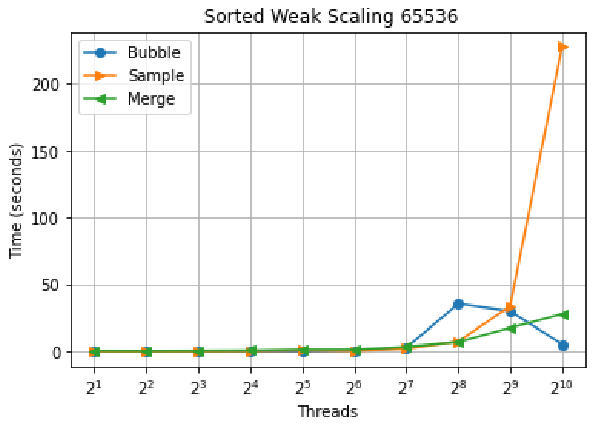
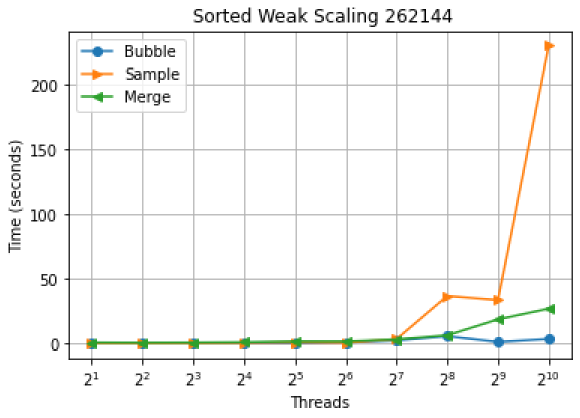
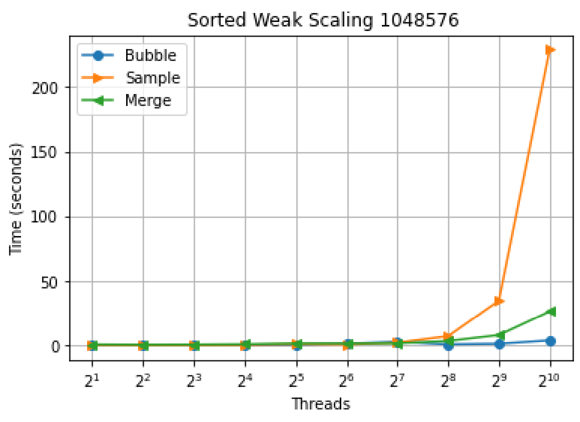
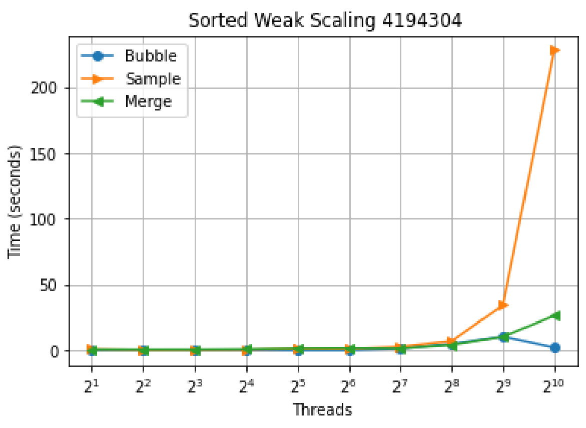
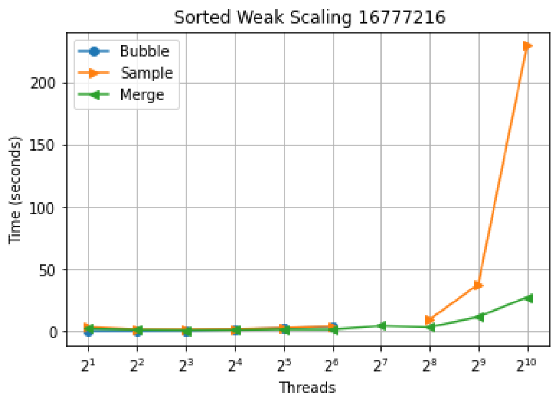
## Weak Scaling Sorted Graphs:

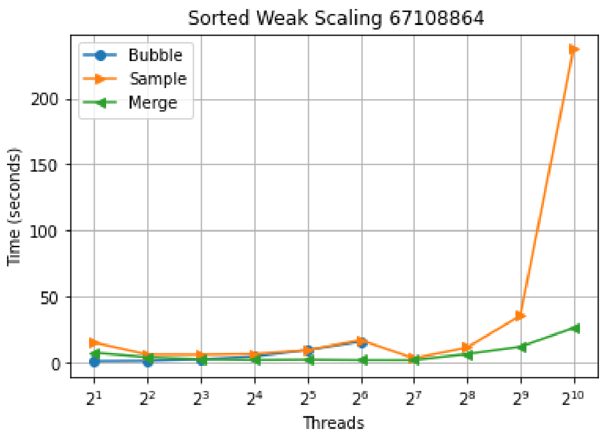


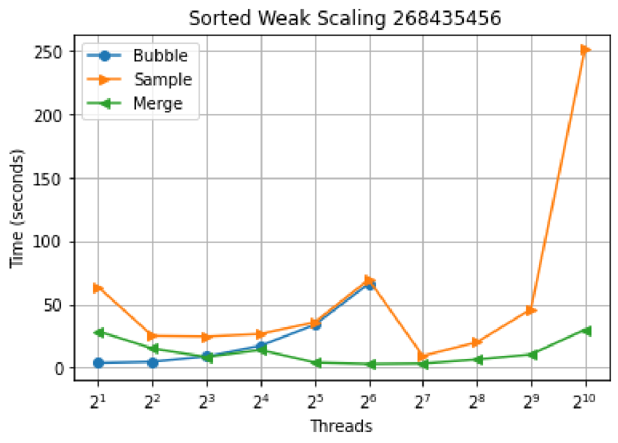






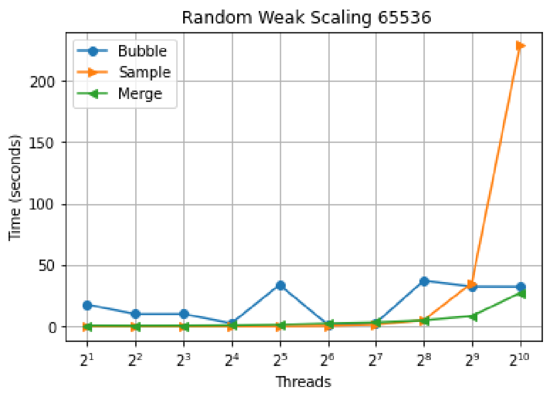


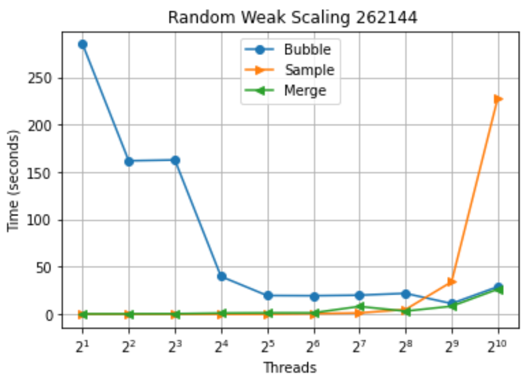


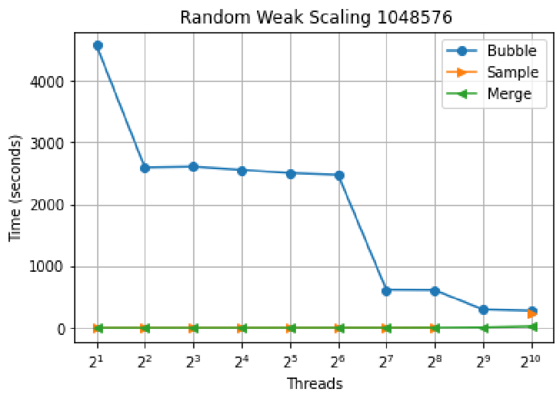


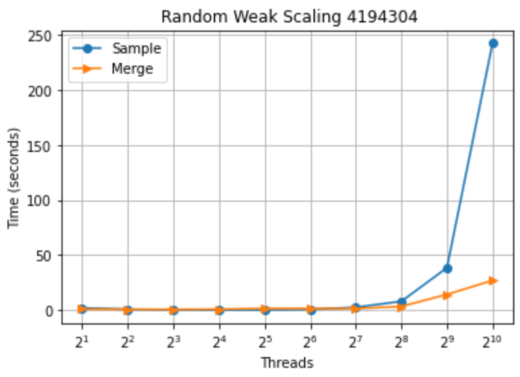
Analyzing the weak scaling graphs above, each of the three MPI sorting algorithms seems to perform relatively similarly and relatively well on this sorted data. As the number of threads increases, time stays relatively the same for merge and bubble sort. However, as the number of threads increases on 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.

