CSCE 435 Group project

0. Group number: 18

1. Group members:

1. Anna Hartman
2. Tate Moskal
3. Nicole Hernandez
4. Vasudev Agarwal

Our team will communicate primarily through Text message for real-time discussions and updates. We will also use GitHub to track code contributions and Trello for project management, ensuring transparency and accountability for task assignments.

2. Parallel Sorting Algorithms

2a. Brief project description

In this project, we will be comparing the efficiencies of different sorting algorithms running in parallel. The algorithms will be implemented and tested on parallel architectures, such as multi-core processors and possibly distributed systems. Each team member will focus on one of the following algorithms:

- Bitonic Sort: Anna Hartman
- Sample Sort: Vasudev Agarwal
- Merge Sort: Nicole Hernandez
- Radix Sort: Tate Moskal

The comparison will assess execution time, we will evaluate the efficiency on different input sizes (small, medium, large) to measure scalability, and analyzing how these algorithms leverage parallelism to improve sorting performance.

2b. Pseudocode for each parallel algorithm

2b.1 Bitonic Sort In Bitonic sort, a bitonic sequence is built (a sequence that first increases and then decreases) and sorted by merging. Within multiple threads, chunks of the given sequence are sorted into bitonic order. The merging process is carried out in parallel, merging each piece into a large bitonic sequence. Threads should be synchronized. The entire algorithm is carried out recursively in order to build the final bitonic sequence.

1. Divide the given array to be sorted into parallel chunks with corresponding threads
2. For each thread ands its piece (in parallel): sort the piece into bitonic order by recursively splitting the piece into two halves and sorting the first half into ascending order and the second into descending order, and then merging the two haves into a bitonic sequence.

3. Synchronize threads by ensuring each thread has completed its sorting before continuing on.
4. Merge all of the pieces bitonically (in parallel), for each chunk and thread: compare and swap such that if the current chunk is in the lower half, merge it in ascending order, and if it is in the upper half, merge it in descending order. Recursively merge the two halves of the sequence to ensure they are fully sorted in either ascending or descending order. This should be carried out log(array_size) times, as each chunk doubles in size after being recursively merged.
5. Output the result.

Actual Code Algorithm Description: 1. Initializes the MPI environment with MPI_Init, retrieving the rank of each process and the total number of processes using MPI_Comm_rank and MPI_Comm_size. The root process (rank 0) parses command-line arguments to determine the size of the array (as a power of 2) and the number of processes to be used. 2. Root process (rank 0) generates a random array of integers of size 2^exponent. This size is calculated from the exponent passed as an argument. The root process seeds the random number generator with the current time to ensure different data for each run. 3. Root process then divides the array into equal-sized chunks and distributes these chunks to all processes, including itself, using MPI_Scatter. Each process receives a local sub-array (of size total array size / number of processes) for sorting, ensuring that the sorting workload is spread evenly across all processes. 4. Once each process receives its portion of the array, it applies the Bitonic Sort algorithm locally. the Bitonic Sort works as follows: a. Recursively divides the sub-array into smaller parts. b. Sorts the smaller parts in ascending and descending order alternately. c. Merges these parts using the Bitonic Merge step, which compares and swaps elements in such a way that the sub-array becomes sorted. 5. After sorting their local sub-arrays, each process sends its sorted sub-array back to the root process using MPI_Gather. The root process collects all the sorted sub-arrays into the original array. 6. Once all sub-arrays are gathered, the root process performs a final Bitonic Sort on the entire array to merge the sorted sub-arrays into a fully sorted array. This final step is necessary because the sub-arrays are only partially sorted, and a global sort ensures that the entire array is in the correct order. 7. The root process then finally checks if the final array is correctly sorted by comparing each element with the next. If any element is out of order, it reports an error; otherwise, it confirms that the array is sorted. 8. Finally, the program shuts down the MPI environment with MPI_Finalize, cleaning up all resources used by MPI.

2b.2 Sample Sort In sample sort a large dataset is divided into smaller paritions, after which each partition is sorted independently, and then these sorted paritions are merged to obtain the final sorted result.

1. Splitting the given dataset into equally divided smaller segments, where each segement is given to a processor

2. Run basic sorting algorithm on each of the segments, where each processor is handling its piece independently.
3. From these sorted segments, each processor selects few samples. then, MPI communication is used to collect samples from all processors
4. Sort the selected samples, which will help us establish a global order among the samples.
5. From the sorted samples, we pick few speical elements. which act as a pivot. which are shared with all processors. MPI_Bcast is used to broadcast the pivots to all processors.
6. Each processor takes its ordered segment and divides it into subsegments based on the choicen pivot.
7. Now, processors share their ordered segments globally with the corresponding processor based on the segment number. using MPI_Alltoall
8. Finally, each processor merges and sorts the recieved elements.

2b.3 Merge Sort Parallel Merge Sort uses the divide and conquer technique, recursively dividing the dataset into smaller parts, sorting them, and merging the results where in parallel ver. it is distributed across multiple processors. Parallel Merge Sort will be implemented using MPI. Each processor will perform its own sorting idenpendently, merging sorted data using MPI communication.

1. Start MPI for Communication between processors.
2. Identify if the process is the master (rank 0) or worker (rank > 0)
3. The master splits the dataset into smaller parts
4. The master sends each part to a worker process
5. Each worker process sorts its part of the dataset independently
6. Workers send their sorted parts back to the master
7. The master merges all sorted parts into one sorted array
8. Close MPI after sorting is complete

2b.4 Radix Sort Radix Sort is an algorithm that sorts by processing through individual digits, sorting along the way. The process can be sped up by allowing each processor to handle a portion of the total array. By sorting a subarray and keeping note of the order of subarray chunks being sorted in each processor, they can be placed accordingly back into the main array. While this example sorts via binary, the process can account for numbers of any base as long as the # of arrays corresponds correctly.

Pseudocode: 1. Initialize MPI for multiprocessor communication 2. Convert array digits into binary (helps with initial implementation). 3. Find the maximum element and its # of digits. 5. Begin iterating through digits starting at the least significant digit up to the maximum digit significance. 7. Split the array into subarrays depending on the # of processors used and send to workers, keeping track of the order in which each subarray gets sent where. 9. Each worker will sort its subarray into 2 arrays, the first with digits that are 0, the second with digits that are 1. 10. Worker returns arrays and master combines

the 0 array in order of worker process, then repeats for the 1 array. 11. Repeat with the next digit until all digits places have been parsed. 12. End MPI once complete.

Coded Algorithm (Differs from pseudocode as this version more optimally parallelizes Radix Sort): 1. Initializes MPI environment with MPI_Init(), MPI_Comm_size(), and MPI_Comm_rank(). 2. Master process at rank 0 creates array based on command line arguments. 3. Total array size is broadcasted to all processes with MPI_Bcast(). 4. Total array is split into subarrays based on # of processers and distributed amongst the processors with MPI_Scatter(). 5. Each process finds the maximum value of its subarray in base 2 as well as the amount of digits it has. 6. Each process sorts its subarray by digit group in countSort(). A histogram counts the occurances of each digit group in the subarray. Using the histogram and summations, it sorts the local subarray and redistributes the info to the rest of the processes. 7. Arrays are gathered together, exchanging information and parts of subarrays to globally sort the total array. 8. Using MPI_Gatherv() All subarrays are collected in the master. 9. The total array is checked if it is sorted and prints the result. 10. Memory is deallocated and MPI_Finalize() ends the MPI environment.

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

- Input sizes, Input types (Alter input array to be: random, sorted, reverse sorted,…)
- Strong scaling (same problem size, increase number of processors/nodes)
- Weak scaling (increase problem size, increase number of processors)
- Parallelization strategies (master/worker vs SPMD, calculating speedup and runtime differences)
- Communication strategies (collectives vs point-to-point, measure runtime differences between code for each communication strategy)

3a. Caliper

Merge Sort Calltree

```
0.504 main
  0.000 MPI_Init
  0.003 MPI_Bcast
  0.000 comm
    0.000 MPI_Scatter
    0.000 comm_large
      0.000 MPI_Gather
  0.000 comp
    0.000 comp_large
  0.000 MPI_Finalize
  0.000 MPI_Initialized
  0.000 MPI_Finalized
```

```
  0.000 MPI_Comm_dup
```

Sample Sort Calltree

```
0.271 main
  0.000 data_init_runtime
  0.267 MPI_Init
    0.000 MPI_Init
  0.003 comm
    0.001 comm_scatter
      0.000 MPI_Scatter
    0.000 comm_gather_samples
      0.000 MPI_Gather
    0.001 comm_bcast_pivots
      0.001 MPI_Bcast
    0.001 comm_all_to_all
      0.001 MPI_Alltoall
    0.000 comm_alltoallv
      0.000 MPI_Alltoallv
    0.000 final_gather
       0.000 MPI_Gatherv
  0.000 comp
    0.000 local_sort
    0.000 pivot_sort
    0.000 pivot_partition
    0.000 merge_elements
  0.000 MPI_Barrier
  0.000 MPI_Gather
  0.000 correctness_check
  0.000 MPI_Finalize
  0.000 MPI_Initialized
  0.000 MPI_Finalized
  0.001 MPI_Comm_dup
```

Radix Sort Calltree

```
2.722 main
  0.000 MPI_Init
  0.051 data_init_runtime
  0.852 comm
    0.825 comm_large
      0.087 MPI_Bcast
      0.055 MPI_Allgather
      0.680 MPI_Gatherv
    0.024 comm_small
       0.013 MPI_Scatter
```

```
      0.001 MPI_Gather
      0.006 MPI_Bcast
      0.003 MPI_Gatherv
  0.045 comp
    0.000 comp_small
    0.044 comp_large
  0.087 MPI_Barrier
  0.049 correctness_check
  0.000 MPI_Finalize
  0.000 MPI_Initialized
  0.000 MPI_Finalized
  0.004 MPI_Comm_dup
```

Bitonic Sort Calltree

```
4.972 main
  0.000 MPI_Init
  2.803 comp
     2.803 comp_large
  2.169 comm
    2.169 comm_large
       2.163 MPI_Scatter
       0.007 MPI_Gather
  0.000 correctness_check
  0.000 MPI_Init
  0.000 MPI_Finalize
  0.000 MPI_Finalized
  0.000 MPI_Initialized
  24.052 MPI_Comm_dup
```

3b. Collect Metadata

Merge Sort Metadata

```
           cali.caliper.version  mpi.world.size  \
profile
3133824840               2.11.0               2

                                                    spot.metrics  \
profile
3133824840  min#inclusive#sum#time.duration,max#inclusive#...

           spot.timeseries.metrics  spot.format.version  \
profile
3133824840                                            2

                                                    spot.options  spot.channels  \
```

```
profile
3133824840  time.variance,profile.mpi,node.order,region.co...  regionprofile

           cali.channel spot:node.order   spot:output spot:profile.mpi  \
profile
3133824840         spot            true  p2-a128.cali             true

           spot:region.count spot:time.exclusive spot:time.variance algorithm  \
profile
3133824840              true                true               true     merge

           programming_model  group_num  input_size  num_procs data_type  \
profile
3133824840               mpi         18         128          2    random

            size_of_data_type scalability
profile
3133824840                 4      strong
```

Sample Sort Metadata

```
   cali.caliper.version  mpi.world.size  \
profile
3133824840              2.11.0              2

                                              spot.metrics  \
profile
3133824840  min#inclusive#sum#time.duration,max#inclusive#...

           spot.timeseries.metrics  spot.format.version  \
profile
3133824840                                           2

                                             spot.options  spot.channels  \
profile
3133824840  time.variance,profile.mpi,node.order,region.co...  regionprofile

           cali.channel spot:node.order   spot:output spot:profile.mpi  \
profile
3133824840         spot            true  p2-a128.cali             true

           spot:region.count spot:time.exclusive spot:time.variance  \
profile
3133824840              true                true               true

              algorithm programming_model  group_num  input_size  num_procs  \
```

```
profile
3133824840  Sample Sort               mpi         18         128          2

           data_type  size_of_data_type scalability
profile
3133824840    random                  4      strong
```

Radix Sort Metadata

```
           cali.caliper.version  mpi.world.size  \
profile
3133824840               2.11.0               2

                                                    spot.metrics  \
profile
3133824840  min#inclusive#sum#time.duration,max#inclusive#...

           spot.timeseries.metrics  spot.format.version  \
profile
3133824840                                          2

                                                    spot.options  spot.channels  \
profile
3133824840  time.variance,profile.mpi,node.order,region.co...  regionprofile

           cali.channel spot:node.order   spot:output spot:profile.mpi  \
profile
3133824840         spot            true  p2-a128.cali             true

           spot:region.count spot:time.exclusive spot:time.variance  \
profile
3133824840              true                true               true

            launchdate                                          libraries  \
profile
3133824840  1729137630  [/scratch/group/csce435-f24/Caliper/caliper/li...

                                cmdline cluster algorithm programming_model  \
profile
3133824840  [./radix_sort, --size, 128]       c     radix               mpi

           data_type  size_of_data_type  input_size input_type  num_procs  \
profile
3133824840       int                  4         128     Random          2

           scalability  group_num implementation_source
```

```
profile
3133824840      strong         18                online
```

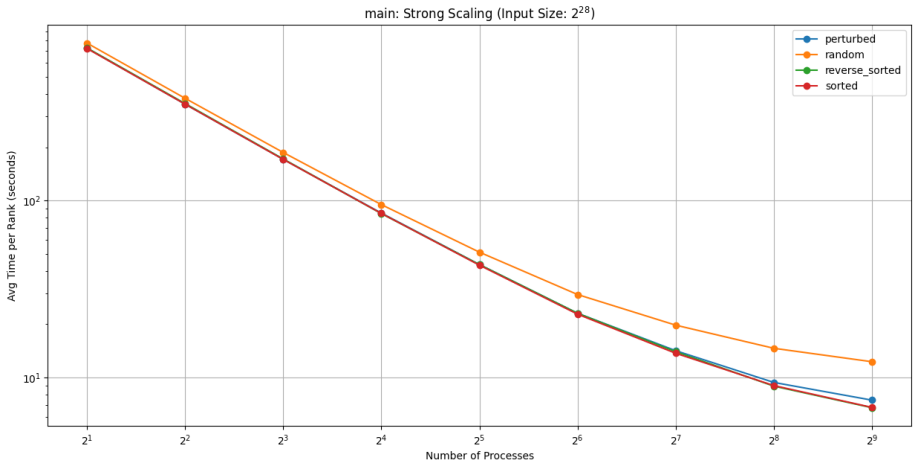
Bitonic Sort Metadata

```
cali.caliper.version  mpi.world.size  \
profile
1981606483               2.11.0             16

                                                spot.metrics  \
profile
1981606483  min#inclusive#sum#time.duration,max#inclusive#...

           spot.timeseries.metrics  spot.format.version  \
profile
1981606483                                          2

                                                spot.options  spot.channels  \
profile
1981606483  time.variance,profile.mpi,node.order,region.co...  regionprofile

           cali.channel spot:node.order   spot:output spot:profile.mpi  \
profile
1981606483         spot            true  p16-a16.cali             true

           spot:region.count spot:time.exclusive spot:time.variance  \
profile
1981606483              true                true               true

            launchdate                                          libraries  \
...

           scalability  group_num implementation_source
profile
1981606483      weak          1           online
```

