CSCE 435 Group project

0. Group number and the communication medium(slack etc.):

Group number : 6 Communication medium : Discord

1. Group members, what algorithm they are implementing

1. Bitonic Sort: Jack Williams
2. Merge Sort: Adeeb Ismail
3. Radix Sort: Soham Nagawanshi
4. Sample Sort: David Armijo

2. Brief project description, what architecture you are comparing your sorting algorithms on.

2a. Pseudocode for Bitonic Sort.

BITONIC_MPI_SORT(comm, local[]): MPI_Init() P ← MPI_Comm_size(comm) r ← MPI_Comm_rank(comm)

```
# 0) Local pre-sort (ascending)
sort(local)   # e.g., quicksort; in-place; ascending

# 1) Process-level bitonic network
size ← 2
while size ≤ P:

    # j walks partner distances inside this size-group: size/2, size/4,
..., 1
    j ← size / 2
    while j ≥ 1:

        partner ← r XOR j

        # Keep-lower vs keep-upper decision:
        # If our 'size' bit in r is 0 ⇒ we are in the group's lower half ⇒
keep lower n.
        # If it's 1 ⇒ we are in the group's upper half ⇒ keep upper n.
        keep_lower ← ((r AND size) == 0)

        # Exchange equal-sized sorted blocks with partner
        temp[] ← buffer of length n (same type as local[])
        MPI_Sendrecv(sendbuf=local, sendcount=n, sendtype=T, dest=partner,
tag=0,
                     recvbuf=temp,  recvcount=n, recvtype=T,
source=partner, recvtag=0,
                     comm, status=IGNORE)

        # Compare–split: merge two sorted blocks and keep our half
        if keep_lower:
            local ← MERGE_KEEP_LOW_N(local, temp)     # keep smallest n
```

```
        else:
            local ← MERGE_KEEP_HIGH_N(local, temp)    # keep largest n

        j ← j / 2
    end while

    size ← size * 2
end while

# (Optional) Gather final globally sorted data to root for
verification/display
# MPI_Gatherv(local → root)

MPI_Finalize()
return
```

---------- Helpers ----------

MERGE_KEEP_LOW_N(A[], B[]): # A and B: each length n, both sorted ascending # Return the smallest n elements of their union without building 2n i ← 0; j ← 0; k ← 0 C[] ← new array length n while k < n: if j == n or (i < n and A[i] ≤ B[j]): C[k] ← A[i]; i ← i + 1 else: C[k] ← B[j]; j ← j + 1 k ← k + 1 return C

MERGE_KEEP_HIGH_N(A[], B[]): # A and B: each length n, both sorted ascending # Return the largest n elements; scan from the end i ← n - 1; j ← n - 1; k ← n - 1 C[] ← new array length n while k ≥ 0: if j < 0 or (i ≥ 0 and A[i] ≥ B[j]): C[k] ← A[i]; i ← i - 1 else: C[k] ← B[j]; j ← j - 1 k ← k - 1 return C

Include MPI calls you will use to coordinate between processes.

2b. Pseudocode for Merge Sort.

Include MPI calls you will use to coordinate between processes.

The MPI calls we will need are

- MPI_Send
- MPI_Recv
- MPI_Init
- MPI_Finalize

Pseudocode (Master/Worker, Point-to-Point)

```
IF master:
  for worker in workers:
    MPI_Send() send equal amounts of data to each process
  MergeSort() array of master
  for worker in workers:
    MPI_Recv() from all the workers
    Merge() the recieved arrays together
```

```
ELSE:
  MPI_Recv() recieve data from the master
  MergeSort() array of each worker
  MPI_Send() send the sorted array to the master


Merge(a,b) merges the two arrays together in sorted order
MergeSort() performs merge sort
```

Pseudocode (SPMD, Broadcast)

```
generate data globally, split up to each process

MergeSort() each array
step = 1;
WHILE (step < size)
  IF rank % (step * 2):
    recv_proc = rank + step recv from its neighbor
    MPI_Recv()
    merge() the two arrays
    validate() each merge array
  ELSE
    send_proc = rank - step send to its neighbor
    MPI_Send()
  step *= 2
```

