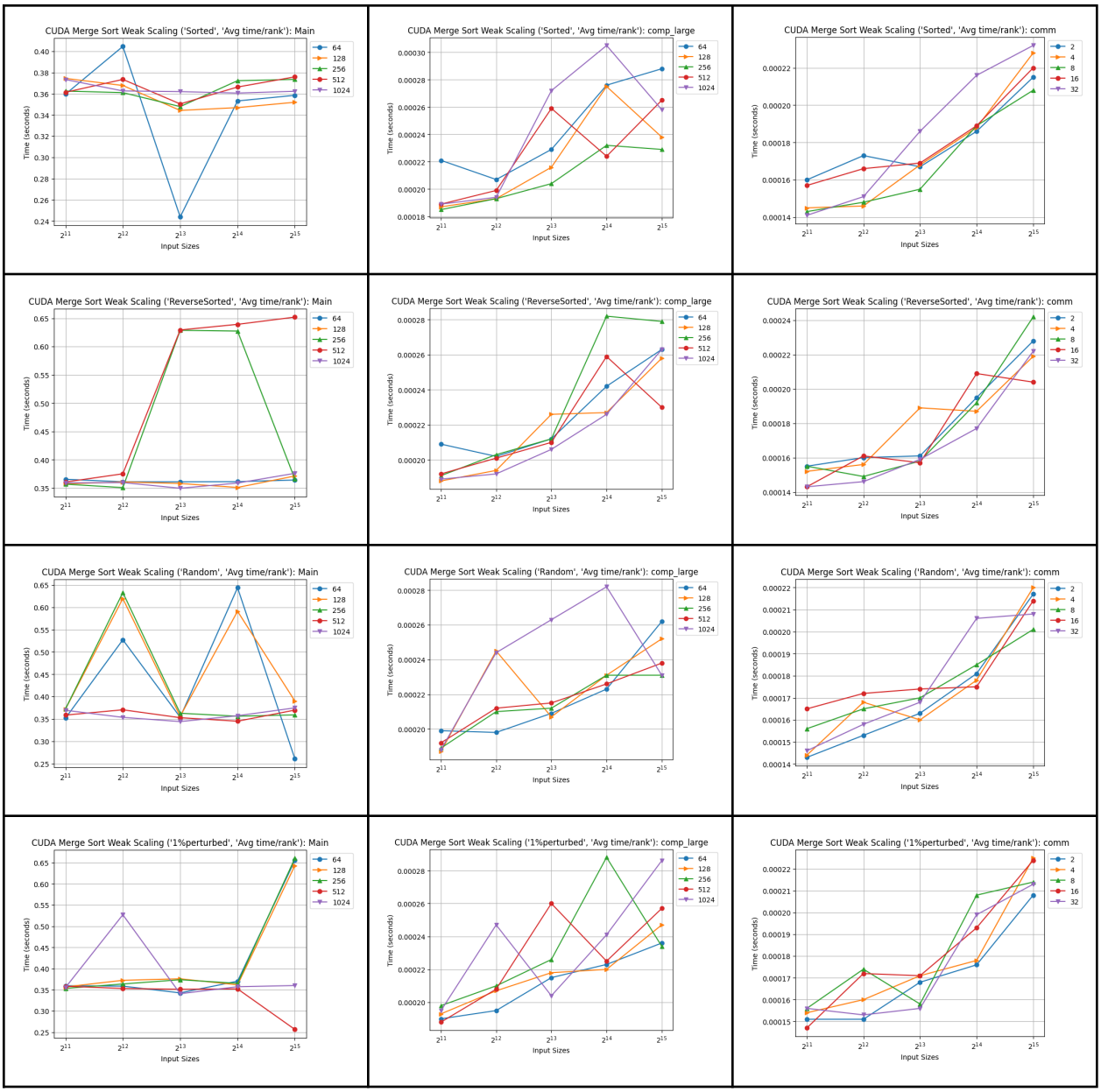
Main	Computation	Communication
		
		



Weak Scaling

There are some slightly clearer trends for the weak scaling graphs below in comparison to the strong scaling graphs, especially for the computation and communication results. All of the computation and communication plots show an upward trend, meaning runtimes increased with increasing input sizes, as expected. However, there wasn't one specific thread count that performed the best or worst across the different input types for computation or communication.

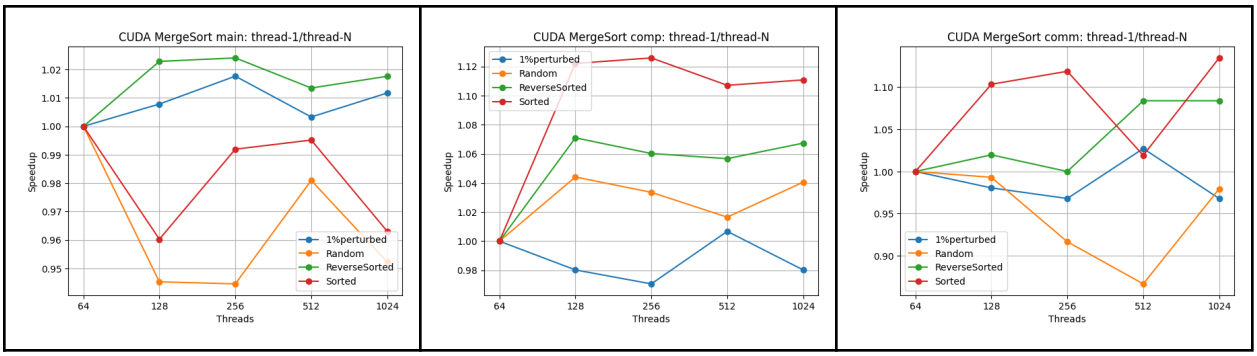
Main	Computation	Communication



Speedup

When comparing speedups across input types, it appears that reverse sorted inputs generally had speedups greater than 1 across the main, computation, and communication sections. For computation specifically, where the greatest amount of speedup was expected, three of the four input types performed fairly well, with only the 1% perturbed input type having any speedups less than 1; however, the amount of speedup was only a maximum of roughly 1.12, meaning that the parallelization did not lead to large reductions in runtime.

Main	Computation	Communication


CUDA MergeSort main: thread-1/thread-N
CUDA MergeSort comp: thread-1/thread-N
CUDA MergeSort comm: thread-1/thread-N
Speedup
Threads
64
128
256
512
1024
1%perturbed
Random
ReverseSorted
Sorted