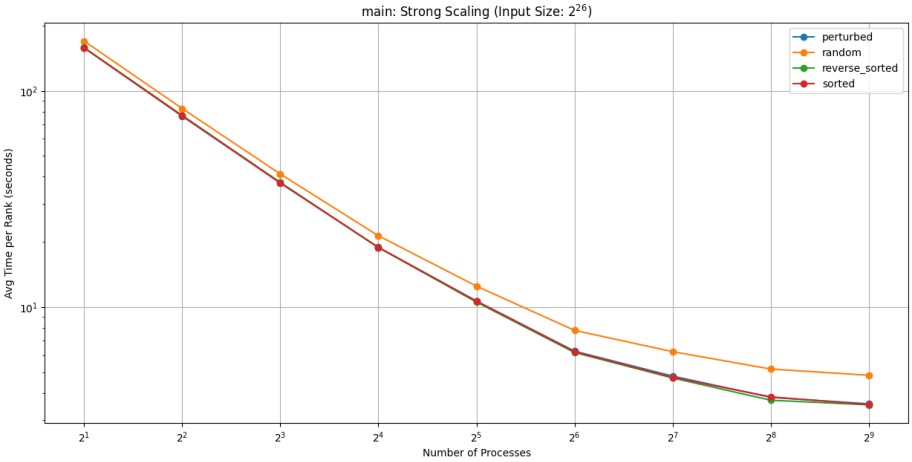
4. Performance evaluation

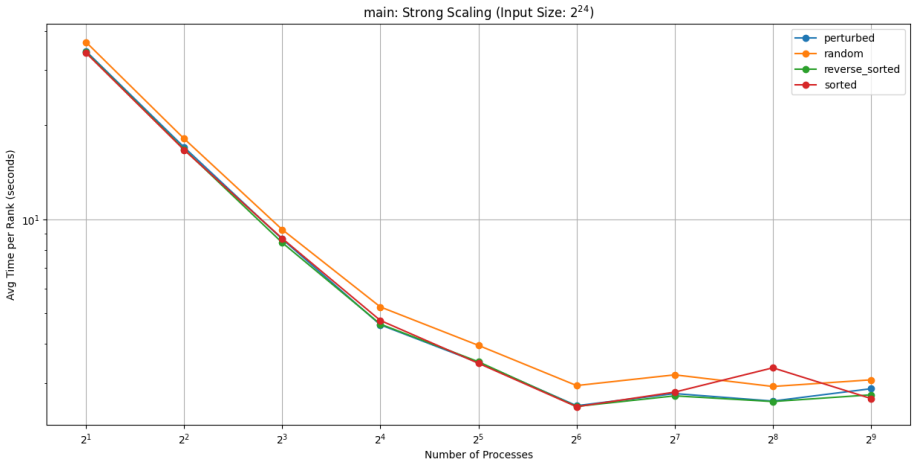
4a. Bitonic Sort

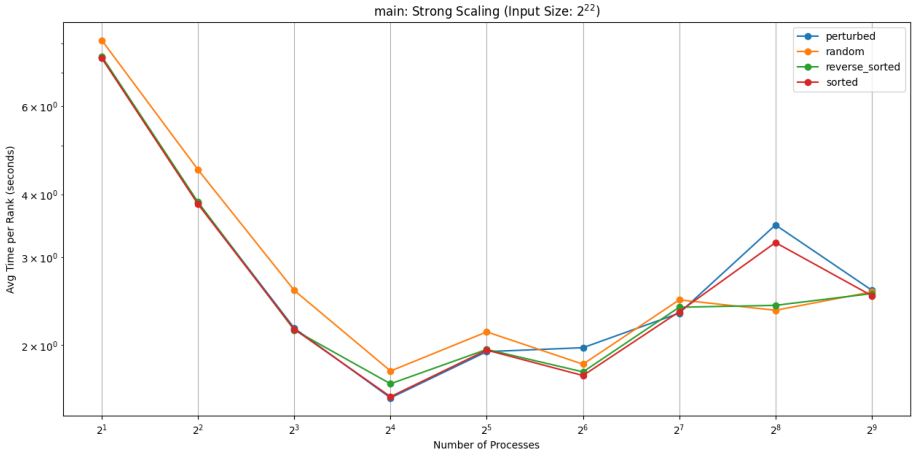
Strong Scaling

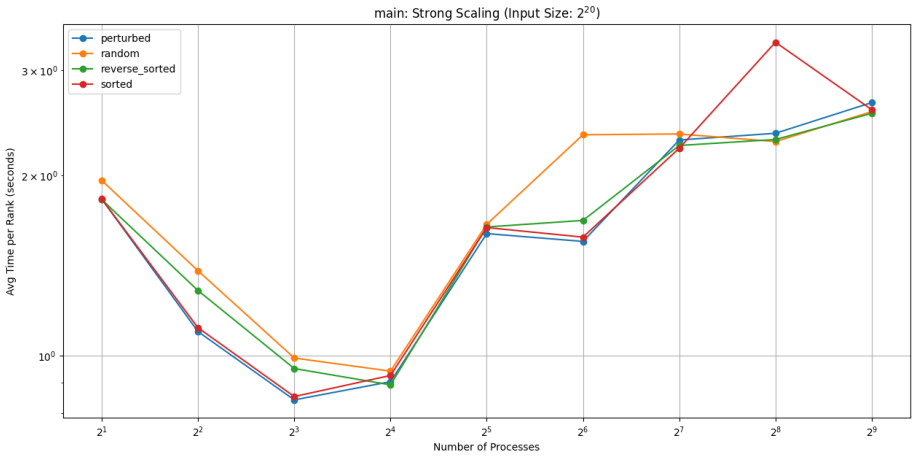


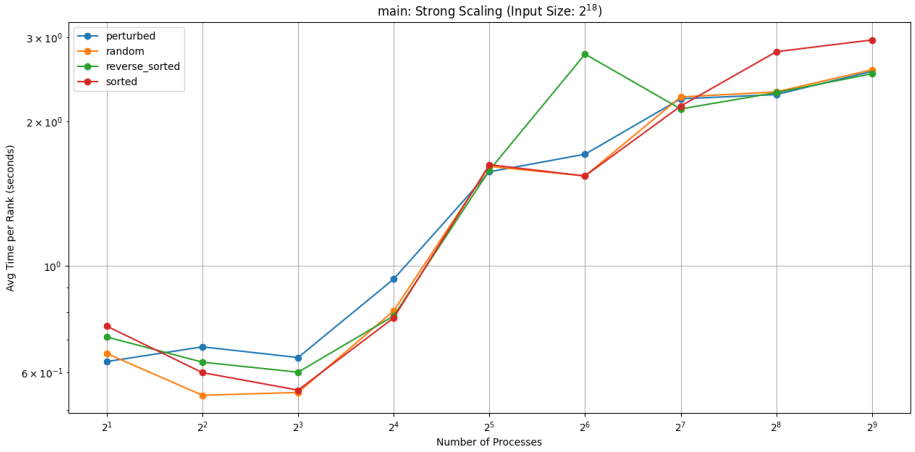
Main

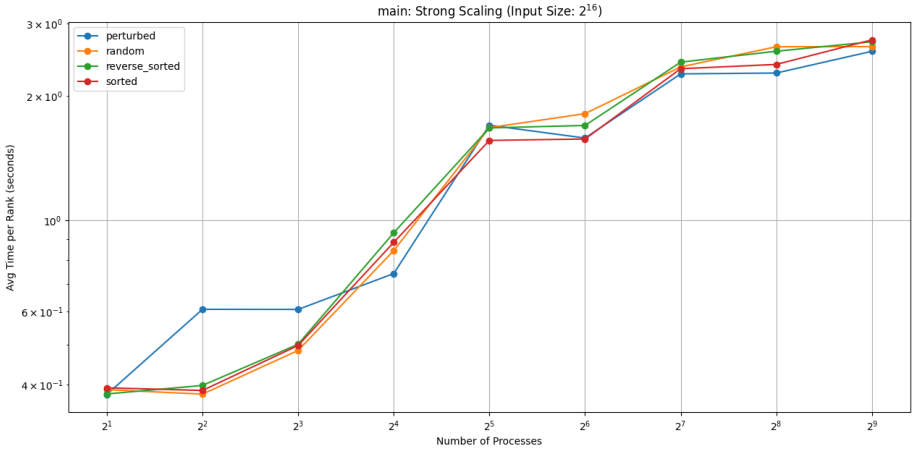


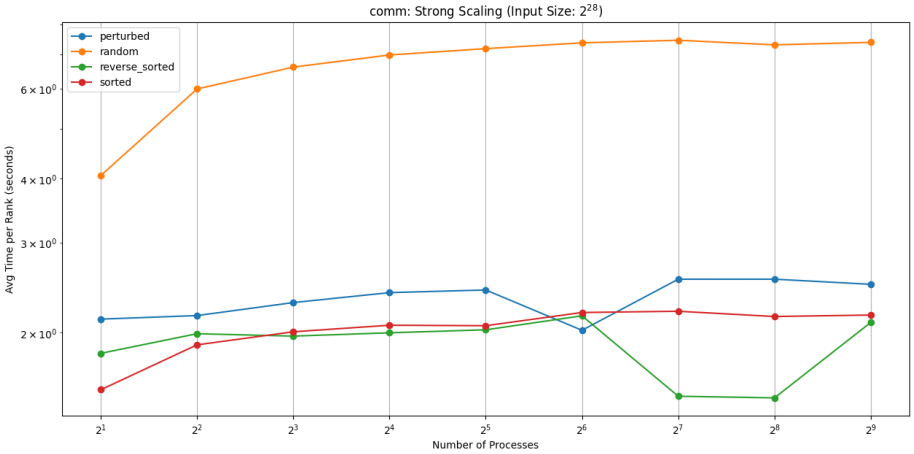

main: Strong Scaling (Input Size: 2^{24})
perturbed
random
reverse_sorted
sorted
Avg Time per Rank (seconds)
10^1
2^1 2^2 2^3 2^4 2^5 2^6 2^7 2^8 2^9
Number of Processes


main: Strong Scaling (Input Size: 2^{22})
perturbed
random
reverse_sorted
sorted
Avg Time per Rank (seconds)
6×10^0
4×10^0
3×10^0
2×10^0
2^1 2^2 2^3 2^4 2^5 2^6 2^7 2^8 2^9
Number of Processes

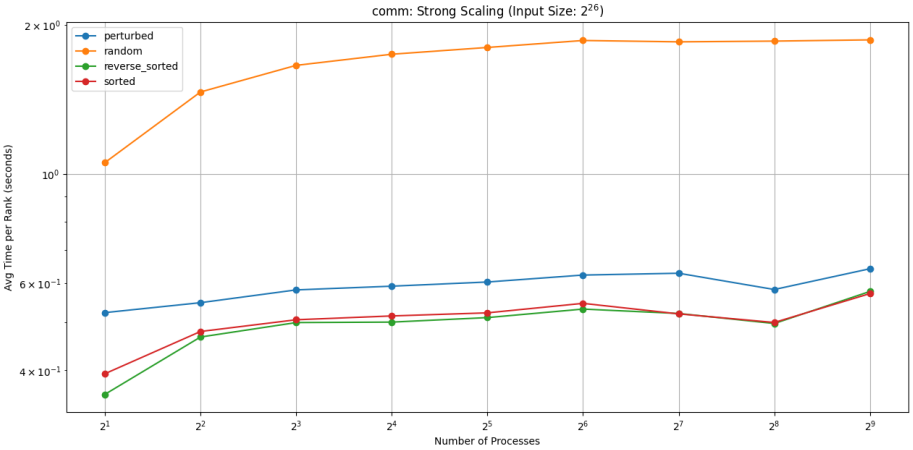

main: Strong Scaling (Input Size: 2^{20})
perturbed
random
reverse_sorted
sorted
Avg Time per Rank (seconds)
3×10^0
2×10^0
10^0
2^1 2^2 2^3 2^4 2^5 2^6 2^7 2^8 2^9
Number of Processes

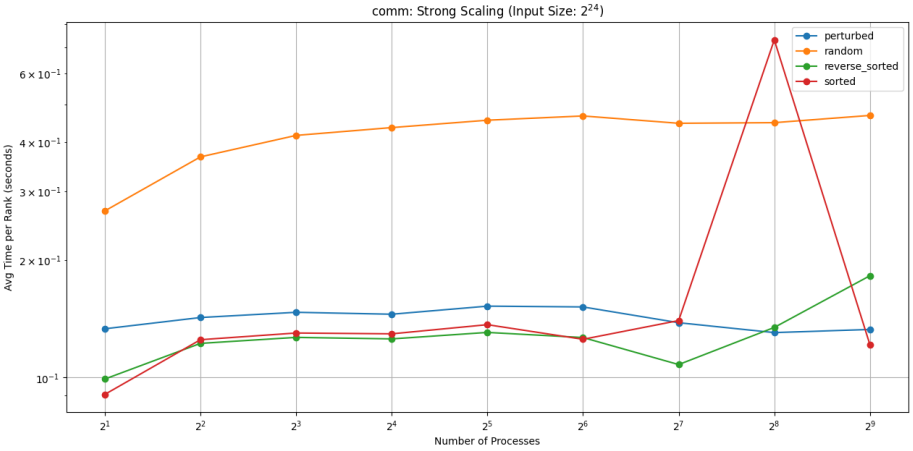

main: Strong Scaling (Input Size: 2^{18})
perturbed
random
reverse_sorted
sorted
Avg Time per Rank (seconds)
3×10^0
2×10^0
10^0
6×10^{-1}
2^1 2^2 2^3 2^4 2^5 2^6 2^7 2^8 2^9
Number of Processes

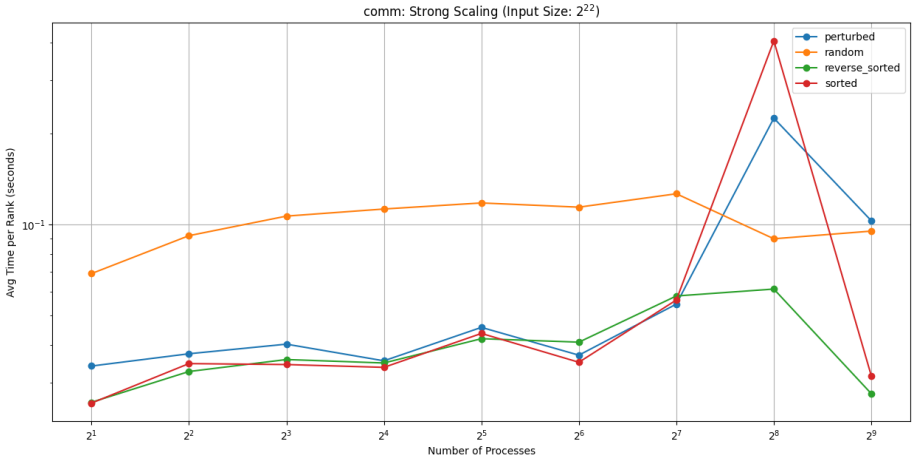

main: Strong Scaling (Input Size: 2^{16})
perturbed
random
reverse_sorted
sorted
Avg Time per Rank (seconds)
3×10^0
2×10^0
10^0
6×10^{-1}
4×10^{-1}
2^1 2^2 2^3 2^4 2^5 2^6 2^7 2^8 2^9
Number of Processes

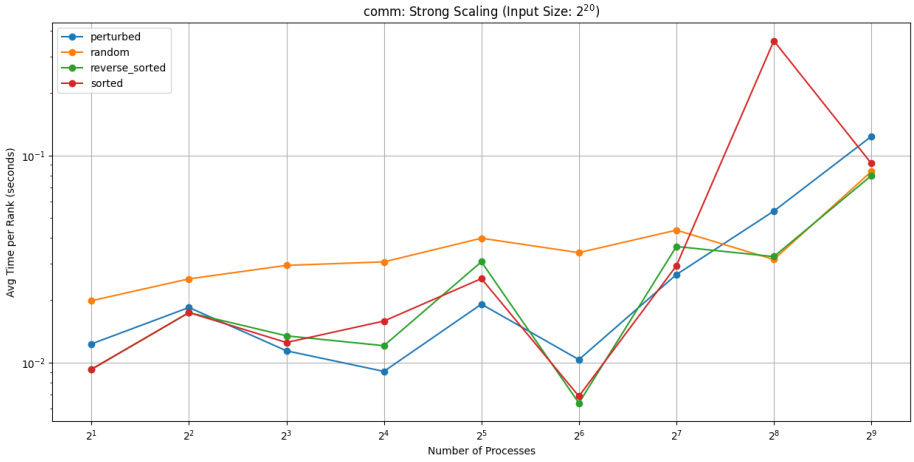

comm: Strong Scaling (Input Size: 2^{28})
perturbed
random
reverse_sorted
sorted
Avg Time per Rank (seconds)
6×10^0
4×10^0
3×10^0
2×10^0
2^1 2^2 2^3 2^4 2^5 2^6 2^7 2^8 2^9
Number of Processes

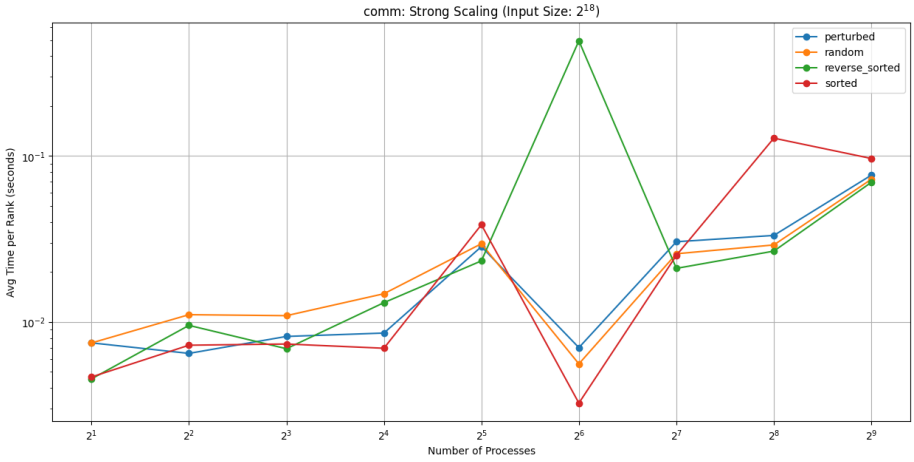
Comm

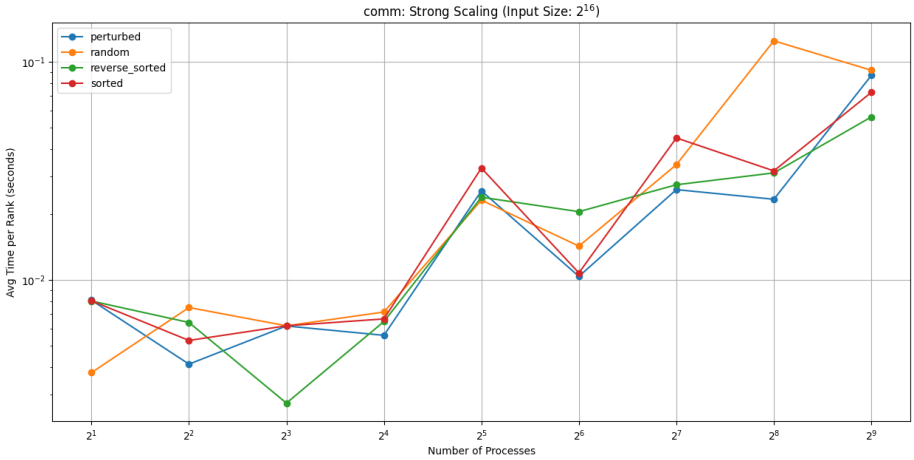

comm: Strong Scaling (Input Size: 2^{26})
perturbed
random
reverse_sorted
sorted
Avg Time per Rank (seconds)
2×10^0
10^0
6×10^{-1}
4×10^{-1}
2^1 2^2 2^3 2^4 2^5 2^6 2^7 2^8 2^9
Number of Processes