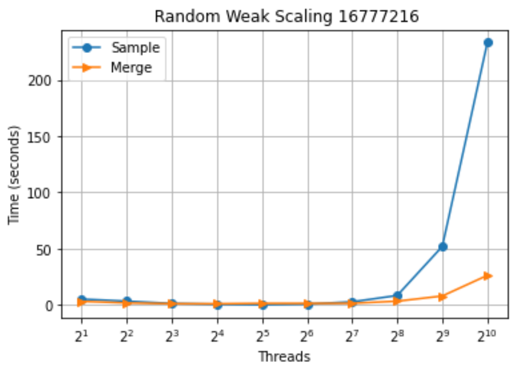
**Weak Scaling Random Graphs:**

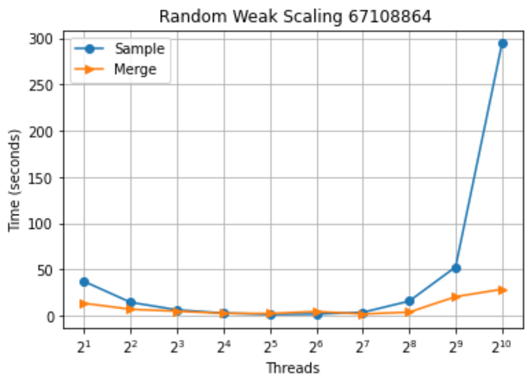


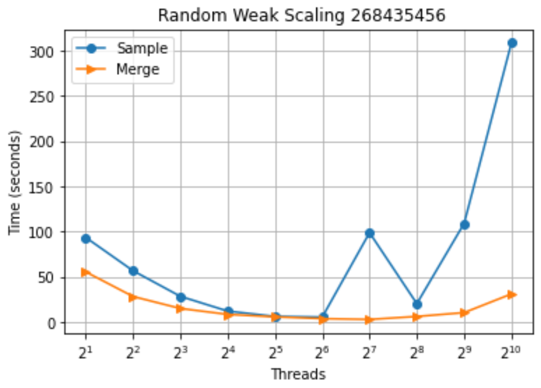






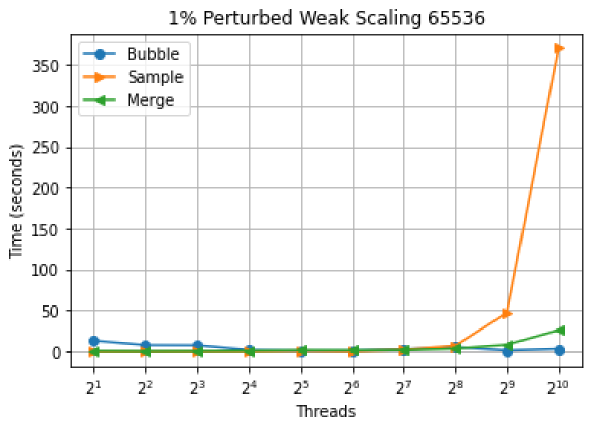


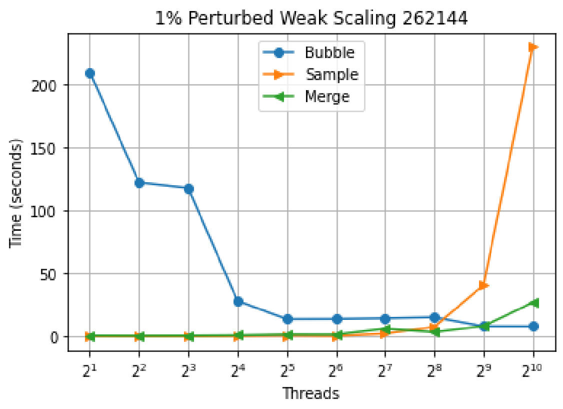


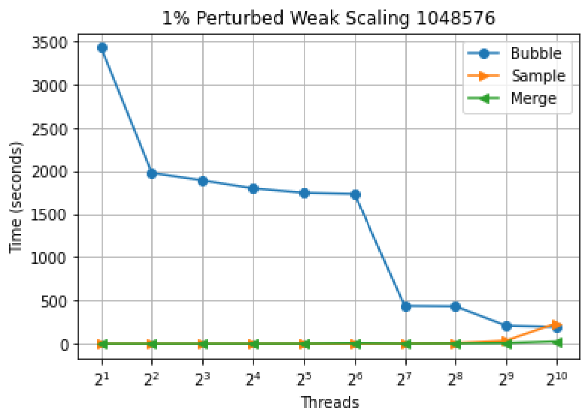


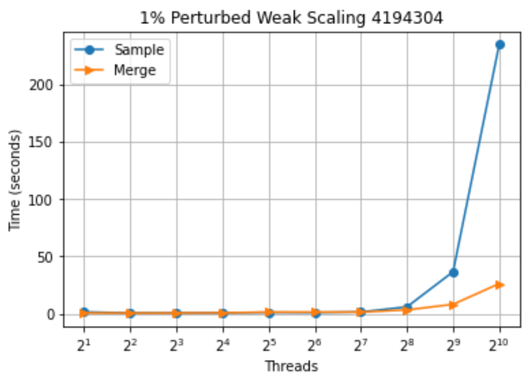
The weak scaling graphs for random generated data are quite similar to that of the sorted data. However, one big difference to note is the performance of bubble sort on the randomly generated data. As it runs into many out of order elements, it has to make a lot of swaps, driving up the run time. However,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