2c. Pseudocode for Radix Sort.

note: LSD = least significant digit

```
data: data to sort
k: maximum number of digits in any element of data
P: processor array

function radix-sort(data, k, P)
  # initialization
  blocks <- divide into |P| continguous blocks (each has size |data|/|P|)

  If Master:
    MPI_Send(blocks_i, p_i) # send each block to corresponding processor

  If Worker:
    MPI_Recv(my_block, Master)
    Repeat for i in {1 ... k}
      buckets <- {0: [], 1: [], ..., 9: []} # itemizes data by i'th LSD
      hist <- [0, 0, .., 0] # keeps track of the size of each digit bucket
      for each element in my_block
        dig <- ith LSD of element
        add element to buckets[dig]
```

```
        hist[dig] += 1
      MPI_Allgather(hist) # broadcast histogram to all workers, result is
|P| x 10 matrix
      # send buckets to processors in sequential fashion (i.e. bucket 0
will go to smaller rank processors but
                                                        bucket 9 will go
to bigger rank processor)
      recv_procs <- calculate which processes you will receive from based
on hist
      send_procs <- calculate which processes you will send to based on
hist
      for each recv_proc:
          MPI_Recv(blocks, recv_procs)
      for each send_proc:
          MPI_Send(blocks, send_procs)
    MPI_Send(blocks, Master)
  If Master:
    sorted_data <- empty array
    for each p in P: # iterate sequentially for correct order
        MPI_Recv(blocks, p)
        place blocks in sorted_data

    output sorted data
```

2d. Pseudocode for Sample Sort.

MPI Calls We will Need

- MPI_Send (For Master and Slave). Whenever point-to-point communication is necessary
- MPI_Recv (For Master and Slave). Whenever point-to-point communication is necessary
- MPI_Init
- **MPI_Allgather**
- **MPI_Alltoall**
- **MPI_Alltoallv**
- MPI_Finalize

Pseudocode

Include MPI calls you will use to coordinate between processes. For array size N and processor count P,

1. Pre-Sorting (Sequential portion):

- Split up and distribute the array evenly to each processor.
- P1 gets matrix [0:1*(N/P)-1], P2 gets matrix [1*(N/P):2*(N/P)-1], ..., Pk gets matrix [(k-1)*(N/P):k*(N/P)-1] and so on.
- Use MPI_Send to send the separated data to all the processes

2. Local Sort:

- In each of the processors in the rank, sort the arrays locally.

3. Local Sampling:

- In each of the local arrays in the rank, select `s = P` evenly spaced elements as its sample.
- Example: `s=P=4`, positions samples will be (0,2,4,6) in each of those local arrays if the local array size is 8 elements

4. Gathering Samples:

- Using `MPI_Allgather` function, gather all samples from every processor.

5. Sort the Gathered Samples:

- Sort the gathered samples array (that came from all the ranks)

6. Partition by Splitters:

- In the sorted gathered array, split that array up into P parts. In other words, make P-1 splitters and make them the values of your bucket list.
- Ex: For 16 samples (from those 4 ranks or Processors), pick indices 1*(16/4), 2*(16/4), and 3*(16/4). These will be your S1 = **, S2 = **, and S3 = __.

7. Partition by splitters.

- Your buckets are the values at the indices of the sorted, combined sample list. All the ranks then receive that list of splitter values to bin their local lists via `MPI_Alltoall`.

8. Sort local lists using global splitter values.

- Make sure to place them in their respective bins if they are less than or equal to that splitter value.

9. All-to-All Exchange

- Each rank sends Bj to rank j. After `MPI_Alltoallv`, each rank holds one bucket's *global* range.