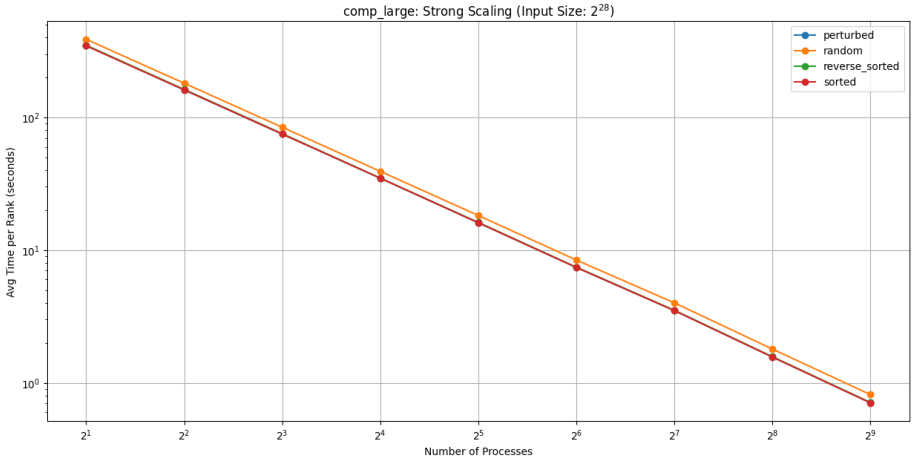

comm: Strong Scaling (Input Size: 2^{24})
perturbed
random
reverse_sorted
sorted
Avg Time per Rank (seconds)
6×10^{-1}
4×10^{-1}
3×10^{-1}
2×10^{-1}
10^{-1}
2^1 2^2 2^3 2^4 2^5 2^6 2^7 2^8 2^9
Number of Processes


comm: Strong Scaling (Input Size: 2^{22})
perturbed
random
reverse_sorted
sorted
Avg Time per Rank (seconds)
10^{-1}
2^1 2^2 2^3 2^4 2^5 2^6 2^7 2^8 2^9
Number of Processes

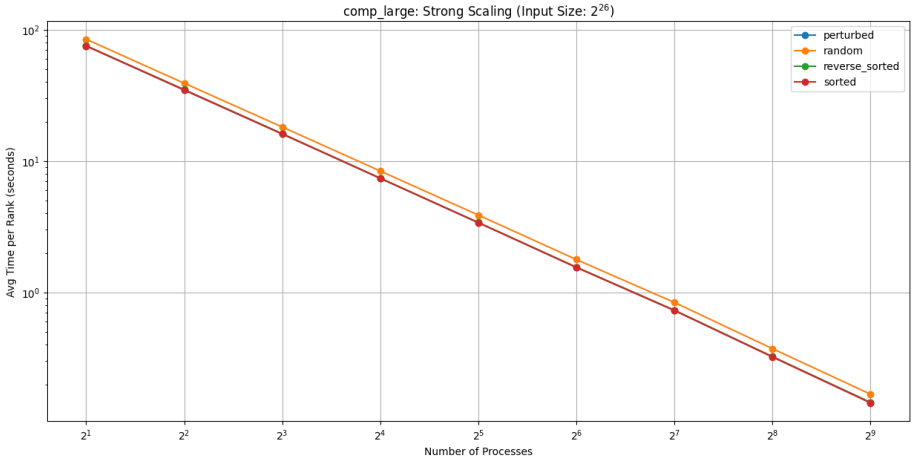

comm: Strong Scaling (Input Size: 2^{20})
perturbed
random
reverse_sorted
sorted
Avg Time per Rank (seconds)
10^{-1}
10^{-2}
2^1 2^2 2^3 2^4 2^5 2^6 2^7 2^8 2^9
Number of Processes

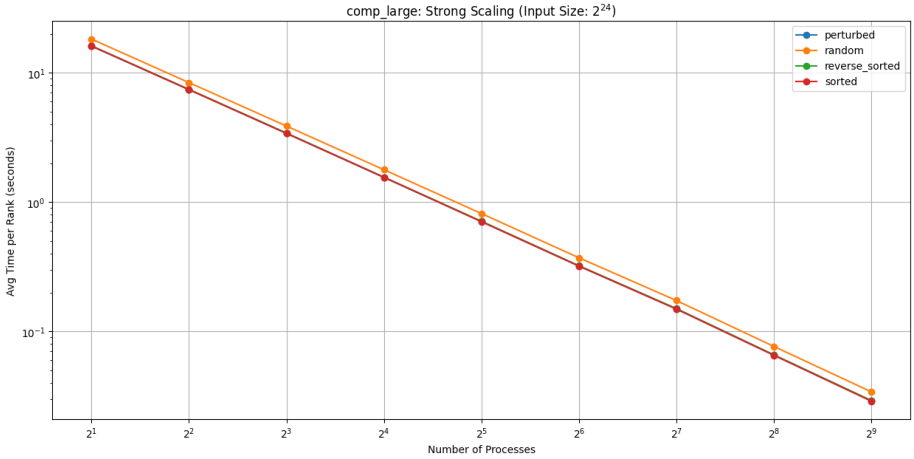

comm: Strong Scaling (Input Size: 2^{18})
perturbed
random
reverse_sorted
sorted
Avg Time per Rank (seconds)
10^{-1}
10^{-2}
2^1 2^2 2^3 2^4 2^5 2^6 2^7 2^8 2^9
Number of Processes

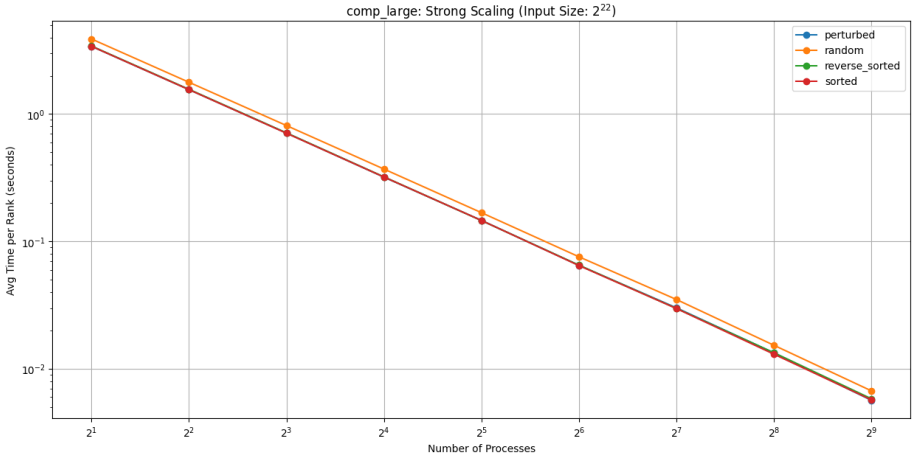

comm: Strong Scaling (Input Size: 2^{16})
perturbed
random
reverse_sorted
sorted
Avg Time per Rank (seconds)
Number of Processes

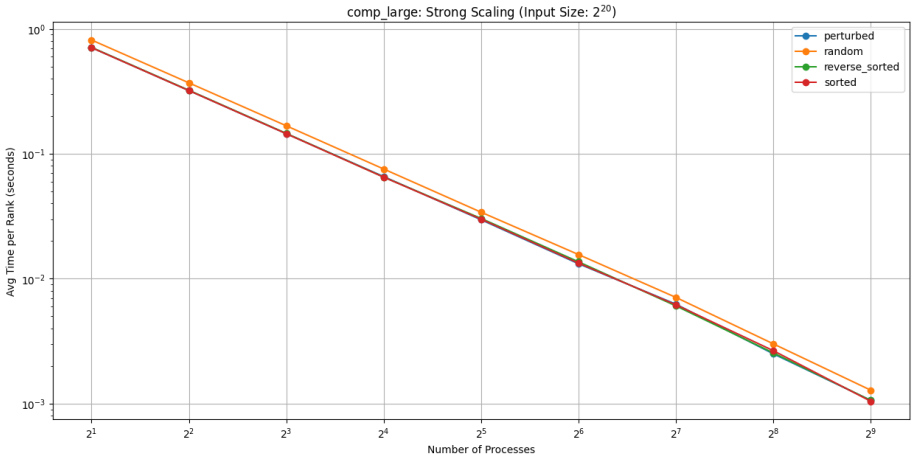

comp_large: Strong Scaling (Input Size: 2^{28})
perturbed
random
reverse_sorted
sorted
Avg Time per Rank (seconds)
Number of Processes

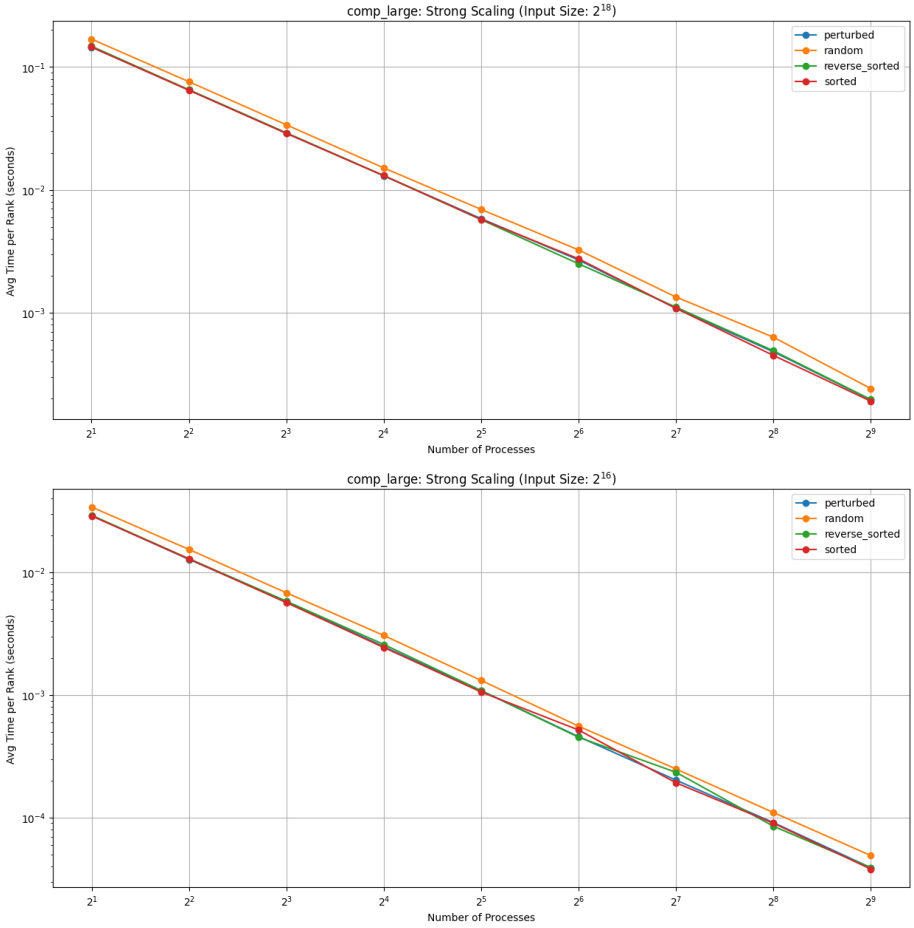
Comp Large


comp_large: Strong Scaling (Input Size: 2^{26})
perturbed
random
reverse_sorted
sorted
Avg Time per Rank (seconds)
Number of Processes


comp_large: Strong Scaling (Input Size: 2^{24})
perturbed
random
reverse_sorted
sorted
Avg Time per Rank (seconds)
Number of Processes


comp_large: Strong Scaling (Input Size: 2^{22})
perturbed
random
reverse_sorted
sorted
Avg Time per Rank (seconds)
Number of Processes

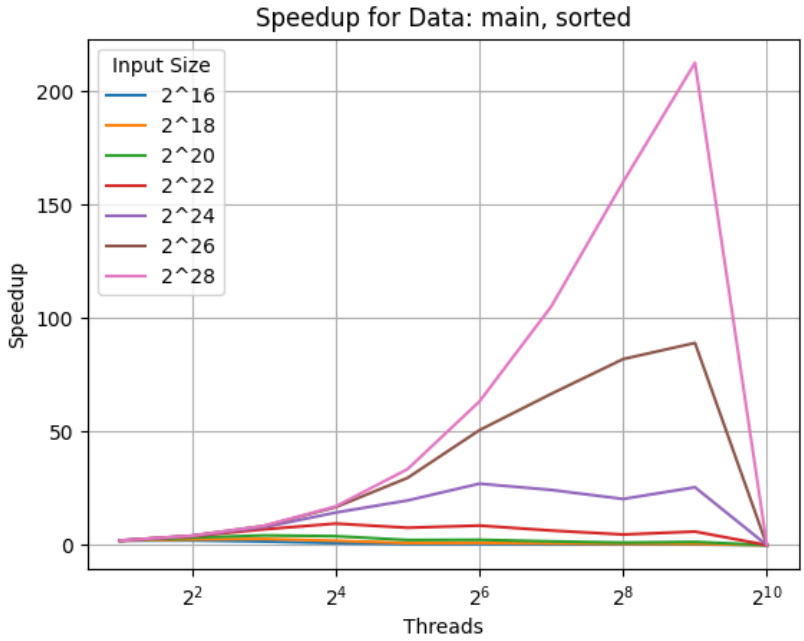

comp_large: Strong Scaling (Input Size: 2^{20})
perturbed
random
reverse_sorted
sorted
Avg Time per Rank (seconds)
Number of Processes



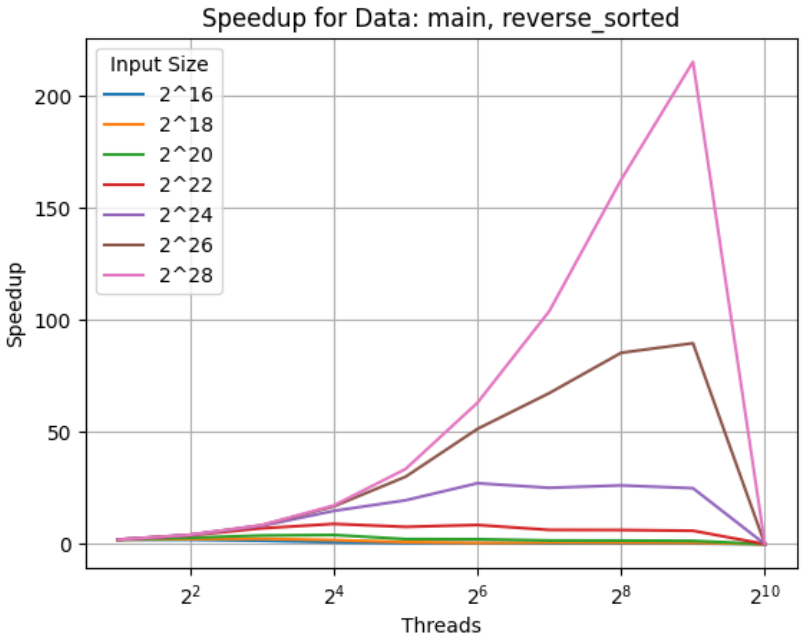
Analysis: The strong scaling plots show how the execution time decreases as the number of processors increases while keeping the problem size constant for sizes 2^16 through 2^28. Ideally, the execution time should decrease proportionally with the increase in the number of processors. With these plots, the strong scaling curve shows diminishing returns as more processors are added. This is due to overhead from inter-processor communication, synchronization, and non-parallelizable portions of the algorithm (as per Amdahl's law). It is worth noting that the time reduces much more linearly with large problem sizes, meaning that parallelization is more effective and has more returns with a larger problem size. For larger problem sizes, the actual computational work increases significantly, and the overhead becomes a smaller fraction of the total work. The efficiency of parallelization improves with larger problem sizes, because the computational gains outweigh the overhead. Another observation is that the communication time is overall much higher for random inputs than for the other types of inputs (as seen in the Comm plots). This could be because random inputs generally result in more frequent and scattered data transfers