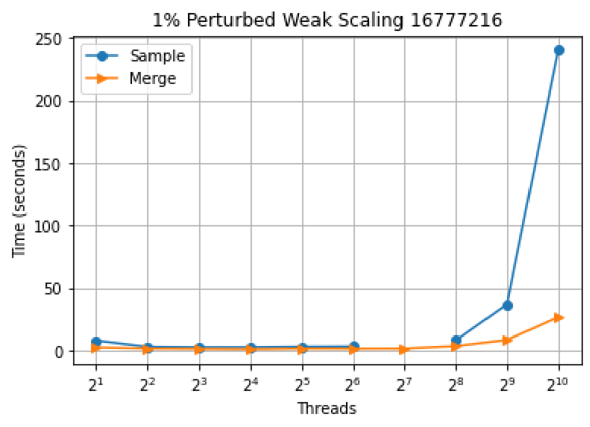
**1% Perturbed Weak Scaling:**

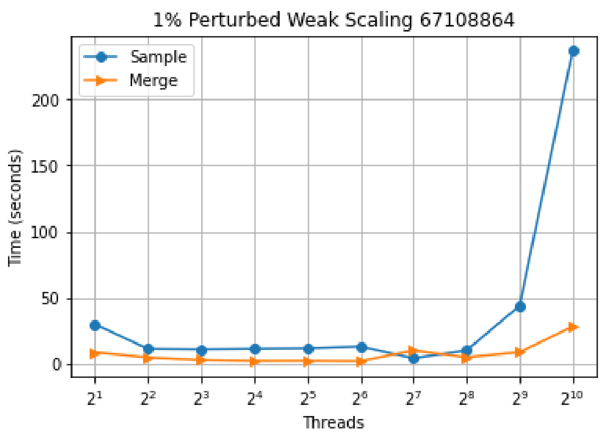

1% Perturbed Weak Scaling 65536
Bubble
Sample
Merge
Time (seconds)
Threads

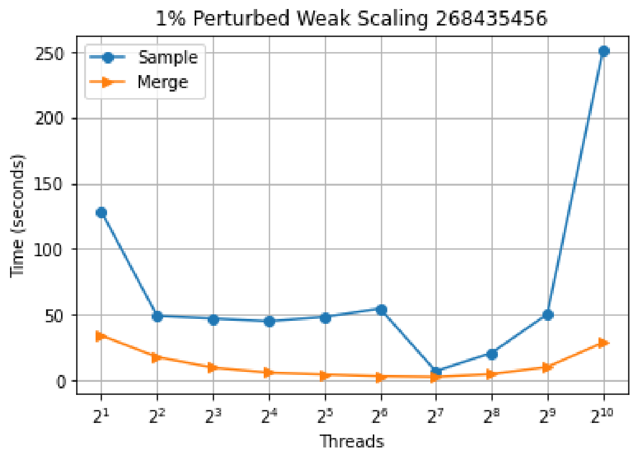

1% Perturbed Weak Scaling 262144
Bubble
Sample
Merge
Time (seconds)
Threads


1% Perturbed Weak Scaling 1048576
Bubble
Sample
Merge
Time (seconds)
Threads


1% Perturbed Weak Scaling 4194304
Sample
Merge
Time (seconds)
Threads

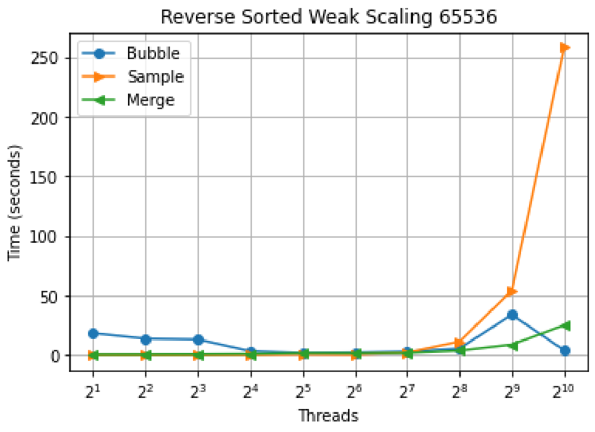
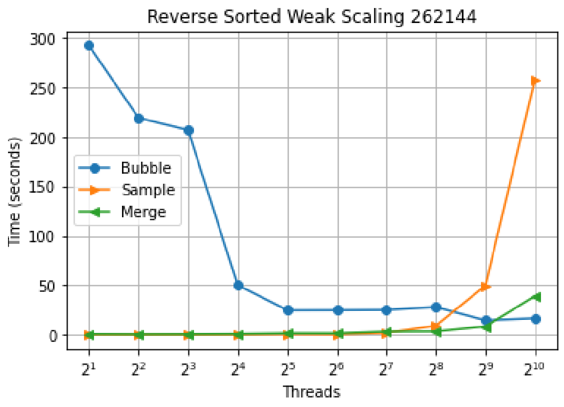
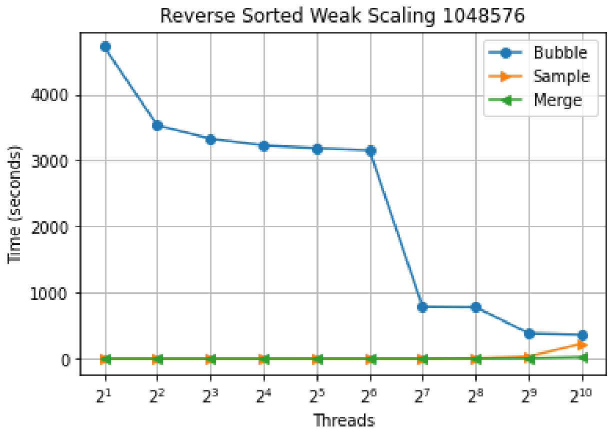
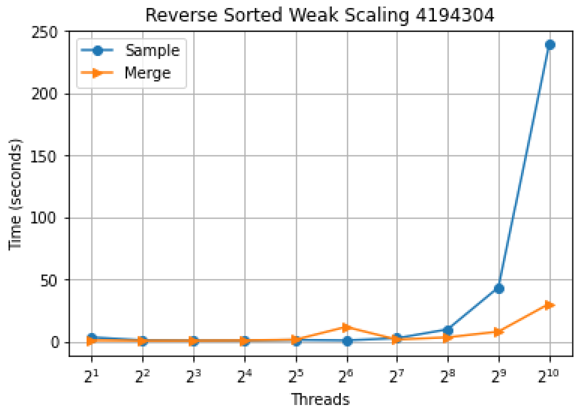


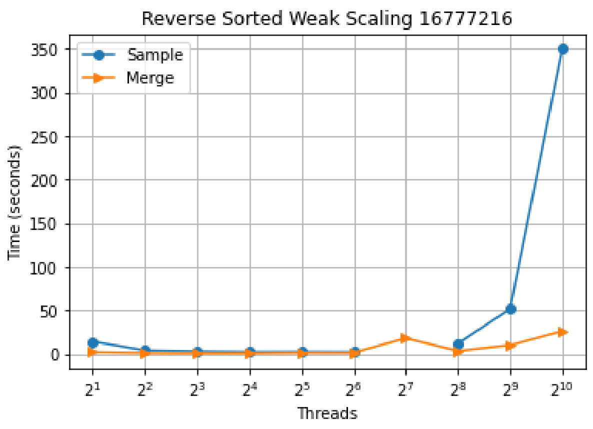


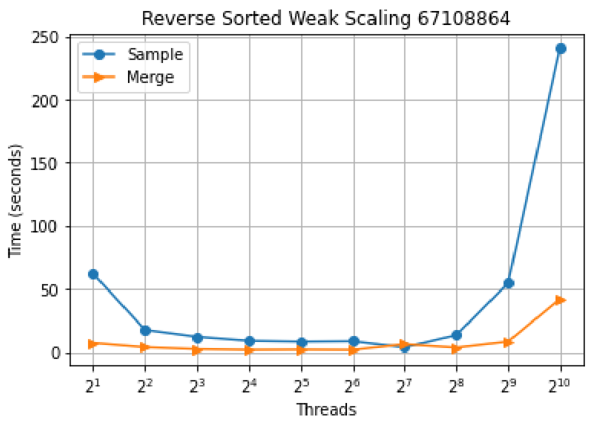


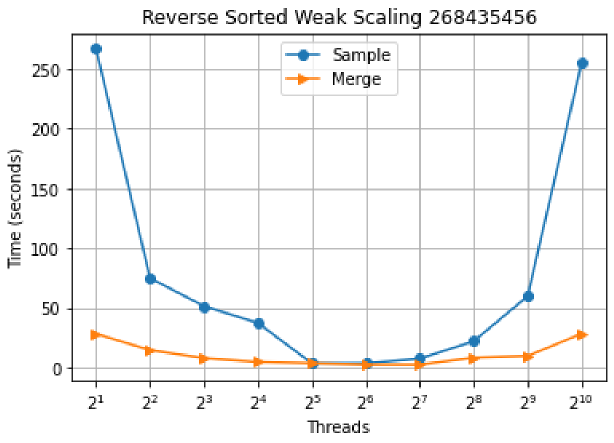
The observations for these graphs are almost exactly the same as the Randomly sorted data above. The only difference to note is that the run times are slightly lower for bubble sort on the 1% perturbed data than on the random data. This is because not as many elements are out of place. Merge sort performed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

**Reverse Sorted Weak Scaling:**


Reverse Sorted Weak Scaling 65536
Bubble
Sample
Merge
Time (seconds)
Threads
0
50
100
150
200
250
$2^1$ $2^2$ $2^3$ $2^4$ $2^5$ $2^6$ $2^7$ $2^8$ $2^9$ $2^{10}$

Reverse Sorted Weak Scaling 262144
Bubble
Sample
Merge
Time (seconds)
Threads
0
50
100
150
200
250
300
$2^1$ $2^2$ $2^3$ $2^4$ $2^5$ $2^6$ $2^7$ $2^8$ $2^9$ $2^{10}$

Reverse Sorted Weak Scaling 1048576
Bubble
Sample
Merge
Time (seconds)
Threads
0
1000
2000
3000
4000
$2^1$ $2^2$ $2^3$ $2^4$ $2^5$ $2^6$ $2^7$ $2^8$ $2^9$ $2^{10}$

Reverse Sorted Weak Scaling 4194304
Sample
Merge
Time (seconds)
Threads
0
50
100
150
200
250
$2^1$ $2^2$ $2^3$ $2^4$ $2^5$ $2^6$ $2^7$ $2^8$ $2^9$ $2^{10}$

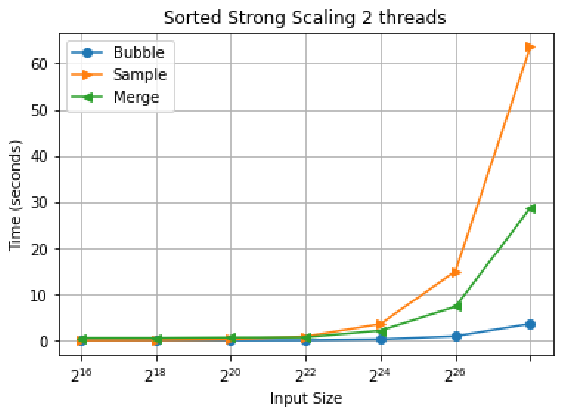


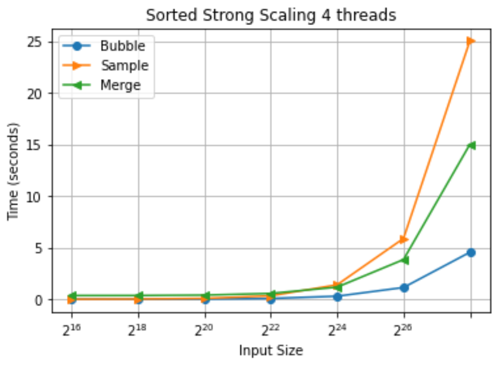


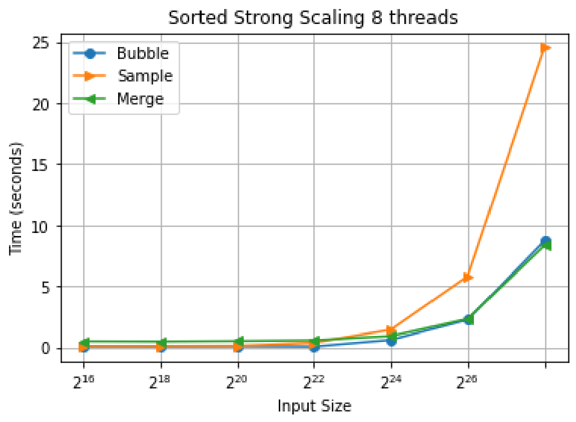


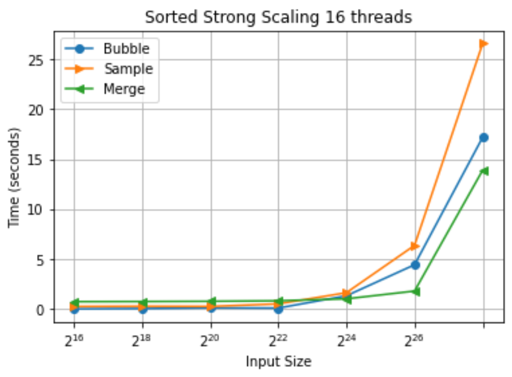
Observations to note on these graphs are also mainly the same as the previous sections. Some things to note are that for bubble sort, this algorithm's run time was the highest. This is because each element is as far from its intended location as it can get, meaning it needs the most swaps to get to its 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.