10. Final local sort/merge:

- If you've done stable partitioning on already sorted local arrays, each rank can *merge* its received, already-sorted chunks (k-way merge). Otherwise, just sort once more locally. The concatenation across (0...3) is the globally sorted order

3. Evaluation plan - what and how will you measure and compare

We will vary multiple parameters and will measure performance using caliper instrumentation. For each experiment, we will take the measurements mentioned in 6c. Our first experiment will vary data types and fix the other parameters. For this experiment, we plan to compare sorting an integer array with a double array. Our second experiment will vary input data sizes while fixing other parameters. We will use 2^16, 2^18, 2^20, 2^22, 2^24, 2^26, and 2^28 elements for evaluation. Our third experiment will vary the sorting level. We will test on a sorted array, a sorted array with 1% perturbation, a randomly generated array, and a reversely sorted array. To gauge strong scaling performance, we will fix the input data size (likely to 2^22 elements) and use 2, 4, 8, 16, 32, 64, 128, 256, 512, and 1024 processors for evaluation. To gauge weak scaling, we will vary the input data size as we increase the processor count. We will likely use the following pairs:

- (2^16 elements, 16 processors)
- (2^18 elements, 32 processors)
- (2^20 elements, 64 processors)
- (2^22 elements, 128 processors)
- (2^24 elements, 256 processors)
- (2^26 elements, 512 processors)
- (2^28 elements, 1024 processors)

We will carry out these experiments for each of the sorting algorithms and compare the performance across different input sizes / processor counts using graphics. We will also include graphics that compare performance between algorithms.

4. Caliper instrumentation

Bitonic Sort

```
0.661 main
├─ 0.001 comm
│  └─ 0.001 comm_large
├─ 0.096 comp
│  └─ 0.096 comp_large
│     ├─ 0.006 merge_keep_high_n
│     └─ 0.005 merge_keep_low_n
├─ 0.004 correctness_check
└─ 0.063 data_init_runtime
```

Radix Sort

```
0.503 main
├─ 0.000 data_init_runtime
├─ 0.000 comp
│  ├─ 0.000 comp_large
│  └─ 0.000 comp_small
├─ 0.030 comm
│  ├─ 0.029 comm_small
│  └─ 0.001 comm_large
└─ 0.000 correctness_check
```

Merge Sort

```
0.867 main
├─ 0.000 data_init_runtime
├─ 0.000 comp
│  ├─ 0.000 comp_small_merge_sort
│  └─ 0.000 comp_large_merge_arrays
├─ 0.032 comm
```

```
│  ├─ 0.063 comm_large_recv
│  └─ 0.000 comm_large_send
└─ 0.000 correctness_check
```

Sample Sort

```
0.031 main
├─ 0.029 comm
│  ├─ 0.000 comm_large
│  └─ 0.029 comm_small
├─ 0.000 comp
│  ├─ 0.000 comp_large
│  └─ 0.000 comp_small
├─ 0.000 correctness_check
└─ 0.000 data_init_runtime
```

5. Collect Metadata

Have the following code in your programs to collect metadata:

```
adiak::init(NULL);
adiak::launchdate(); // launch date of the job
adiak::libraries(); // Libraries used
adiak::cmdline(); // Command line used to launch the job
adiak::clustername(); // Name of the cluster
adiak::value("algorithm", algorithm); // The name of the algorithm you are
using (e.g., "merge", "bitonic")
adiak::value("programming_model", programming_model); // e.g. "mpi"
adiak::value("data_type", data_type); // The datatype of input elements
(e.g., double, int, float)
adiak::value("size_of_data_type", size_of_data_type); // sizeof(datatype)
of input elements in bytes (e.g., 1, 2, 4)
adiak::value("input_size", input_size); // The number of elements in input
dataset (1000)
adiak::value("input_type", input_type); // For sorting, this would be
choices: ("Sorted", "ReverseSorted", "Random", "1_perc_perturbed")
adiak::value("num_procs", num_procs); // The number of processors (MPI
ranks)
adiak::value("scalability", scalability); // The scalability of your
algorithm. choices: ("strong", "weak")
adiak::value("group_num", group_number); // The number of your group
(integer, e.g., 1, 10)
adiak::value("implementation_source", implementation_source); // Where you
got the source code of your algorithm. choices: ("online", "ai",
"handwritten").
```

They will show up in the `Thicket.metadata` if the caliper file is read into Thicket.