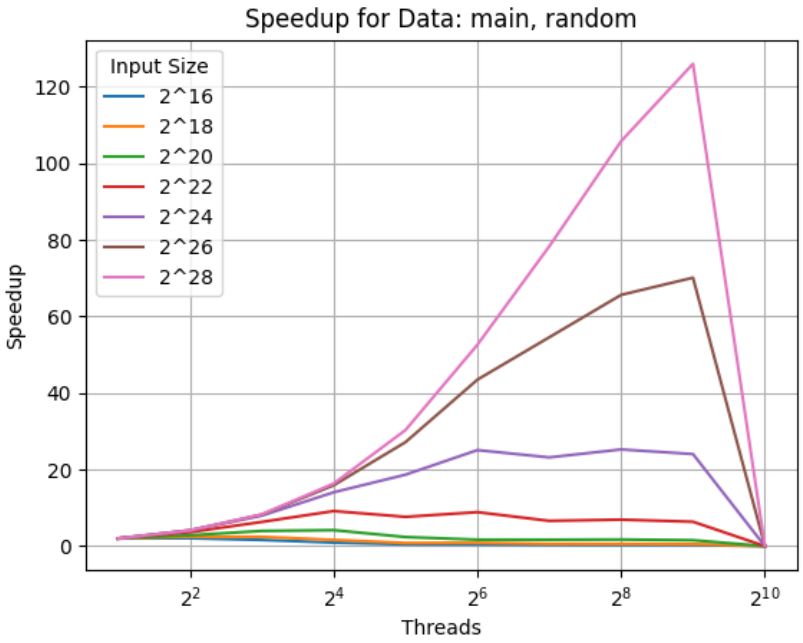
between processors, which increases the need for synchronization. Each time processors exchange data, they must often wait for each other to complete their communication before proceeding to the next step. This synchronization could introduce delays, especially when communication patterns are irregular and unpredictable, as with random data. For the comp_large graphs, it is clear that the times decreased linearly consistently with the increase in the number of processes. The underlying bitonic algorithm used for comp_large scales well with the number of processes. The computational workload increases enough to offset the overhead of managing multiple processes, as with comp_large, where the algorithm is designed to take full advantage of the parallel architecture, minimizing any bottlenecks that might otherwise prevent linear scaling.

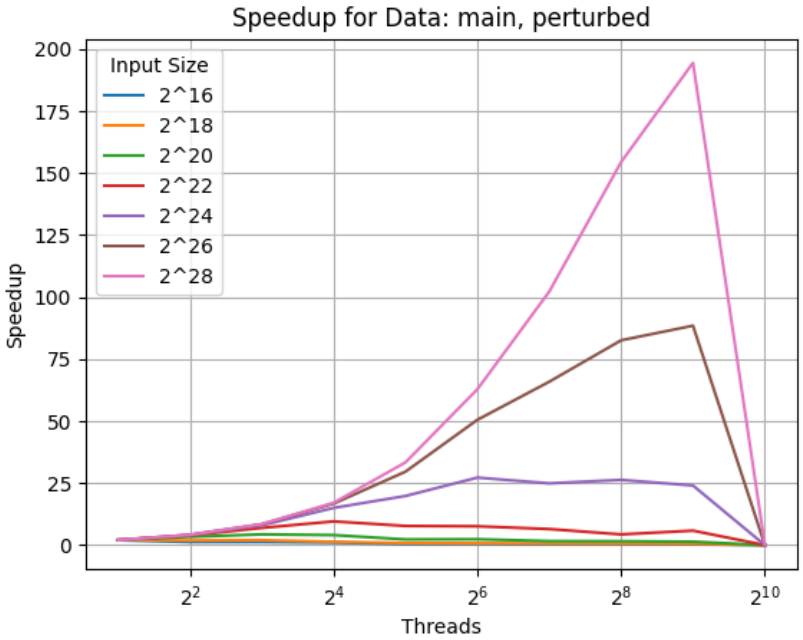
Strong Speed-Up

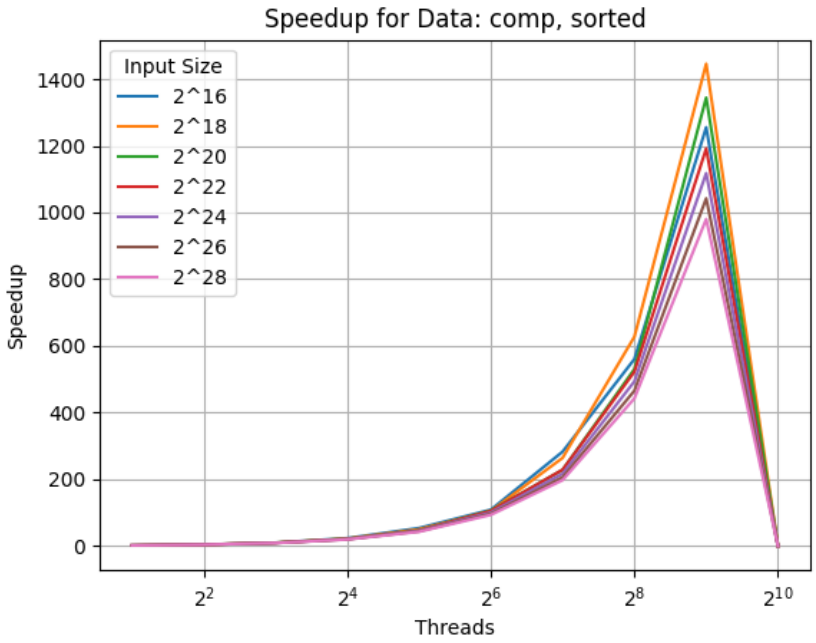


Main

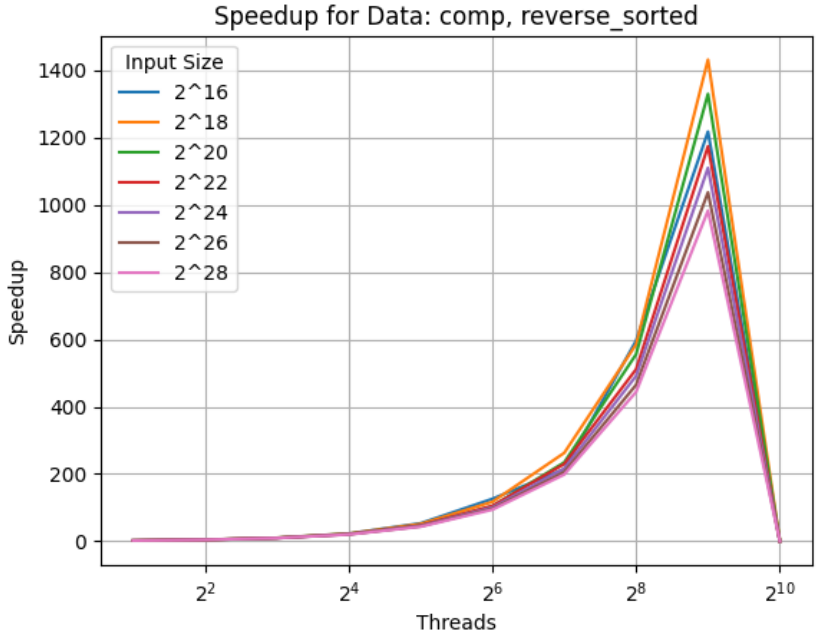


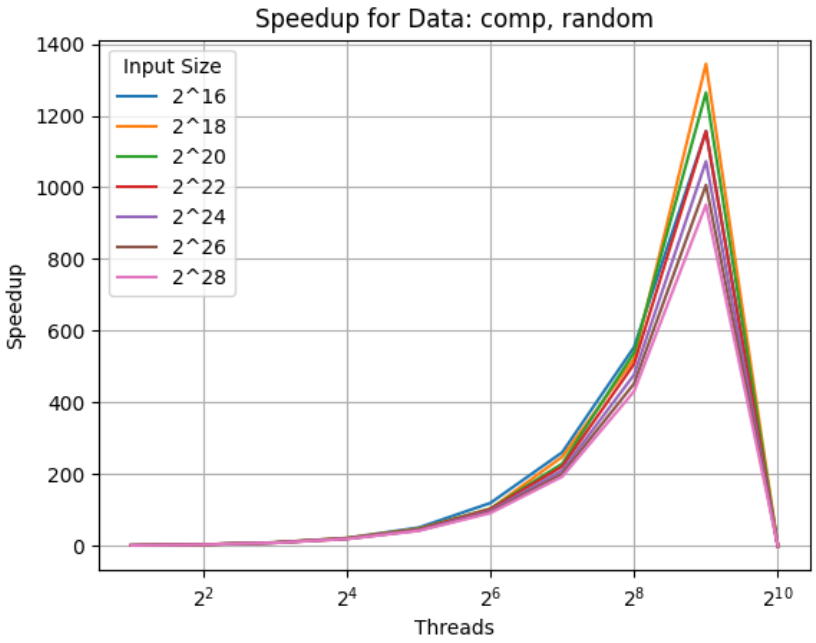

Speedup for Data: main, random
Input Size
2^16
2^18
2^20
2^22
2^24
2^26
2^28
Speedup
Threads
0
20
40
60
80
100
120
2^2
2^4
2^6
2^8
2^{10}

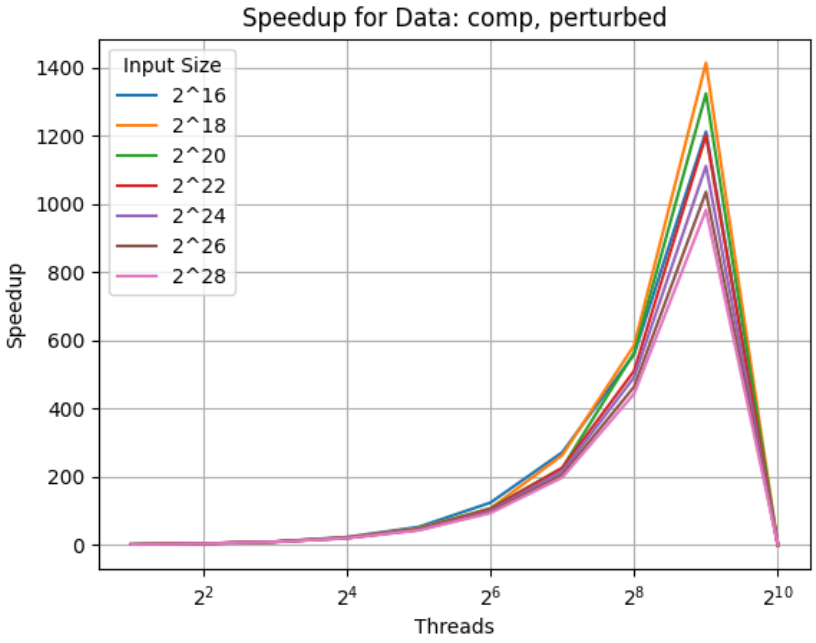

Speedup for Data: main, perturbed
Input Size
2^16
2^18
2^20
2^22
2^24
2^26
2^28
Speedup
Threads
0
25
50
75
100
125
150
175
200
2^2
2^4
2^6
2^8
2^{10}

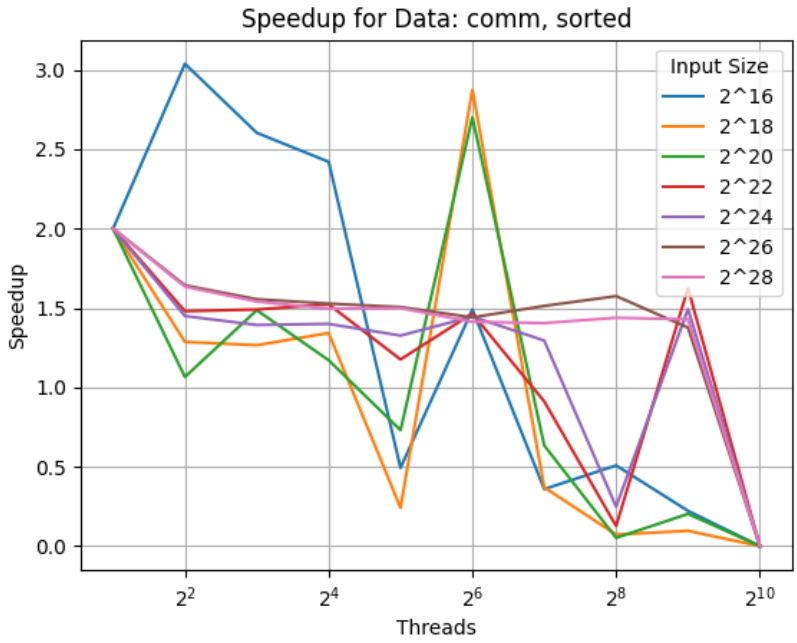

Speedup for Data: comp, sorted
Input Size
2^16
2^18
2^20
2^22
2^24
2^26
2^28
Speedup
Threads
0
200
400
600
800
1000
1200
1400
2^2
2^4
2^6
2^8
2^{10}

Comp

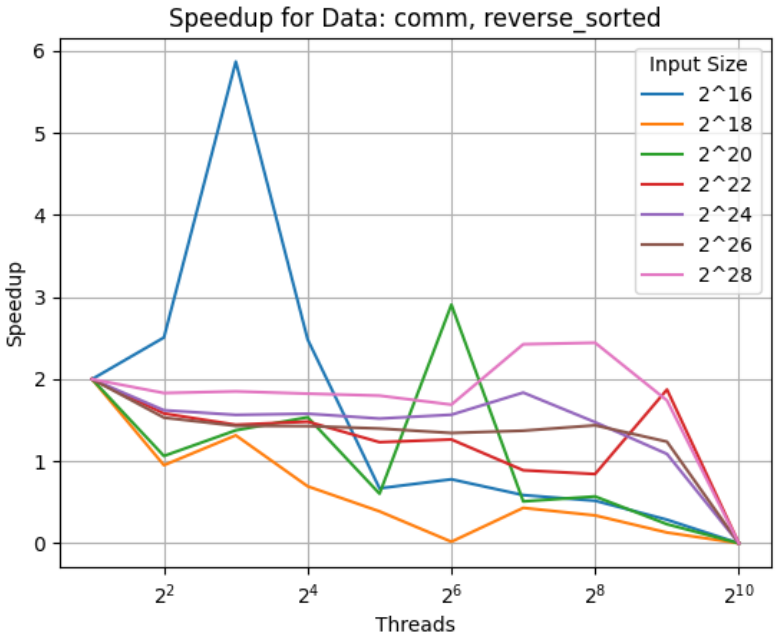

Speedup for Data: comp, reverse_sorted
Input Size
2^16
2^18
2^20
2^22
2^24
2^26
2^28
Speedup
Threads
0
200
400
600
800
1000
1200
1400
2^2
2^4
2^6
2^8
2^{10}

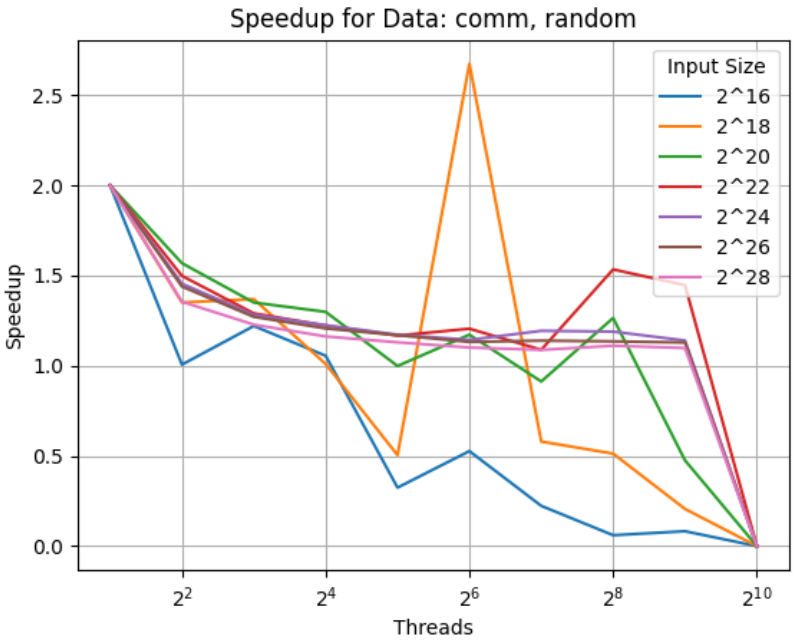

Speedup for Data: comp, random
Input Size
2^16
2^18
2^20
2^22
2^24
2^26
2^28
Speedup
Threads
0
200
400
600
800
1000
1200
1400
2^2
2^4
2^6
2^8
2^{10}