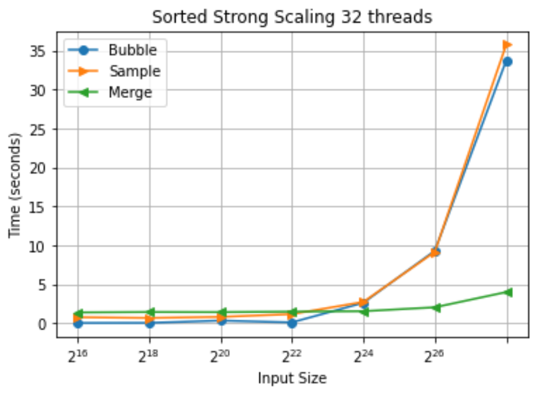
**Strong Scaling Sorted:**

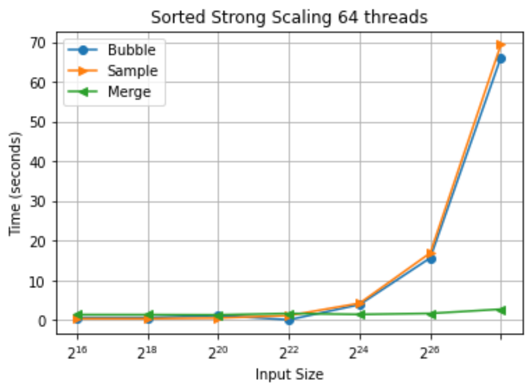

Sorted Strong Scaling 2 threads
Bubble
Sample
Merge
Time (seconds)
Input Size

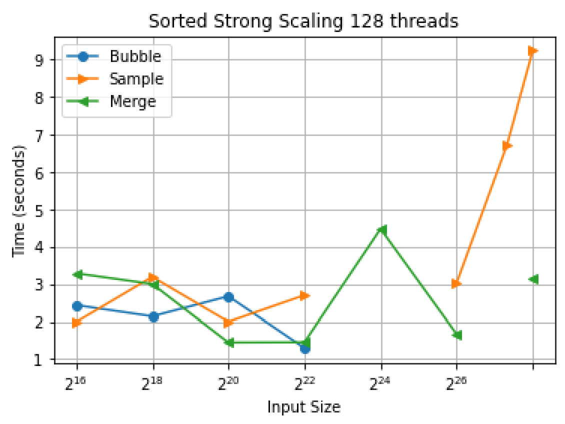

Sorted Strong Scaling 4 threads
Bubble
Sample
Merge
Time (seconds)
Input Size

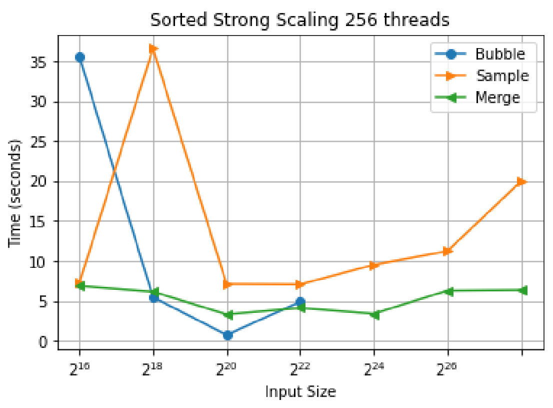

Sorted Strong Scaling 8 threads
Bubble
Sample
Merge
Time (seconds)
Input Size

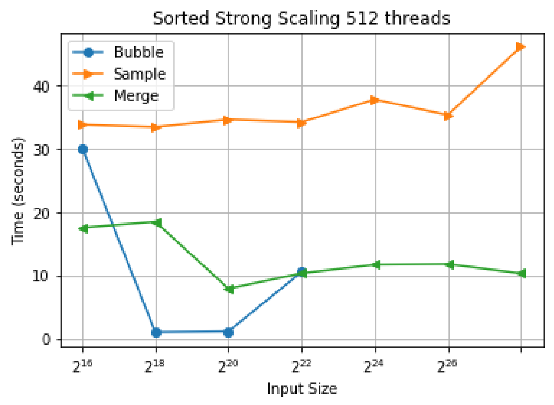

Sorted Strong Scaling 16 threads
Bubble
Sample
Merge
Time (seconds)
Input Size

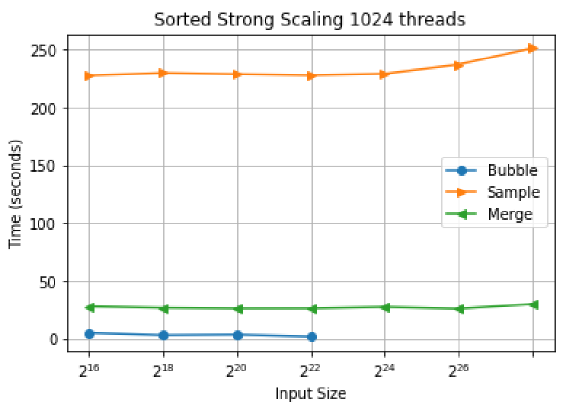

Sorted Strong Scaling 32 threads
Bubble
Sample
Merge
Time (seconds)
Input Size


Sorted Strong Scaling 64 threads
Bubble
Sample
Merge
Time (seconds)
Input Size



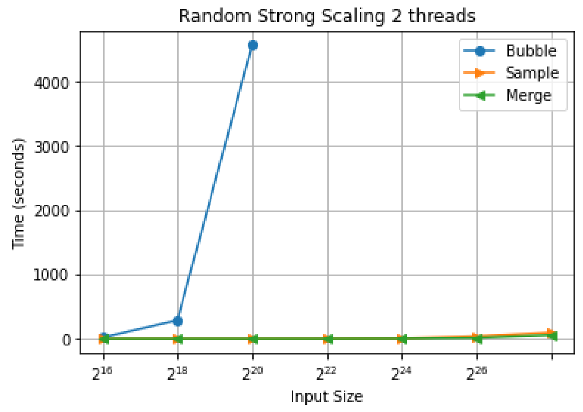


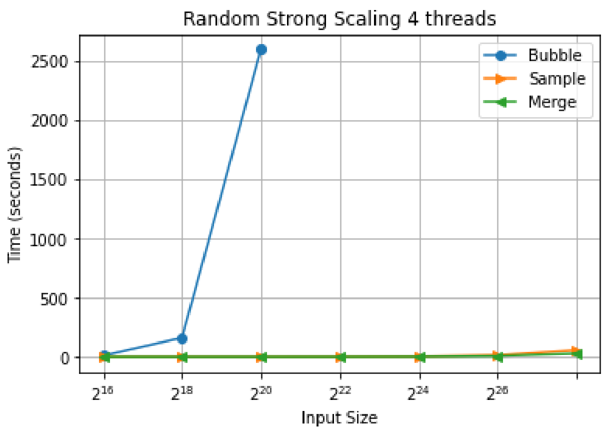


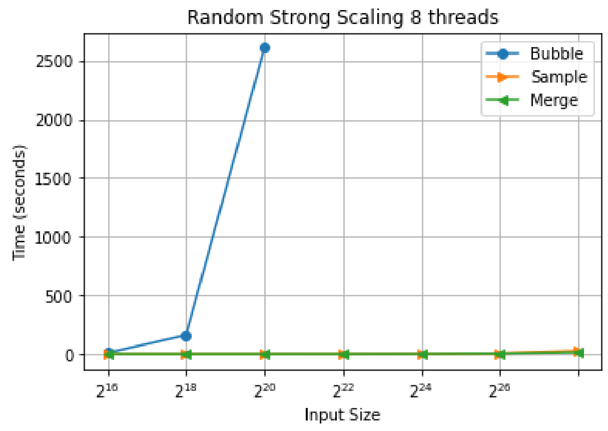


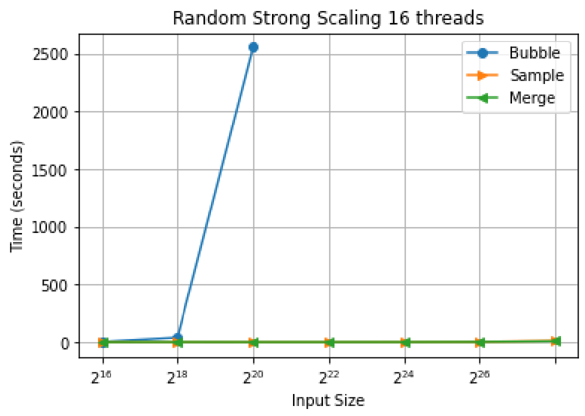
For the graphs above, from 2-64 threads the runtime for each starting algorithm starts low and remains low until the 2^24 input size, at which it rockets up. This makes sense because as the input size goes up, you would ideally have more threads to split the work. A lower number of threads then results 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.

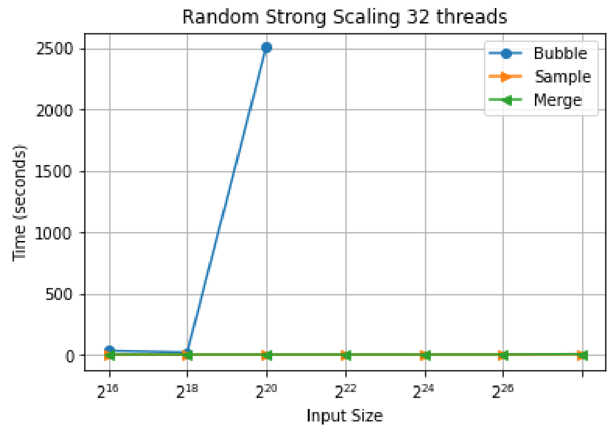
**Strong Scaling Random:**

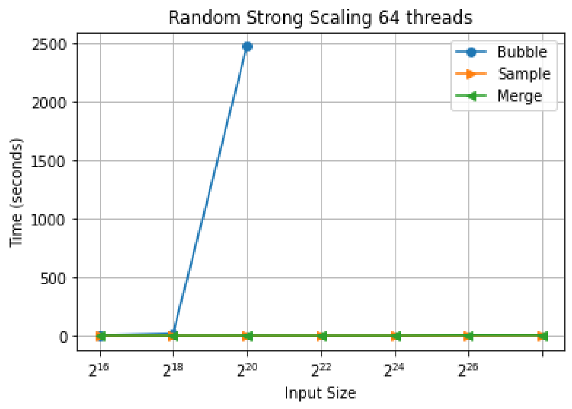

Random Strong Scaling 2 threads
Bubble
Sample
Merge
Time (seconds)
0
1000
2000
3000
4000
Input Size
$2^{16}$
$2^{18}$
$2^{20}$
$2^{22}$
$2^{24}$
$2^{26}$

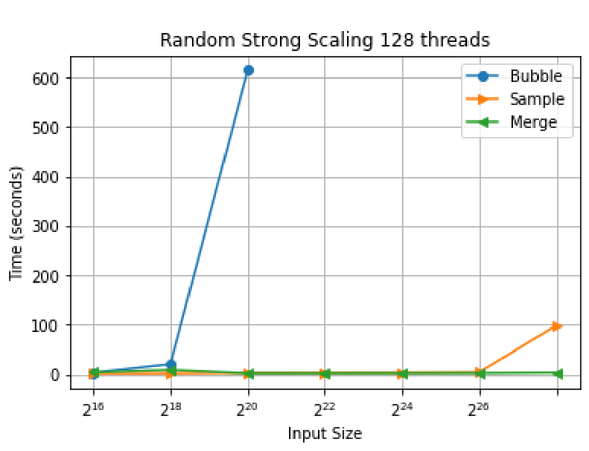

Random Strong Scaling 4 threads
Bubble
Sample
Merge
Time (seconds)
0
500
1000
1500
2000
2500
Input Size
$2^{16}$
$2^{18}$
$2^{20}$
$2^{22}$
$2^{24}$
$2^{26}$

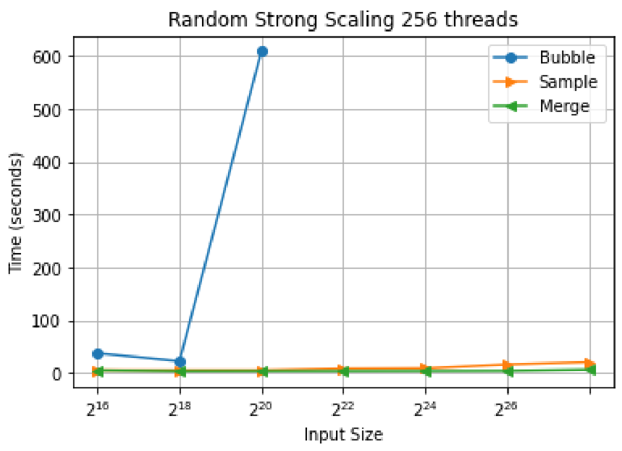

Random Strong Scaling 8 threads
Bubble
Sample
Merge
Time (seconds)
0
500
1000
1500
2000
2500
Input Size
$2^{16}$
$2^{18}$
$2^{20}$
$2^{22}$
$2^{24}$
$2^{26}$

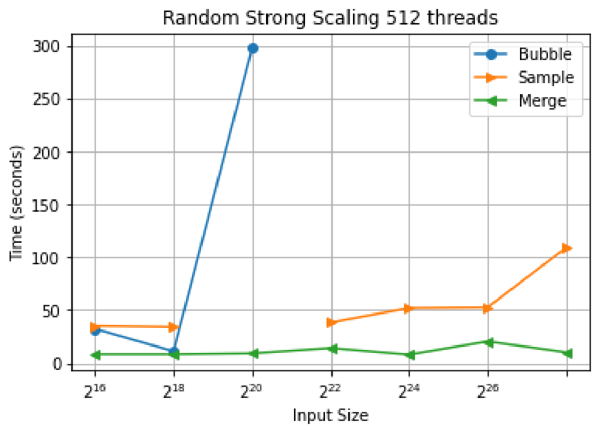

Random Strong Scaling 16 threads
Bubble
Sample
Merge
Time (seconds)
0
500
1000
1500
2000
2500
Input Size
$2^{16}$
$2^{18}$
$2^{20}$
$2^{22}$
$2^{24}$
$2^{26}$

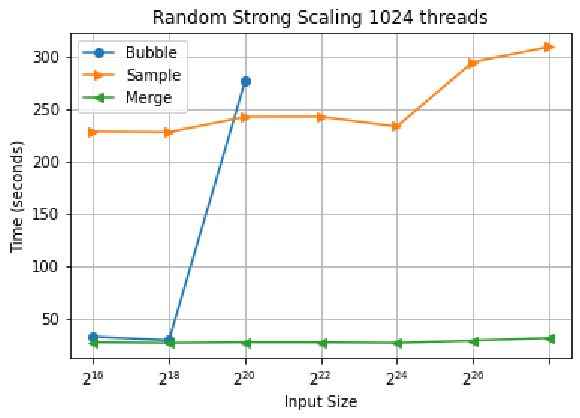

Random Strong Scaling 32 threads
Bubble
Sample
Merge
Time (seconds)
0
500
1000
1500
2000
2500
Input Size
$2^{16}$
$2^{18}$
$2^{20}$
$2^{22}$
$2^{24}$
$2^{26}$


Random Strong Scaling 64 threads
Bubble
Sample
Merge
Time (seconds)
0
500
1000
1500
2000
2500
Input Size
$2^{16}$
$2^{18}$
$2^{20}$
$2^{22}$
$2^{24}$
$2^{26}$



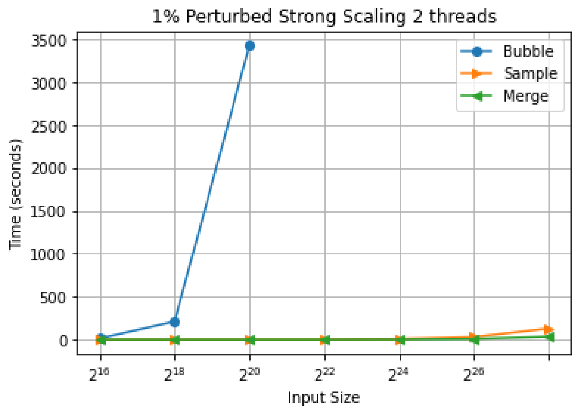


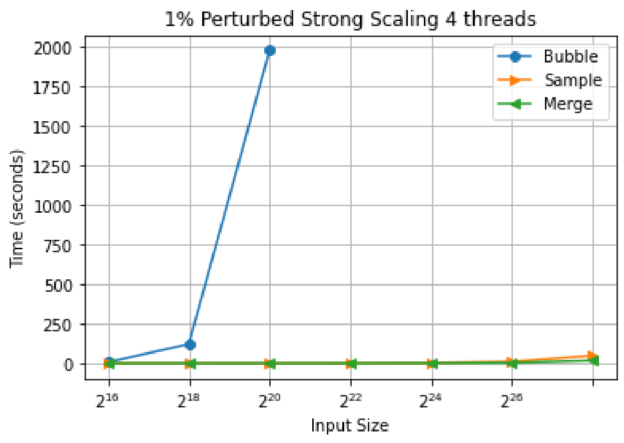


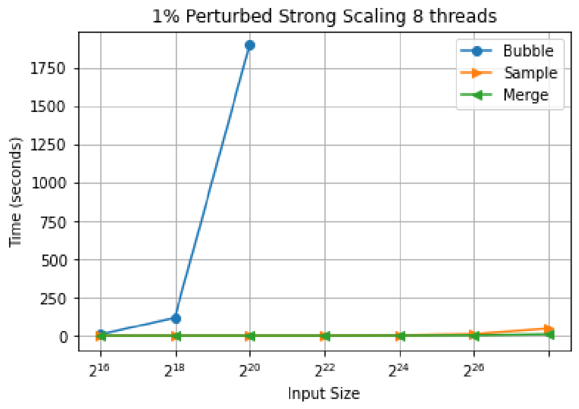


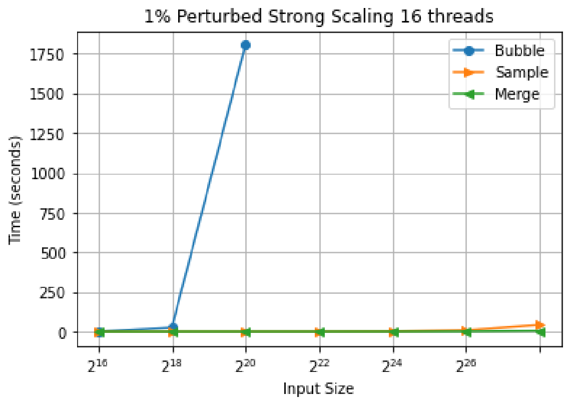
When comparing the graphs above, it appears that while merge and sample sort have fairly constant runtimes for the same number of threads regardless of input size, bubble sort has a sharp increase in runtime from the input size of 2^18 to 2^20; these trends can be observed in all of the graphs, meaning 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.