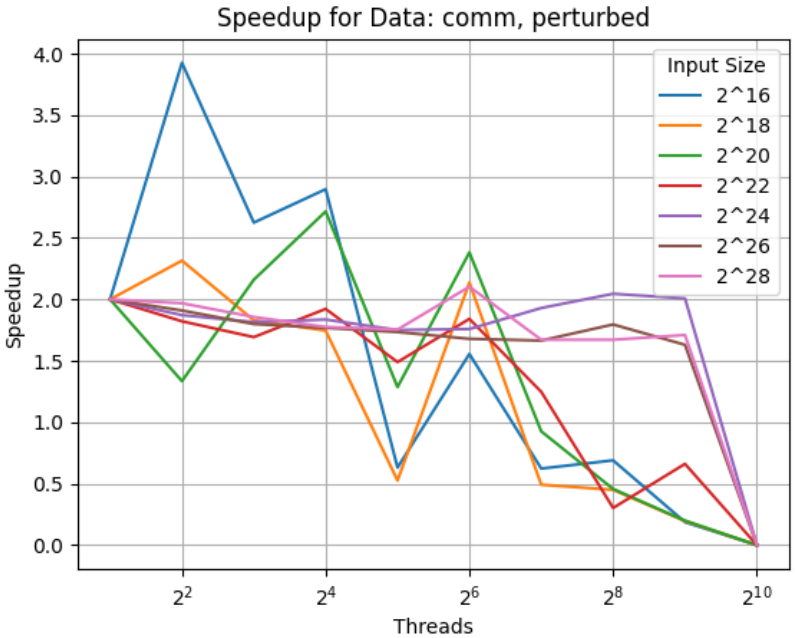

Speedup for Data: comp, perturbed
Input Size
2^16
2^18
2^20
2^22
2^24
2^26
2^28
Speedup
Threads
0
200
400
600
800
1000
1200
1400
2^2
2^4
2^6
2^8
2^{10}


Speedup for Data: comm, sorted
Input Size
2^16
2^18
2^20
2^22
2^24
2^26
2^28
Speedup
Threads
3.0
2.5
2.0
1.5
1.0
0.5
0.0
2^2
2^4
2^6
2^8
2^{10}

Comm


Speedup for Data: comm, reverse_sorted
Input Size
2^16
2^18
2^20
2^22
2^24
2^26
2^28
Speedup
Threads
6
5
4
3
2
1
0
2^2
2^4
2^6
2^8
2^{10}

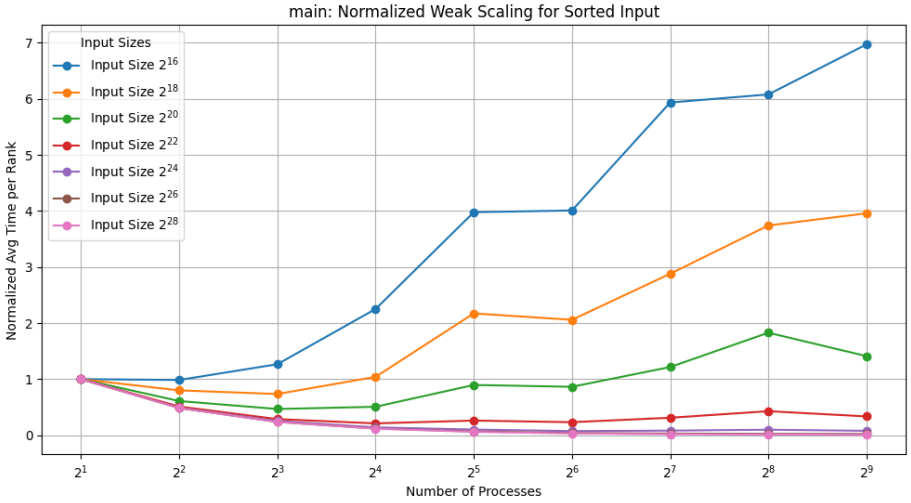


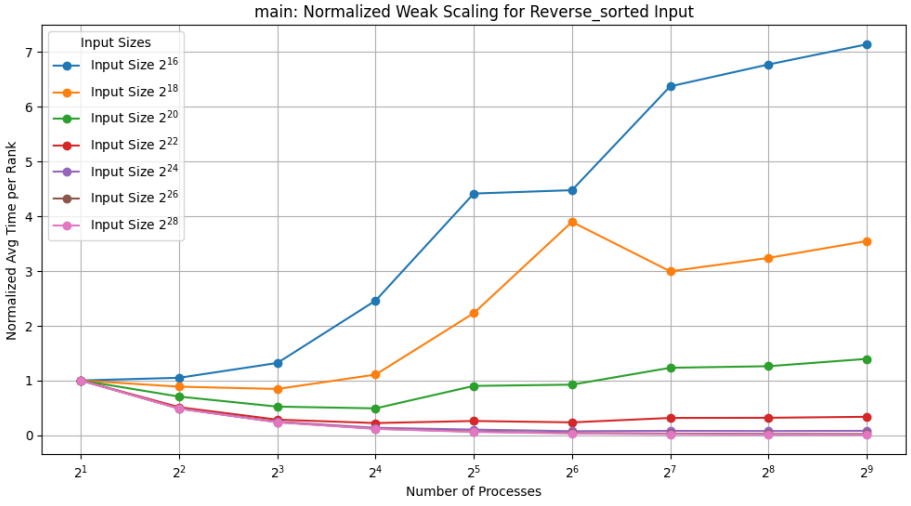


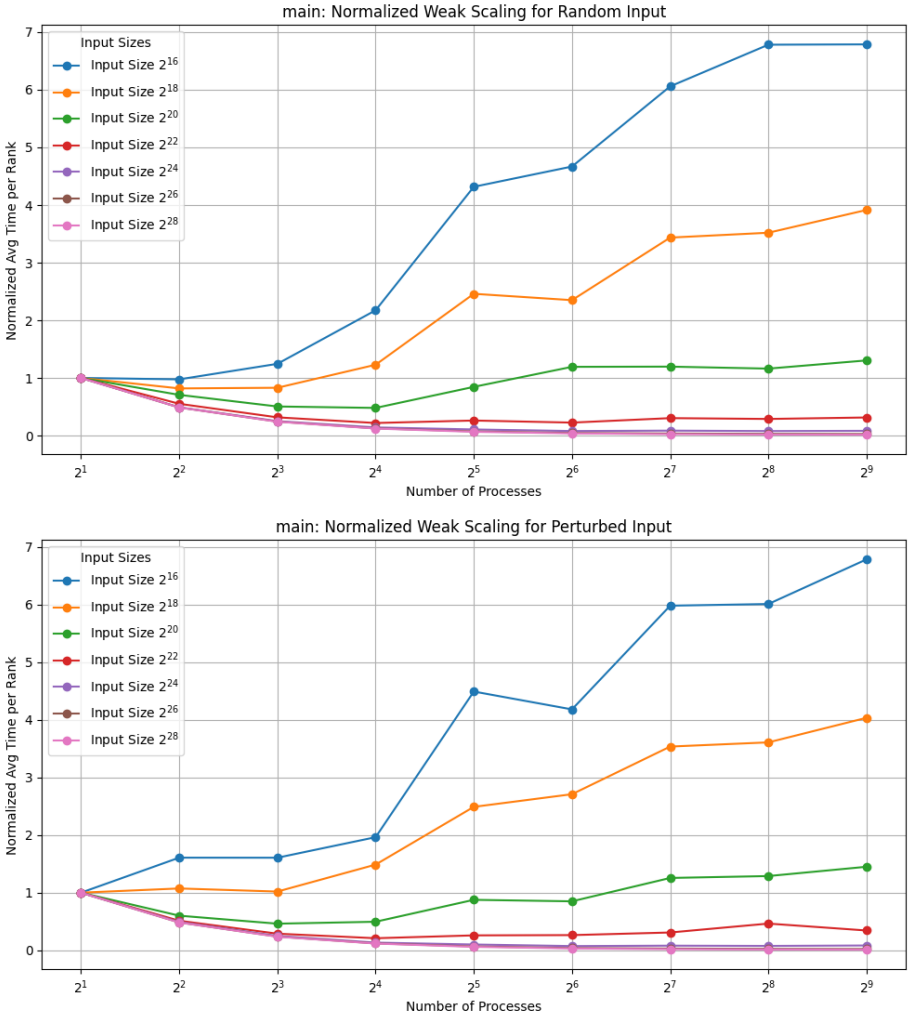
Analysis: It is clear from the plots that the speedup peaks at around 256 threads for all of the types of inputs and problem sizes, though it is the highest for the

larger problem sizes. The decrease in speedup is likely due to growing communication overhead, as more processes require more data exchanges, and synchronization delays increase. Amdahl's Law plays a role by limiting the speedup as non-parallelizable parts of the algorithm, like merging and comparing, become bottlenecks. Load imbalances between processes, increased cache contention, and memory bandwidth issues could also further reduce efficiency. The communication overhead can start to dominate the benefits of parallelization, especially as the number of processes grows. Therefore, the speedup peaks at a certain number of processes before the overhead becomes too great. Up until then, the added parallelization with more proceses speeds up the time for the algorithm.

Weak Scaling







Analysis: It is observed that the average time per rank increases for the smaller problem sizes and decreases for the larger problem sizes. For smaller problem sizes, the communication overhead becomes more significant compared to the actual computation each process has to perform. Bitonic sort involves frequent communication between processes, so smaller inputs don't have enough computational work to offset the communication time. As a result, the average time per rank increases because the overhead dominates the computation. But for larger problem sizes, the amount of computation per process increases significantly, which helps amortize the communication overhead. The processes spend more time on computation relative to communication, leading to a decrease in the average time per rank. In addition, with small inputs, adding more processes doesn't fully utilize the available resources, meaning that some processes may remain idle or underused. The increased number of processes introduces

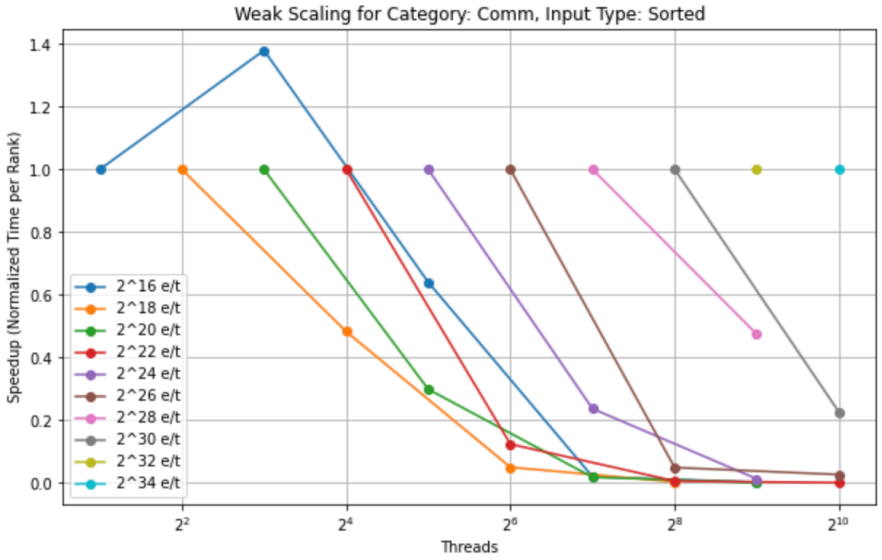
unnecessary communication and synchronization overhead for small workloads, resulting in increased time per rank. For larger inputs, however, the workload scales with the number of processes, making each process more efficiently utilized, which lowers the average time per rank.

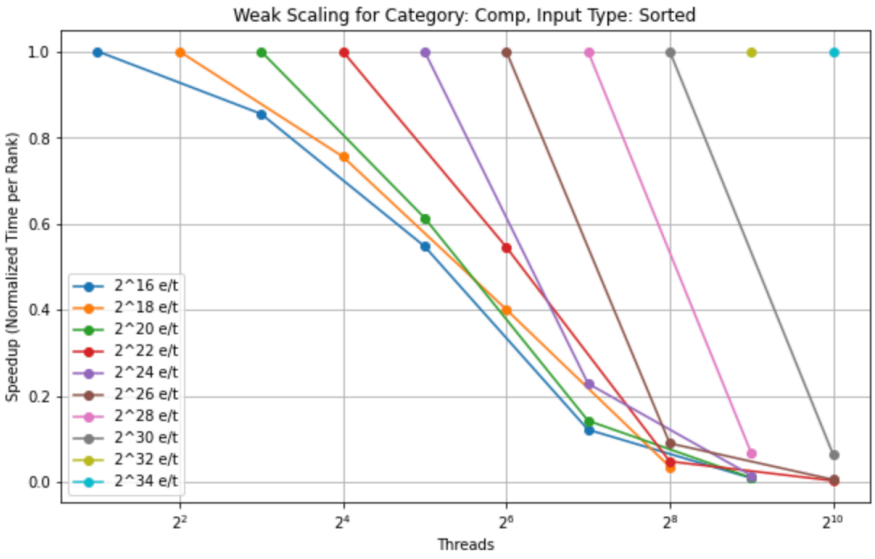
Overall Analysis

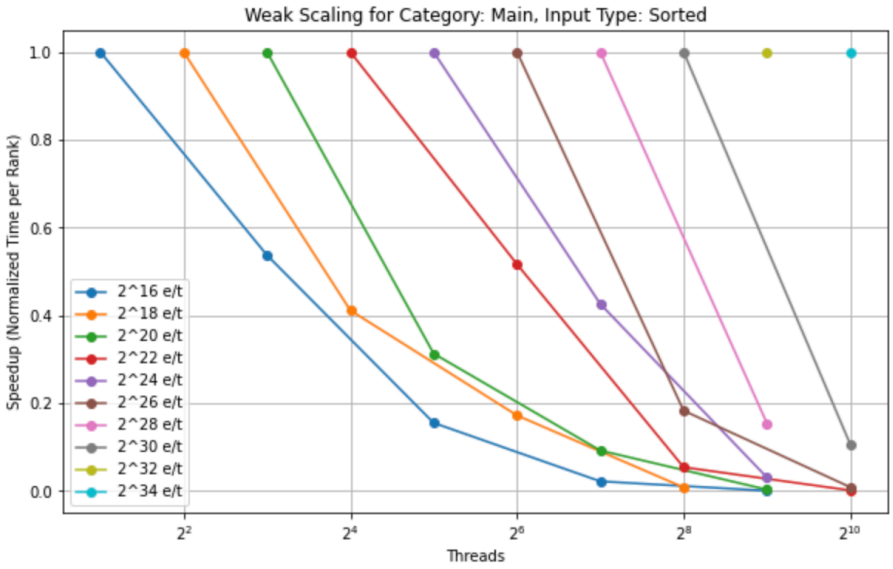
Based on the resulting plots, the performance of the bitonic sort algorithm shows a balance between parallel efficiency and the inherent challenges of communication and synchronization. The algorithm benefits from parallelization, particularly with larger problem sizes, where computation dominates and overheads like communication are amortized, leading to near-linear reductions in execution time as the number of processes increases. This is shown in the consistent linear decrease in comp_large times, indicating that the algorithm scales well for large datasets. However, the algorithm has diminishing returns as more processes are added, especially with smaller problem sizes. This is because communication overhead and synchronization begin to dominate, as shown by the decreasing speedup with additional processes. Amdahl's Law limits the potential speedup due to the non-parallelizable parts of the algorithm, such as merging and comparing, which require extensive communication between processes. Additionally, random inputs make communication overhead worse due to unpredictable data movement, while the other input types lead to more efficient execution due to more predictable data patterns. In weak scaling, the average time per rank initially increases for smaller problem sizes due to communication overhead, but decreases with larger problems as processes are more efficiently utilized. This indicates that while the bitonic sort algorithm can handle large-scale parallelism well, its performance is limited by communication costs and load imbalances at smaller scales. Overall, the parallelized bitonic sort algorithm performs well for large problem sizes with high computation-to-communication ratios but struggles with communication overhead and diminishing returns as the number of processes increases, especially for smaller inputs.