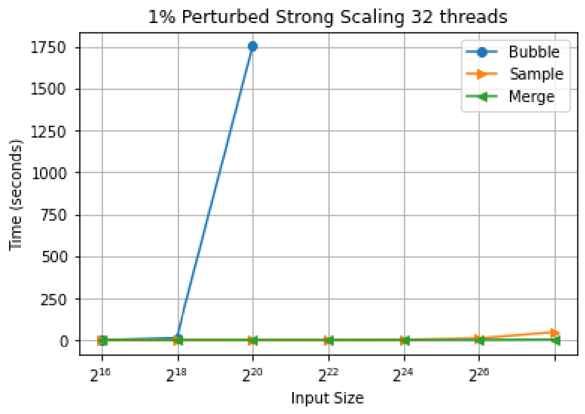
## 1% Perturbed Strong Scaling

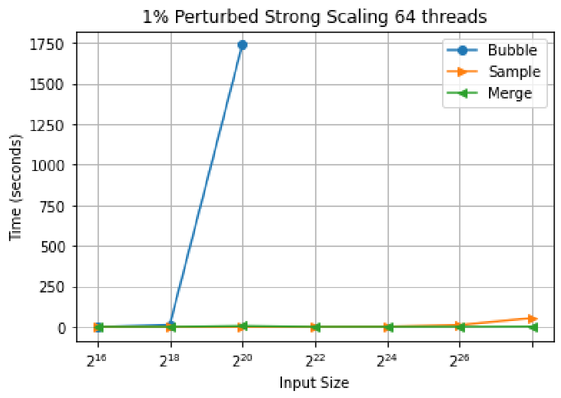

1% Perturbed Strong Scaling 2 threads
Bubble
Sample
Merge
Time (seconds)
3500
3000
2500
2000
1500
1000
500
0
$2^{16}$
$2^{18}$
$2^{20}$
$2^{22}$
$2^{24}$
$2^{26}$
Input Size

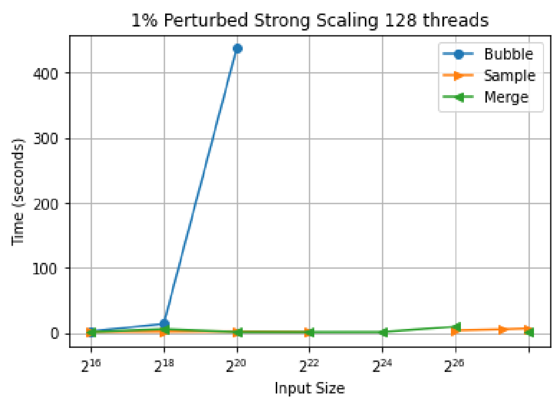

1% Perturbed Strong Scaling 4 threads
Bubble
Sample
Merge
Time (seconds)
2000
1750
1500
1250
1000
750
500
250
0
$2^{16}$
$2^{18}$
$2^{20}$
$2^{22}$
$2^{24}$
$2^{26}$
Input Size

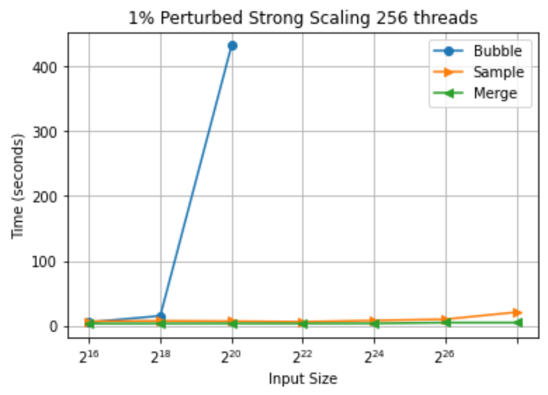

1% Perturbed Strong Scaling 8 threads
Bubble
Sample
Merge
Time (seconds)
1750
1500
1250
1000
750
500
250
0
$2^{16}$
$2^{18}$
$2^{20}$
$2^{22}$
$2^{24}$
$2^{26}$
Input Size

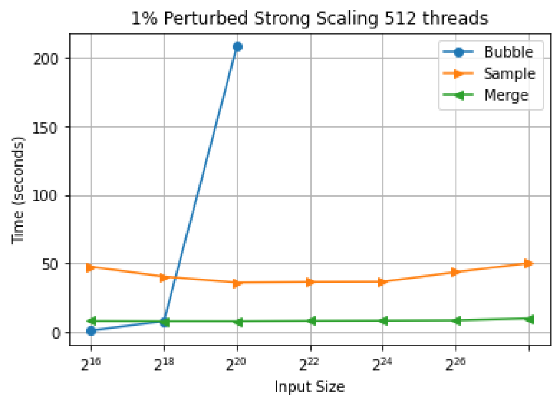

1% Perturbed Strong Scaling 16 threads
Bubble
Sample
Merge
Time (seconds)
1750
1500
1250
1000
750
500
250
0
$2^{16}$
$2^{18}$
$2^{20}$
$2^{22}$
$2^{24}$
$2^{26}$
Input Size

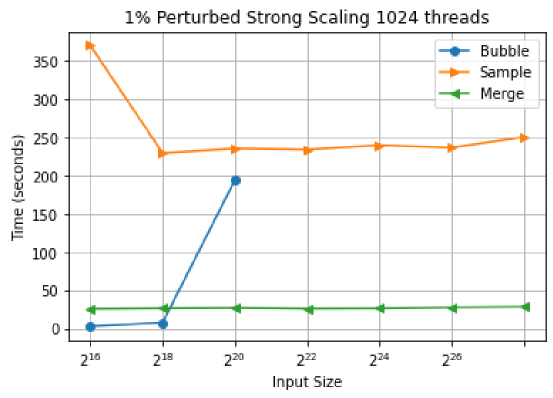

1% Perturbed Strong Scaling 32 threads
Bubble
Sample
Merge
Time (seconds)
1750
1500
1250
1000
750
500
250
0
$2^{16}$
$2^{18}$
$2^{20}$
$2^{22}$
$2^{24}$
$2^{26}$
Input Size


1% Perturbed Strong Scaling 64 threads
Bubble
Sample
Merge
Time (seconds)
1750
1500
1250
1000
750
500
250
0
$2^{16}$
$2^{18}$
$2^{20}$
$2^{22}$
$2^{24}$
$2^{26}$
Input Size



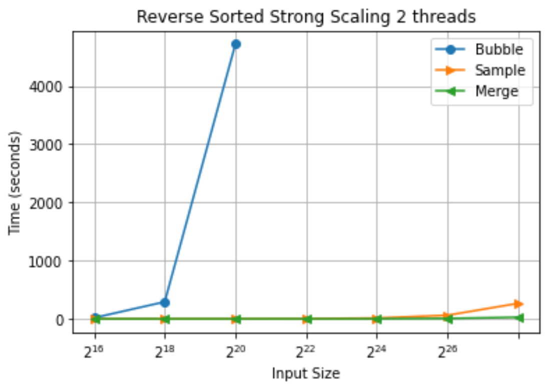


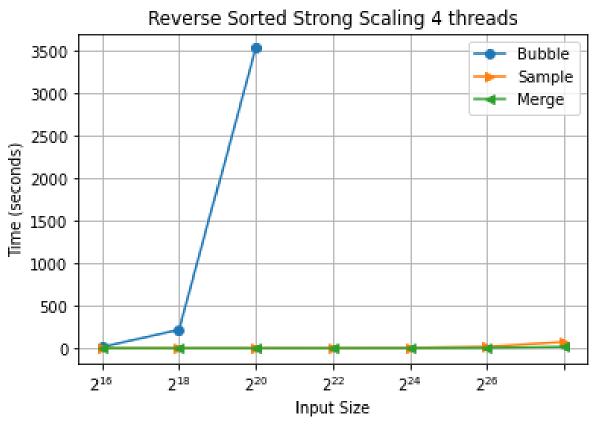


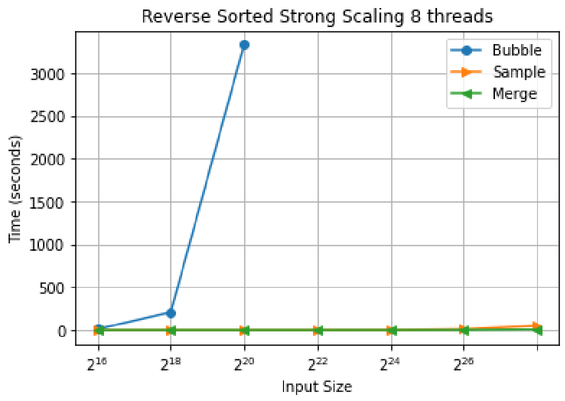


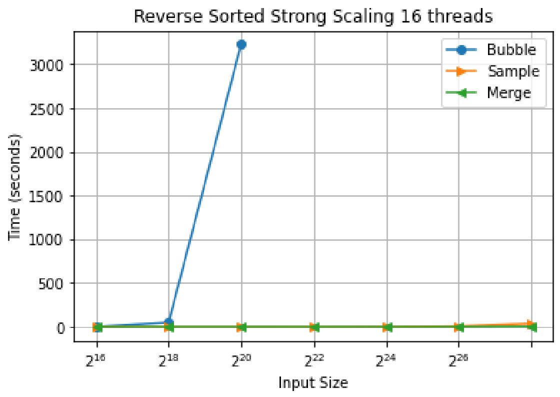
From 2-256 threads, the graphs above are very similar. Sample and merge sort have a relatively low run time (with sample sort taking a little longer), while bubble sort's run time shoots up for each as the input size increases. This makes sense because although not many elements are out of order it takes 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.

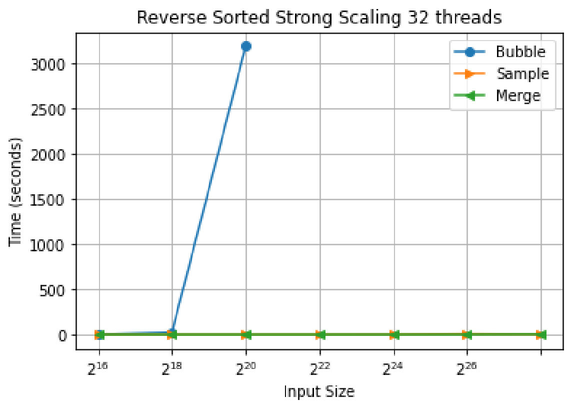
**Reverse Sorted Strong Scaling:**

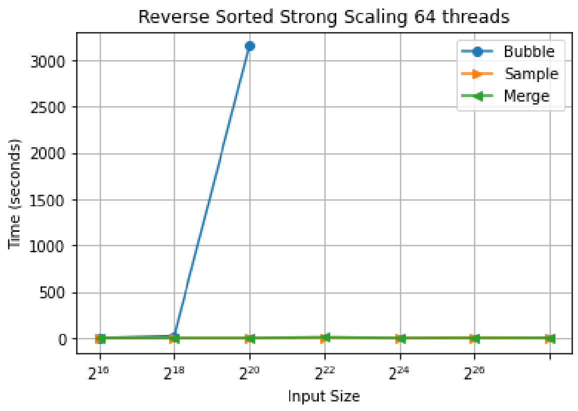

Reverse Sorted Strong Scaling 2 threads
Bubble
Sample
Merge
Time (seconds)
Input Size
0
1000
2000
3000
4000
$2^{16}$
$2^{18}$
$2^{20}$
$2^{22}$
$2^{24}$
$2^{26}$

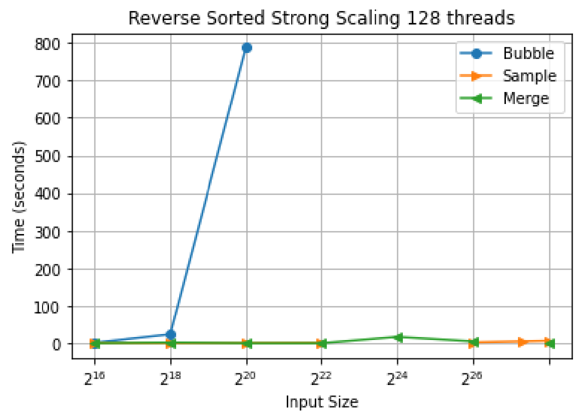

Reverse Sorted Strong Scaling 4 threads
Bubble
Sample
Merge
Time (seconds)
Input Size
0
500
1000
1500
2000
2500
3000
3500
$2^{16}$
$2^{18}$
$2^{20}$
$2^{22}$
$2^{24}$
$2^{26}$

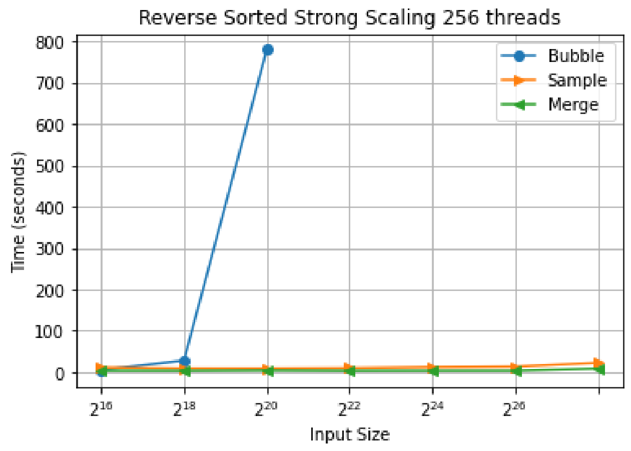

Reverse Sorted Strong Scaling 8 threads
Bubble
Sample
Merge
Time (seconds)
Input Size
0
500
1000
1500
2000
2500
3000
$2^{16}$
$2^{18}$
$2^{20}$
$2^{22}$
$2^{24}$
$2^{26}$

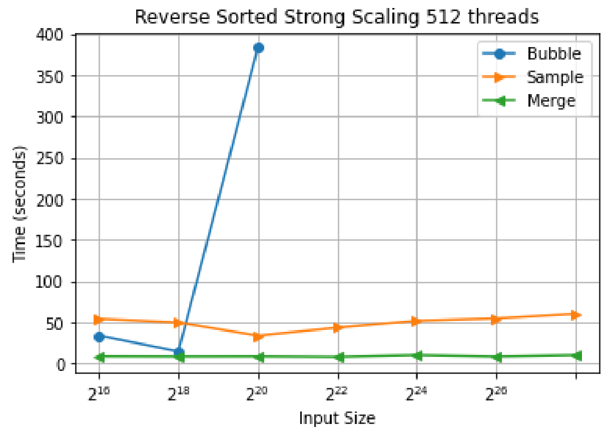

Reverse Sorted Strong Scaling 16 threads
Bubble
Sample
Merge
Time (seconds)
Input Size
0
500
1000
1500
2000
2500
3000
$2^{16}$
$2^{18}$
$2^{20}$
$2^{22}$
$2^{24}$
$2^{26}$

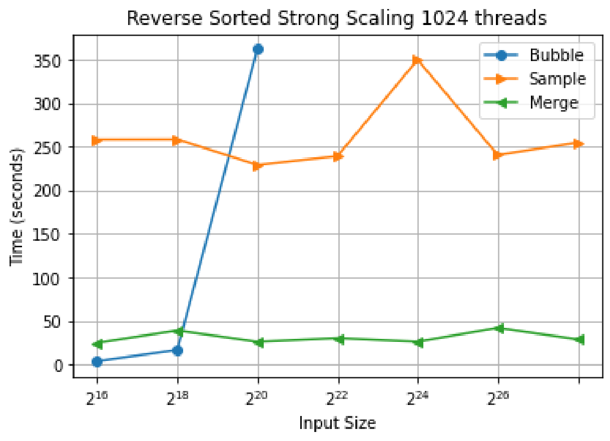

Reverse Sorted Strong Scaling 32 threads
Bubble
Sample
Merge
Time (seconds)
Input Size
0
500
1000
1500
2000
2500
3000
$2^{16}$
$2^{18}$
$2^{20}$
$2^{22}$
$2^{24}$
$2^{26}$


Reverse Sorted Strong Scaling 64 threads
Bubble
Sample
Merge
Time (seconds)
Input Size
0
500
1000
1500
2000
2500
3000
$2^{16}$
$2^{18}$
$2^{20}$
$2^{22}$
$2^{24}$
$2^{26}$



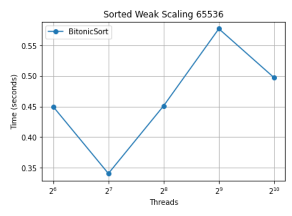
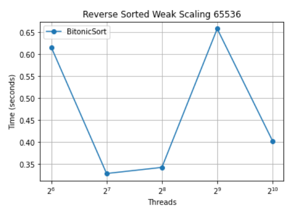
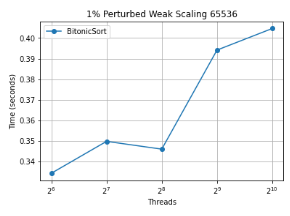
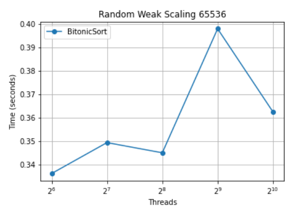