4b. Merge Sort

Weak Scaling

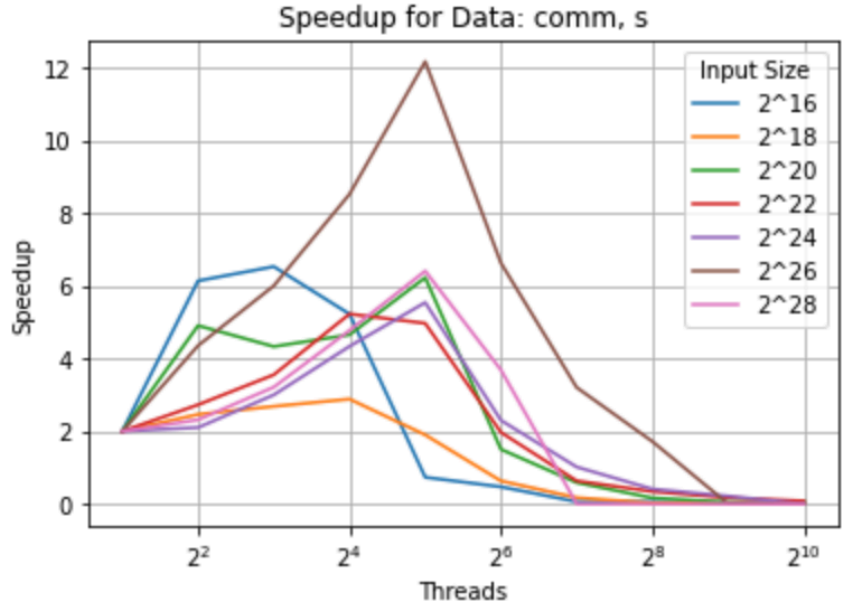


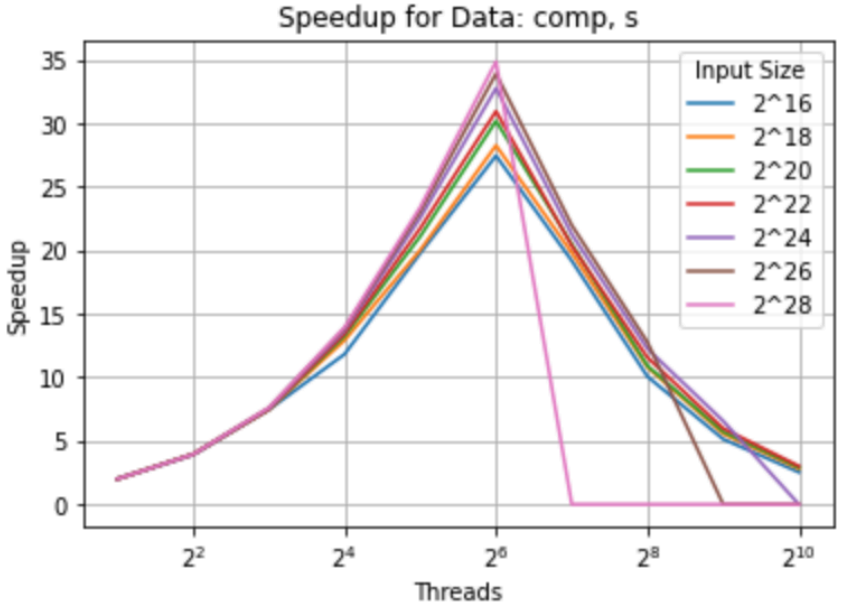


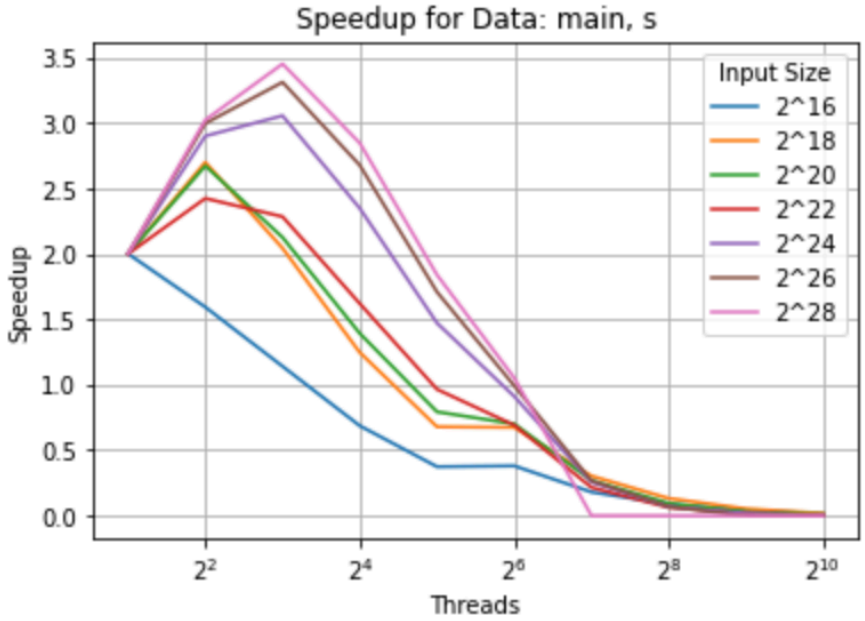


The three graphs represent weak scaling performance for communication (Comm), computation (Comp), and the main workload, all using sorted input data. As the number of threads increases, the speedup generally declines across all categories, highlighting the impact of overhead on scaling efficiency. Communication is the most affected, with a steep drop in performance, indicating that the overhead associated with communication becomes increasingly limiting as more threads are added. In contrast, computation maintains a more gradual decline, suggesting it scales more effectively and is less sensitive to the increase in threads. The main workload, which combines both communication and computation, shows a balanced trend with an intermediary decline in speedup, reflecting the cumulative impact of both types of overhead. Overall, these graphs reveal that communication overhead poses the greatest challenge to weak scaling, while computation retains relatively better efficiency.

Speed Up

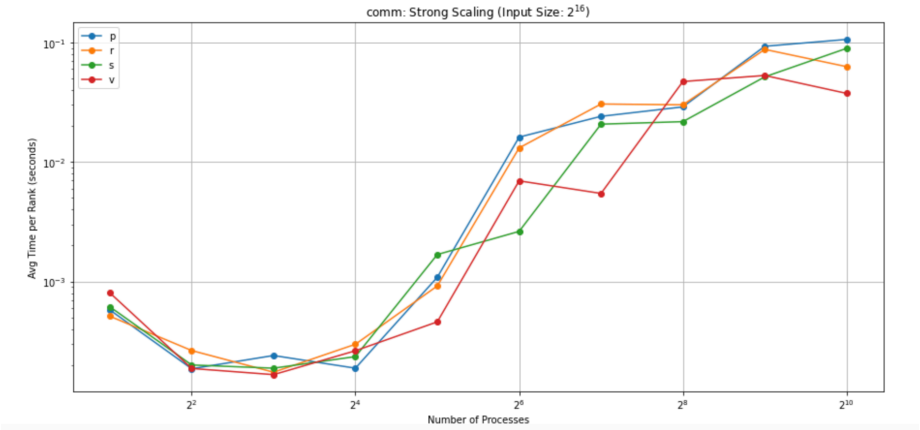

Speedup for Data: comm, s
Input Size
2^16
2^18
2^20
2^22
2^24
2^26
2^28
Speedup
Threads
0
2
4
6
8
10
12
2^2
2^4
2^6
2^8
2^{10}

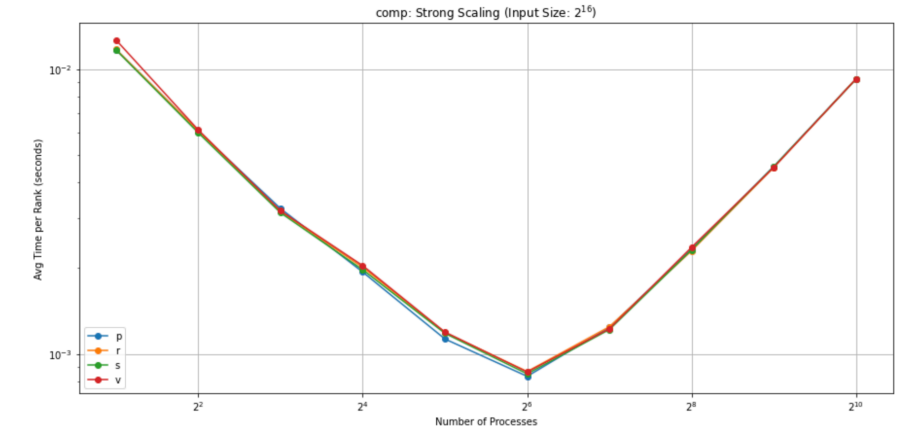

Speedup for Data: comp, s
Input Size
2^16
2^18
2^20
2^22
2^24
2^26
2^28
Speedup
Threads
0
5
10
15
20
25
30
35
2^2
2^4
2^6
2^8
2^{10}

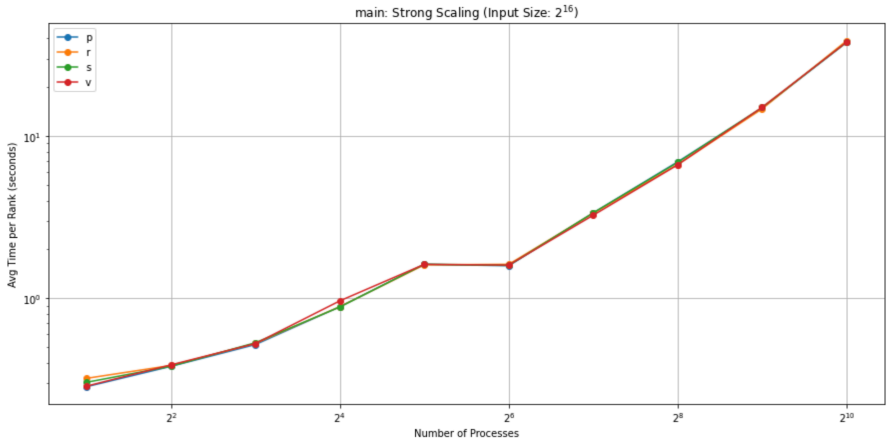


The three graphs display speedup trends for weak scaling across different categories—communication (Comm), computation (Comp), and the main workload—with sorted data input. In the communication graph, speedup reaches a peak with a moderate number of threads but declines sharply as thread counts increase, suggesting significant communication overhead in handling larger thread numbers. The computation graph, however, shows a much higher peak speedup, especially for larger input sizes, indicating that computation benefits substantially from increased parallelism up to a certain point before diminishing returns set in. The main workload graph combines aspects of both communication and computation, exhibiting a moderate peak followed by a steady decline as threads increase. This mixed behavior reflects the balance between computation gains and communication costs, with the latter limiting overall efficiency at higher thread counts. Overall, while computation shows the highest scalability, communication becomes a bottleneck as thread count grows, impacting the main workload's speedup performance.

Strong Scaling







The three graphs show strong scaling performance across different components—

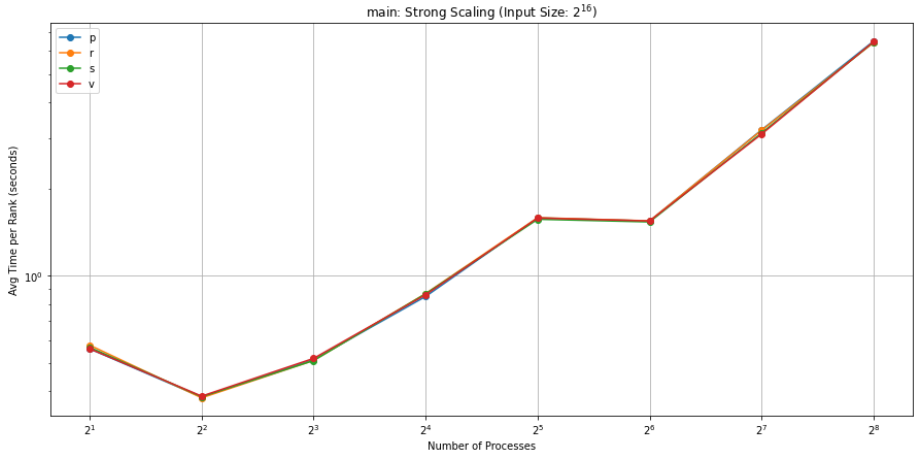
main, communication (comm), and computation (comp)—for an input size of (2^16). In the main workload graph, the average time per rank steadily increases as the number of processes rises, indicating limited scalability and suggesting that adding more processes doesn't significantly improve performance. In contrast, the communication graph initially shows a decline in average time per rank, but after reaching a minimum point, it rises sharply with more processes, indicating that communication overhead increases significantly, especially beyond (2^5) processes. This suggests that, in this setup, communication efficiency deteriorates with higher process counts. The computation graph presents the most favorable scaling, with a distinct U-shaped curve; time per rank decreases as processes increase until around (2^6), after which it begins to rise slightly. This indicates that computation benefits from parallelism up to a point, but further increases in processes lead to diminishing returns, possibly due to synchronization overhead or resource contention. Overall, these graphs highlight the limitations of strong scaling, especially for communication-heavy tasks, while computation retains better scalability.

Overal Analysis

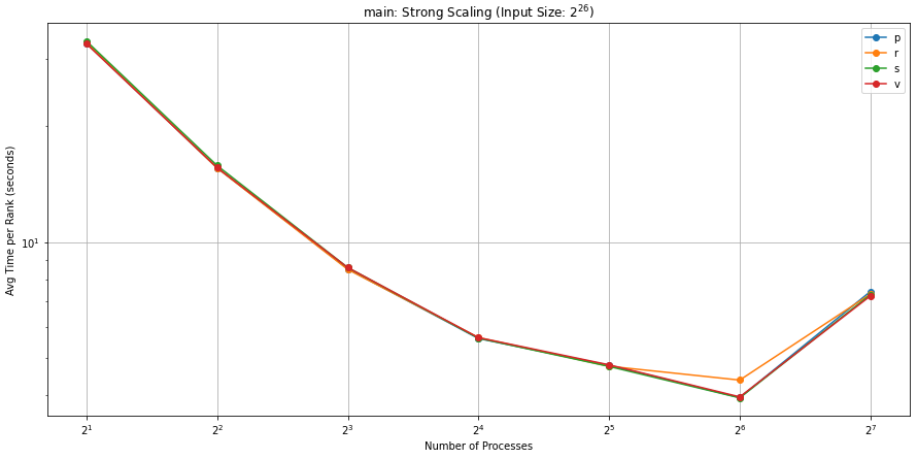
In my parallel implementation of Merge Sort, I observed that computation scaled efficiently up to a certain point, with the average time per rank decreasing as the number of processes increased, indicating effective load distribution. However, communication overhead became a limiting factor as the process count grew, significantly impacting performance and hindering scalability. This suggests that while the computational aspects of my implementation benefit from parallelism, further optimization of communication could enhance overall performance and scalability.

4c. Sample Sort

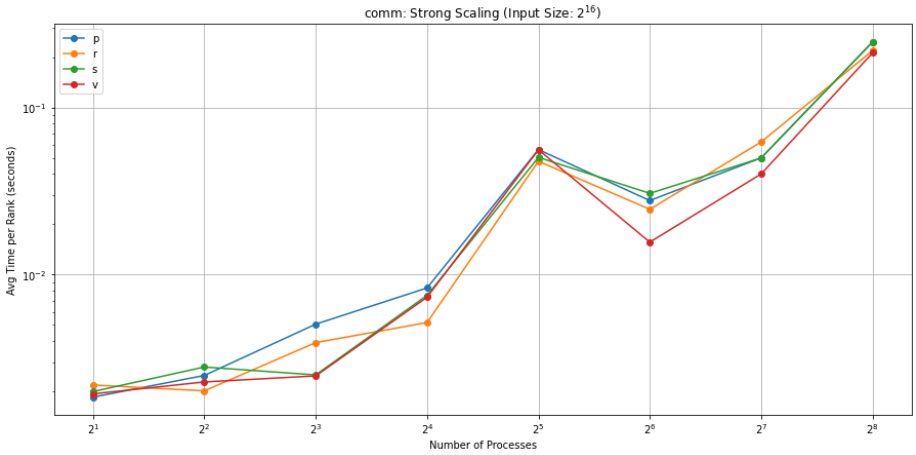
Strong Scaling



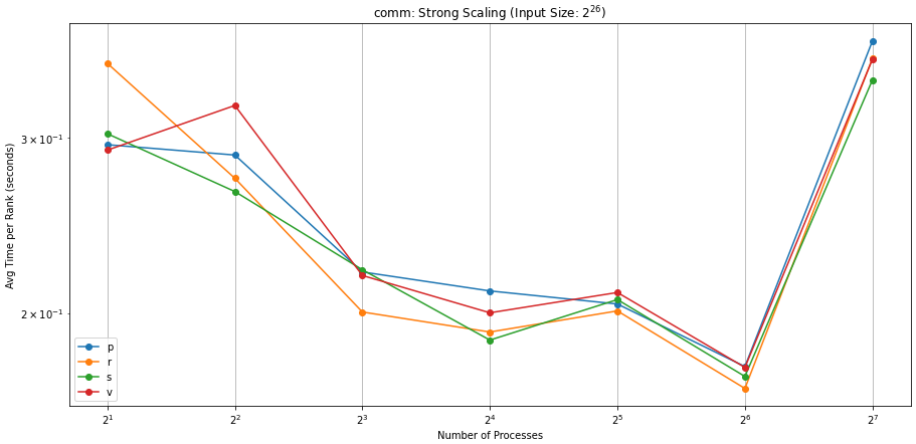
Main



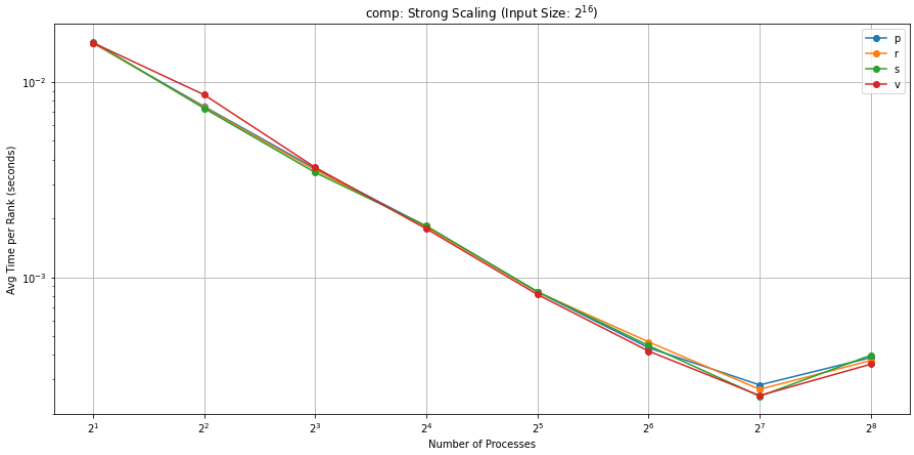
In the above two plots we observe that when the sample size is small there's an increase in time to process the whole program, but when the sample size is large there's actually an decrease in the time when we increase the no. of processors. So, we see an better strong scaling for larger sample size.