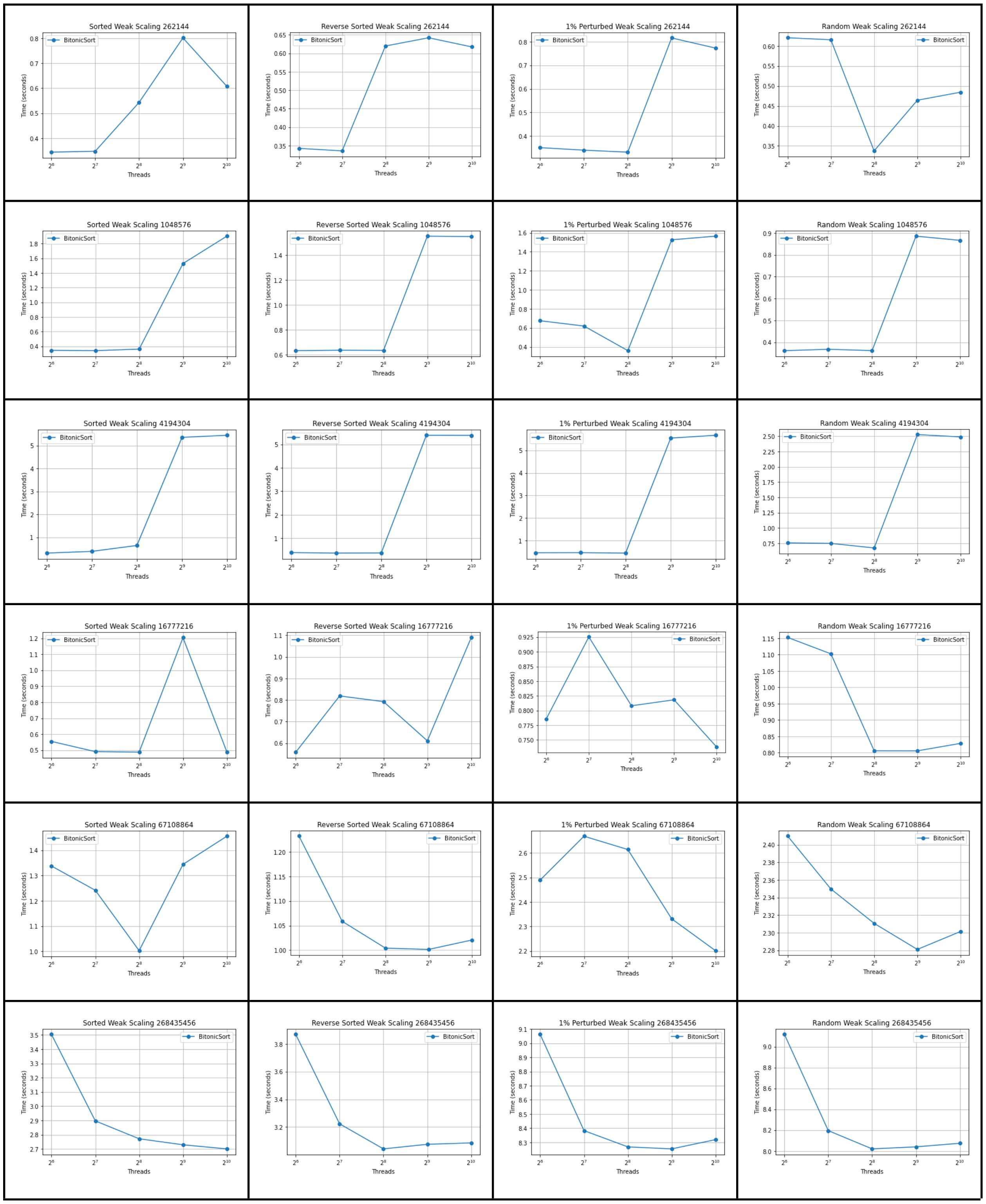




From looking at the graphs above, similar trends to the other input types can be observed, with bubble sort having sharp increases in runtime with increases in input size, while merge and sample sort have relatively constant, lower runtimes across input sizes for a fixed number of threads. Again, the 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.

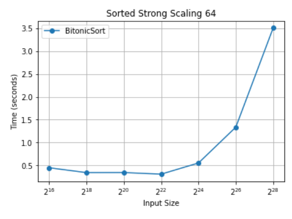
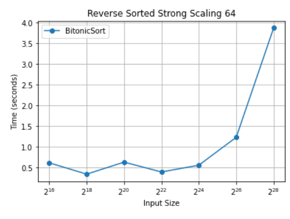
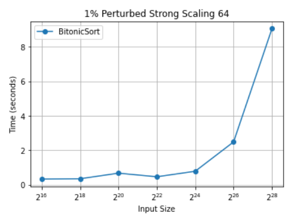
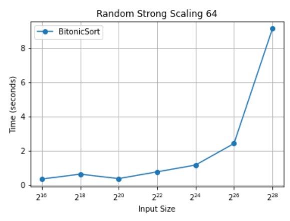
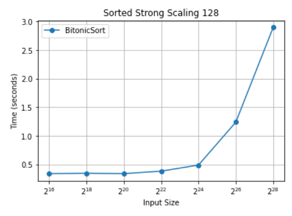
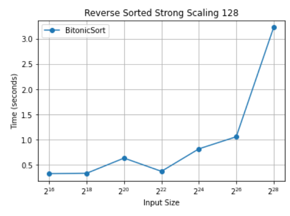
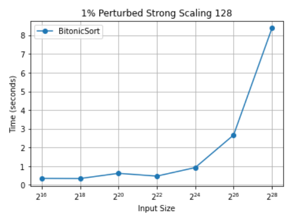
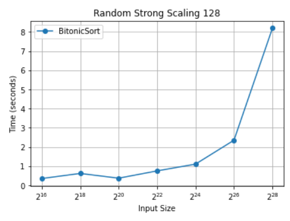
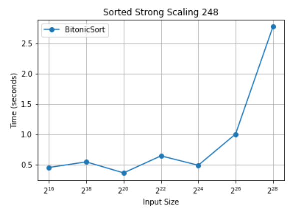
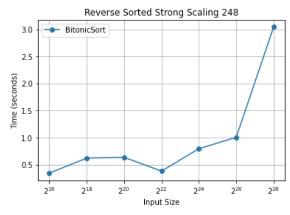
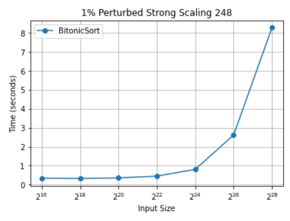
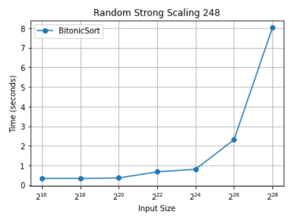
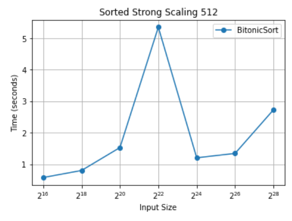
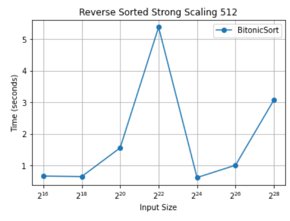
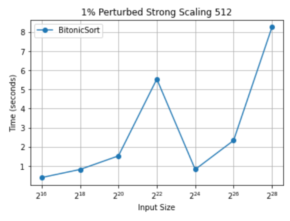
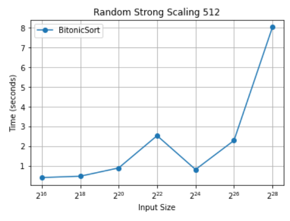
Weak Scaling CUDA

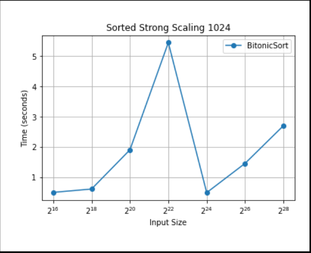
| Sorted | Reverse Sorted | 1% Perturbed | Random |
|---|---|---|---|
|  |  |  |  |

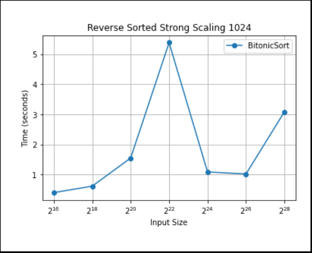


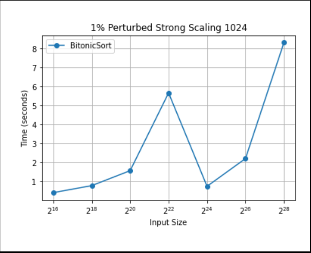
Comparing a change in the number of threads running a given input size, it's no surprise that over larger input sizes increasing threads produces faster runtimes. The threshold appears to be around 2^24 when a decrease in runtimes begins to appear as thread sizes increase, and this only gets more prevalent 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.

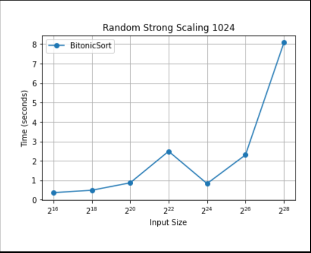
Strong Scaling CUDA

| Sorted | Reverse Sorted | 1% Perturbed | Random |
|---|---|---|---|
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |









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.