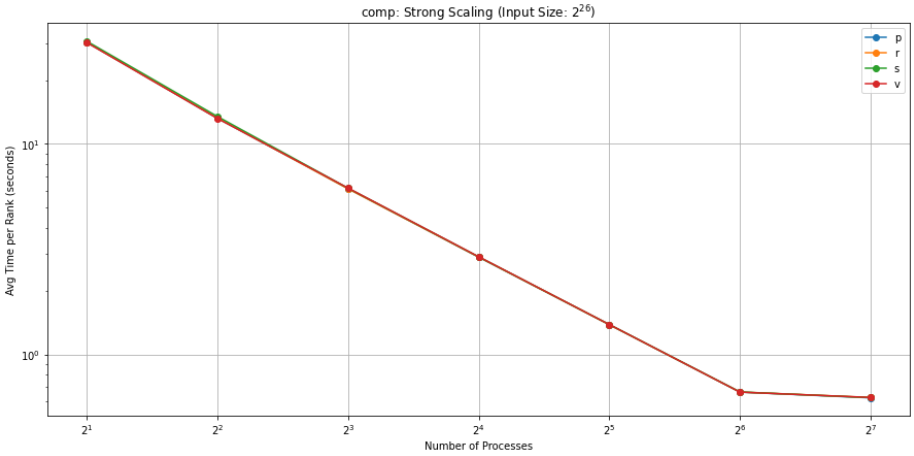
Comm



We observe similar trend in our communication time, where when sample size is small we observe an increase in communication time but when sample size is large we observe an decrease in communication time when we increase the no. of processors. which says that parallelism helped our sample sort algorithm run efficiently and faster.



Comp



Regardless of our sample size, we observe an decrease in computation time when we increase the no. of processors which implies our sample sort scales.

Strong Speed-Up

Main We observe the strongest speed-up from larger sample size, particularly when our sample size was 2^26 the algo experienced drastic speed-up until 2^6 processor after which the speed-up decreased

Comm We observe the strongest speed-up when our sample size is the largest while for smaller sample size we observe that the speed-up decreases as we increase the no. of processors.

Comp Regardless of the sample size we observe strong speed-up as we increase the no. of processors.

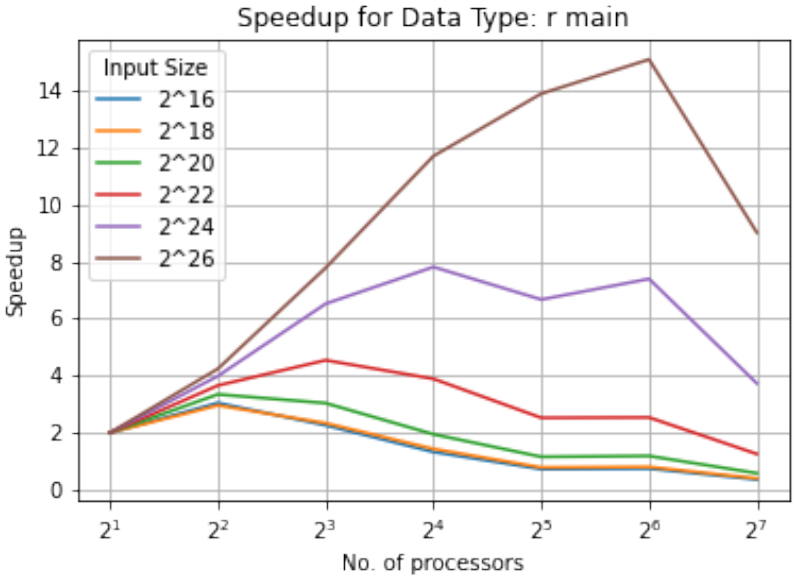


Figure 1: speedup_plot_r_main

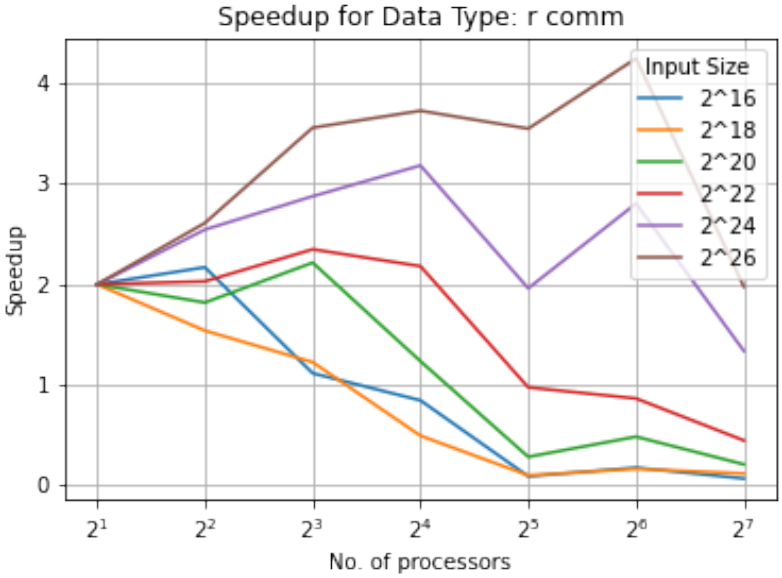


Figure 2: speedup_plot_r_comm

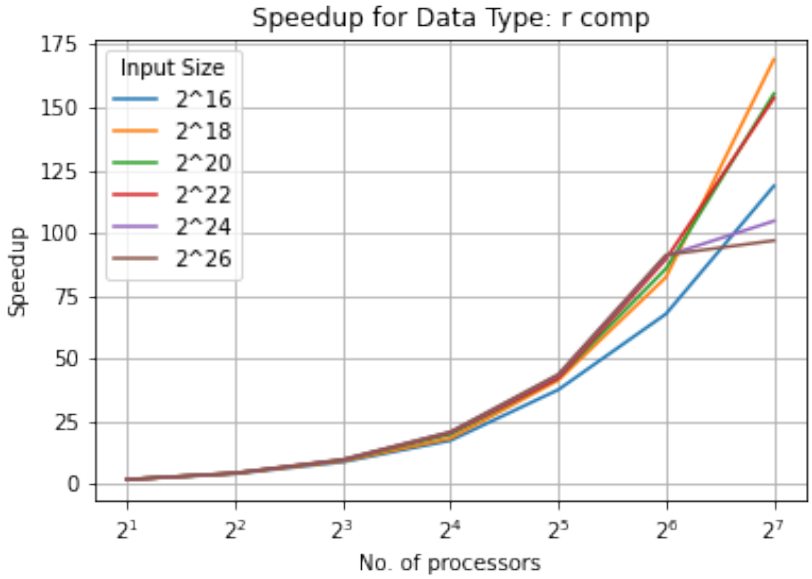


Figure 3: speedup_plot_r_comp

Weak Scaling

main

Comm

Comp In all above plots we observe an decreasing trend when we increase both the no. of processor and sample.

Overall Analysis

Our sample sort algo scales efficiently and is helped by the parallelism. which can be observed particularly well in our strong scaling plots where we observed decreasing computation time when we increase the no. of processor for all sample size. But the sample size did matter for both overall and communication time, where we observed that when sample size is small we don't get the benefits of parallelism but when our sample size is larger we observed improvement in both the overall and communciation time.

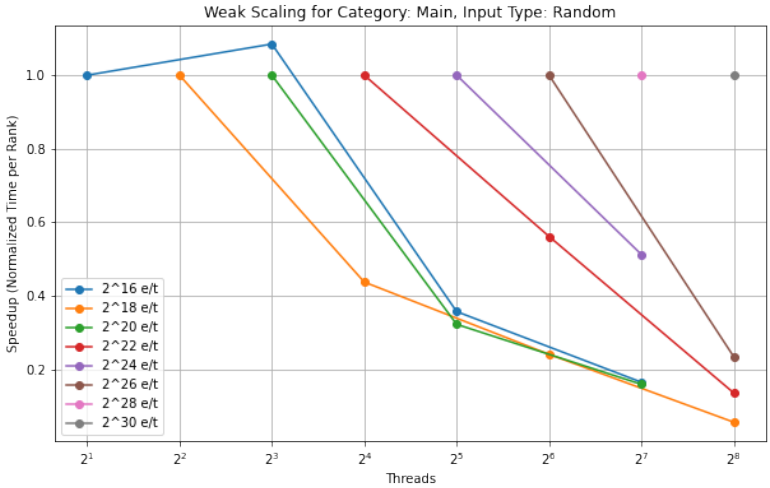


Figure 4: weak_scaling_main_random

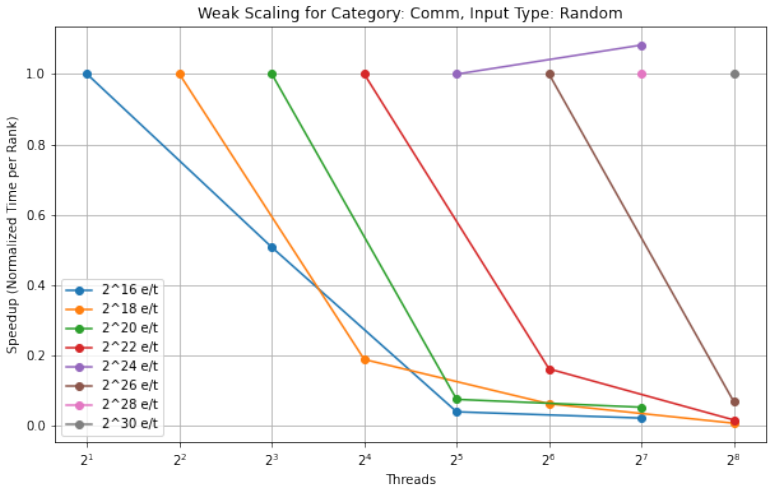


Figure 5: weak_scaling_comm_random

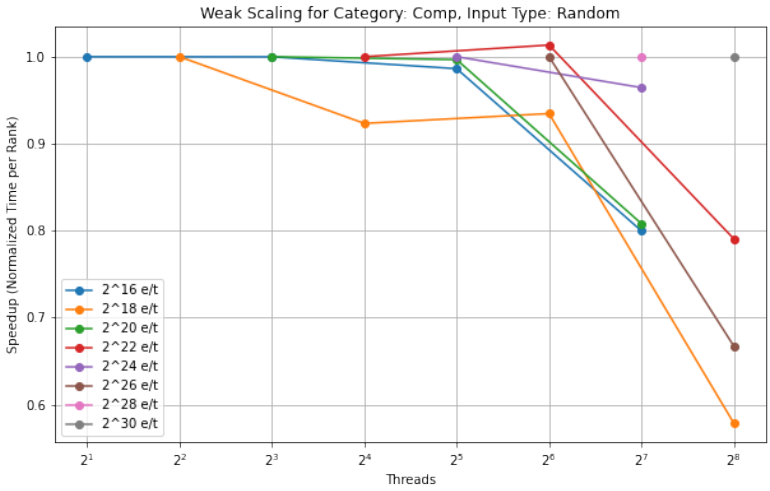


Figure 6: weak_scaling_comp_random

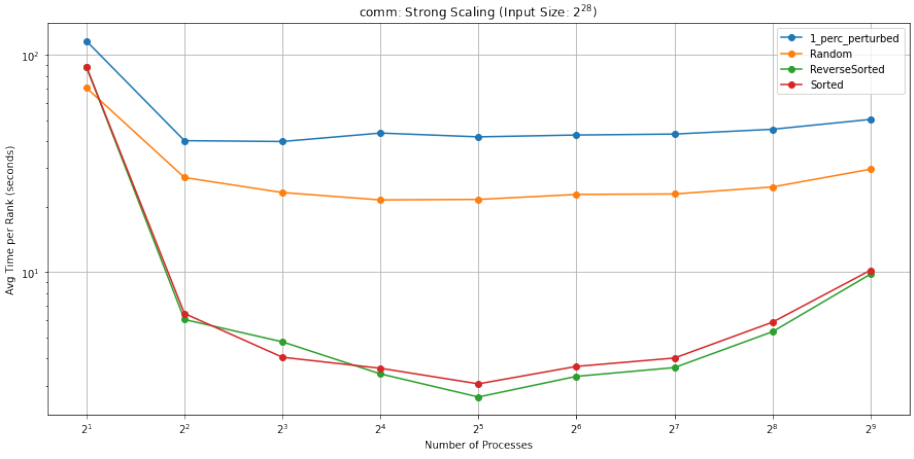


Figure 7: comm28

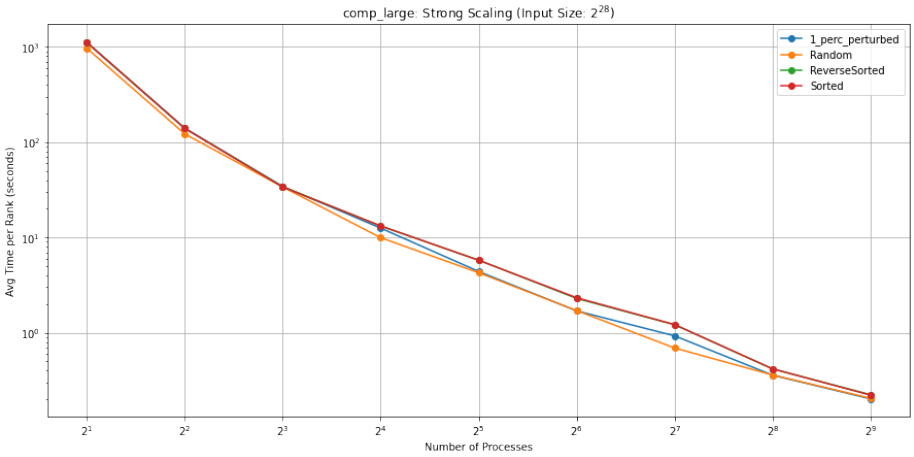


Figure 8: comp_large28

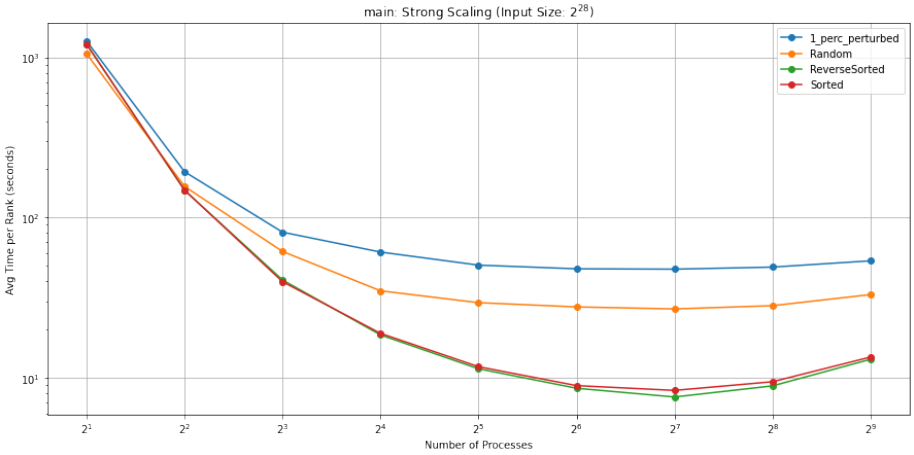


Figure 9: main28

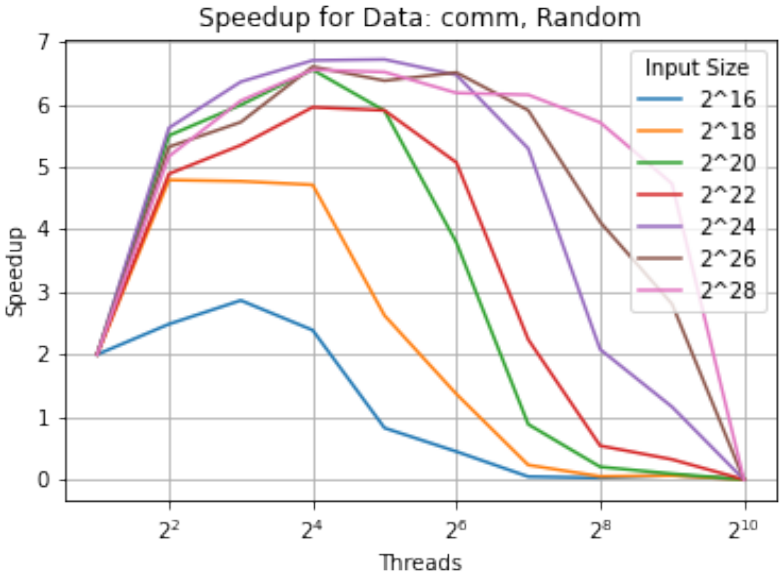


Figure 10: speedup_comm_data_type_Random

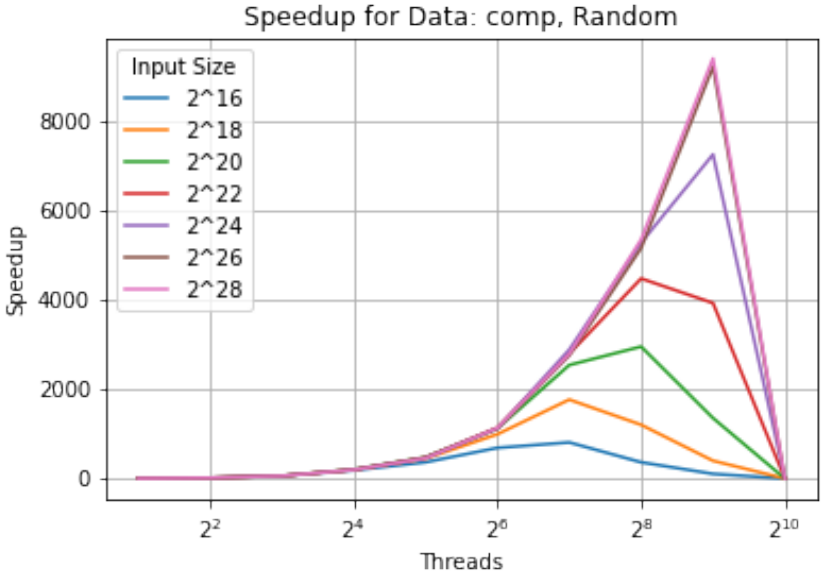


Figure 11: speedup_comp_data_type_Random

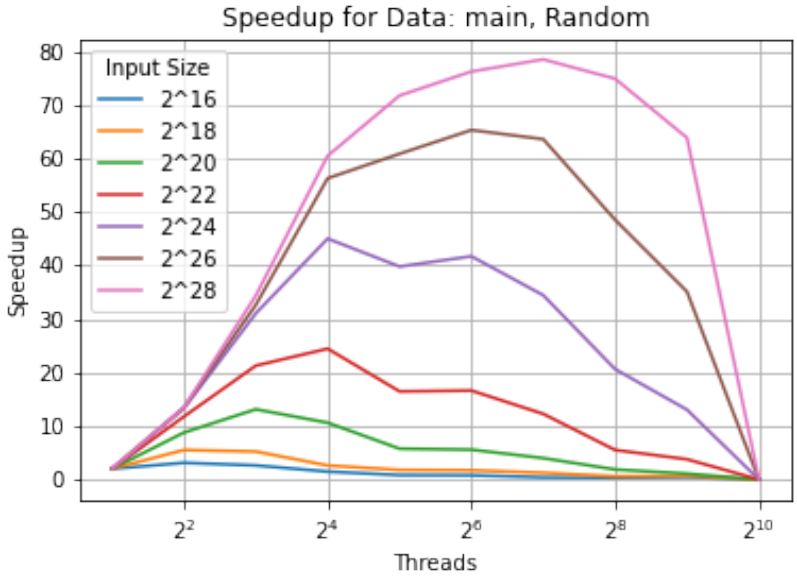


Figure 12: speedup_main_data_type_Random

4d. Radix Sort

Strong Scaling

Strong Scaling Speed-Up

Weak Scaling

Analysis

Radix Sort Overall Performance: The plots showed behaviors mostly in line with behaviors observed in previous labs when working with different input sizes and processor counts. This time however, there were different input types, and they mostly had impacts on communication. As input sizes increased, communication times often took longer. As processor counts increased, computation times often took less time. There were some kinks that could be worked out to improve the Caliper data, but this is mostly representative of the true data.

Notes: Due to the networking issues that many other students were facing, I was unable to successfully produce runs for 1024 processors. I was able to produce a few, which can be seen in a few of the plots above, but most other runs would hang and fail. To save credits from being unnecessarily used, I opted to just not run these and to possibly try over the weekend when there are not as many users.

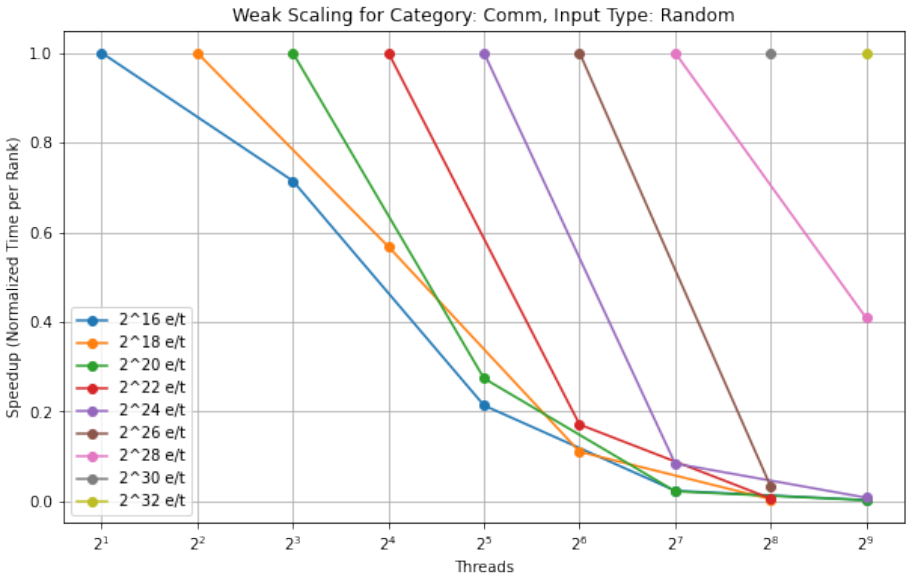


Figure 13: comm_random

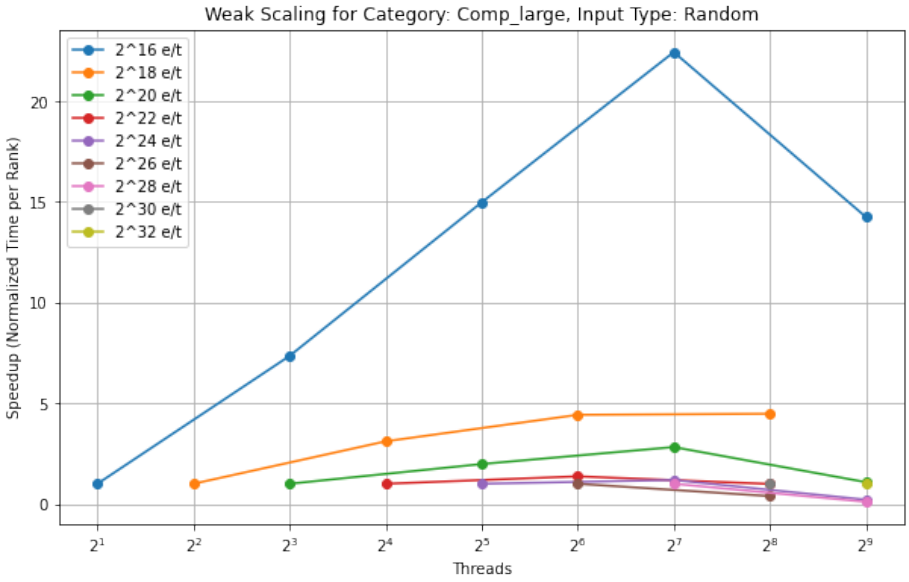


Figure 14: comp_random

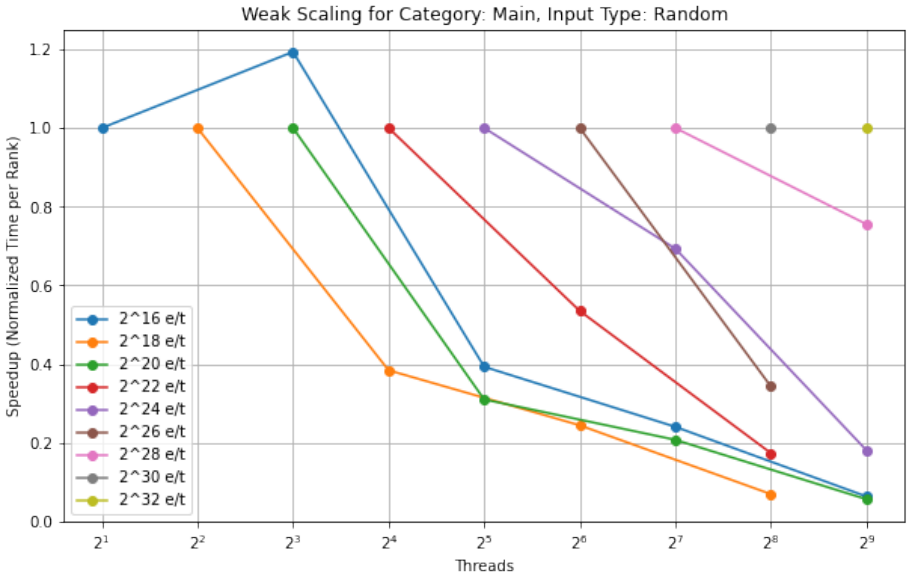
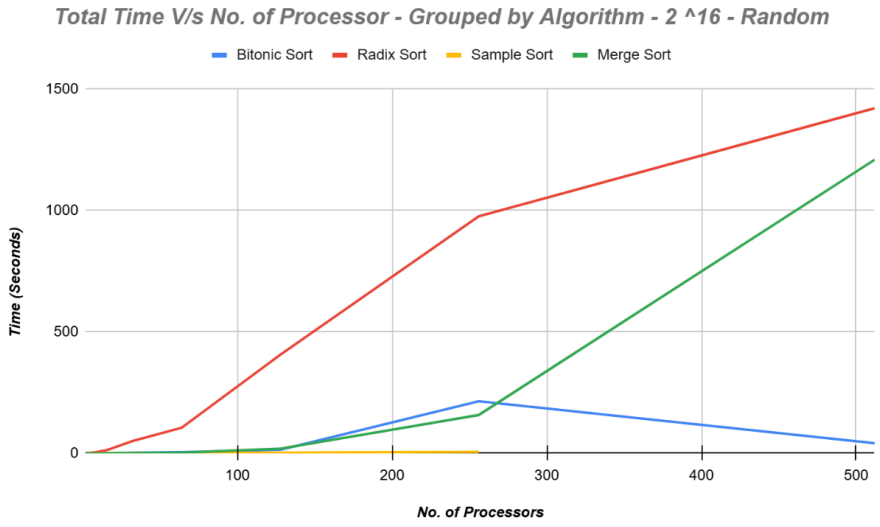
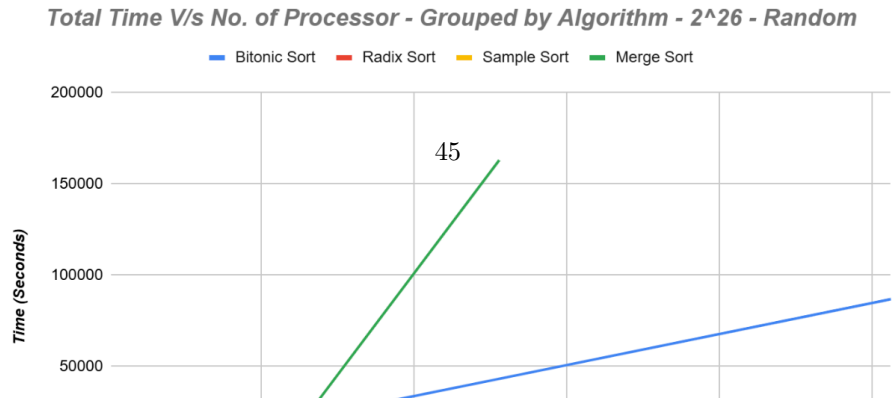


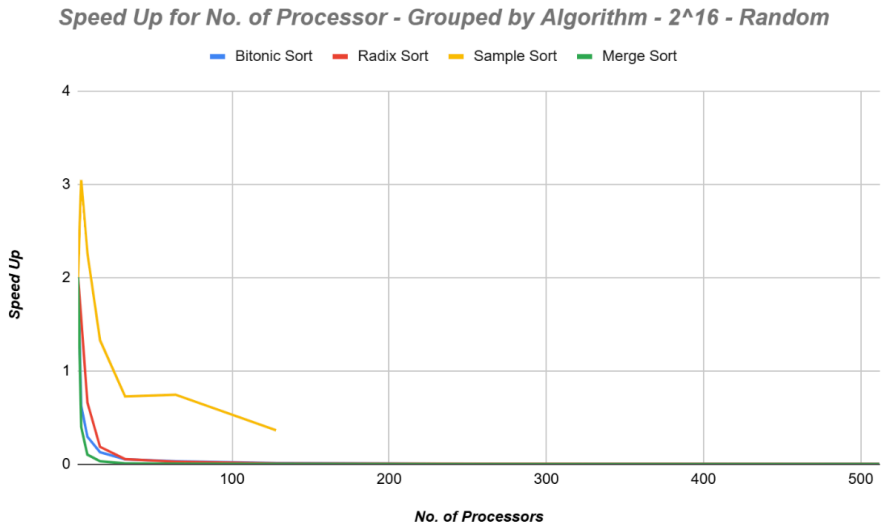
Figure 15: main_random

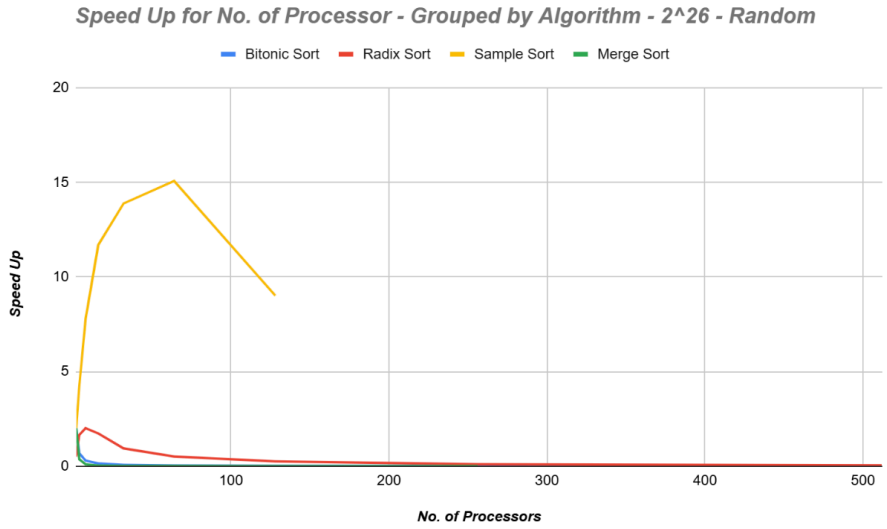
Comparing All Sorting Algoriths

Input Type: Main & Random









Overal Analysis

The graphs display the performance of Bitonic Sort, Radix Sort, Sample Sort, and Merge Sort across different processor counts for random input data sizes (2^16) and (2^26). In terms of total execution time, Bitonic Sort and Sample Sort show strong scalability with increasing processors, maintaining low execution times, while Merge Sort and Radix Sort see significant time increases, especially at larger input sizes. For speedup, Sample Sort exhibits the highest initial gains, particularly at (2^26), but its speedup declines as the number of

processors increases, suggesting inefficiency at higher scales. Bitonic Sort also achieves moderate speedup, though with a less dramatic initial peak. Both Radix Sort and Merge Sort have minimal speedup gains, indicating limited scalability in a parallel setting for this random input type. Overall, Bitonic and Sample Sort perform more efficiently in parallel on random data, while Radix and Merge Sort struggle to benefit from added processors, particularly with larger inputs, showing limitations in their parallel scalability.