See the `Builds/` directory to find the correct Caliper configurations to get the performance metrics. They will show up in the `Thicket.dataframe` when the Caliper file is read into Thicket.

6. Performance evaluation

Include detailed analysis of computation performance, communication performance. Include figures and explanation of your analysis.

6a. Vary the following parameters

For input_size's:

- 2^16, 2^18, 2^20, 2^22, 2^24,W 2^26, 2^28

For input_type's:

- Sorted, Random, Reverse sorted, 1%perturbed

MPI: num_procs:

- 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024

This should result in 4x7x10=280 Caliper files for your MPI experiments.

6b. Hints for performance analysis

To automate running a set of experiments, parameterize your program.

- input_type: "Sorted" could generate a sorted input to pass into your algorithms
- algorithm: You can have a switch statement that calls the different algorithms and sets the Adiak variables accordingly
- num_procs: How many MPI ranks you are using

When your program works with these parameters, you can write a shell script that will run a for loop over the parameters above (e.g., on 64 processors, perform runs that invoke algorithm2 for Sorted, ReverseSorted, and Random data).

6c. You should measure the following performance metrics

- `Time`
  - Min time/rank
  - Max time/rank
  - Avg time/rank
  - Total time
  - Variance time/rank

7. Presentation

Plots for the presentation should be as follows:

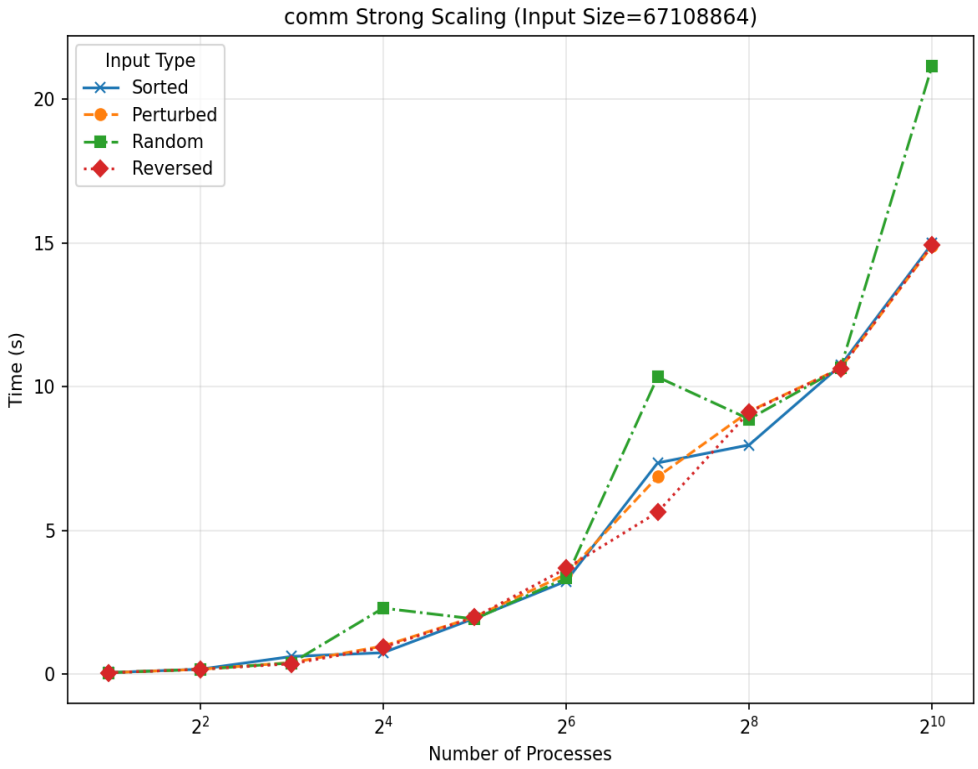
- For each implementation:

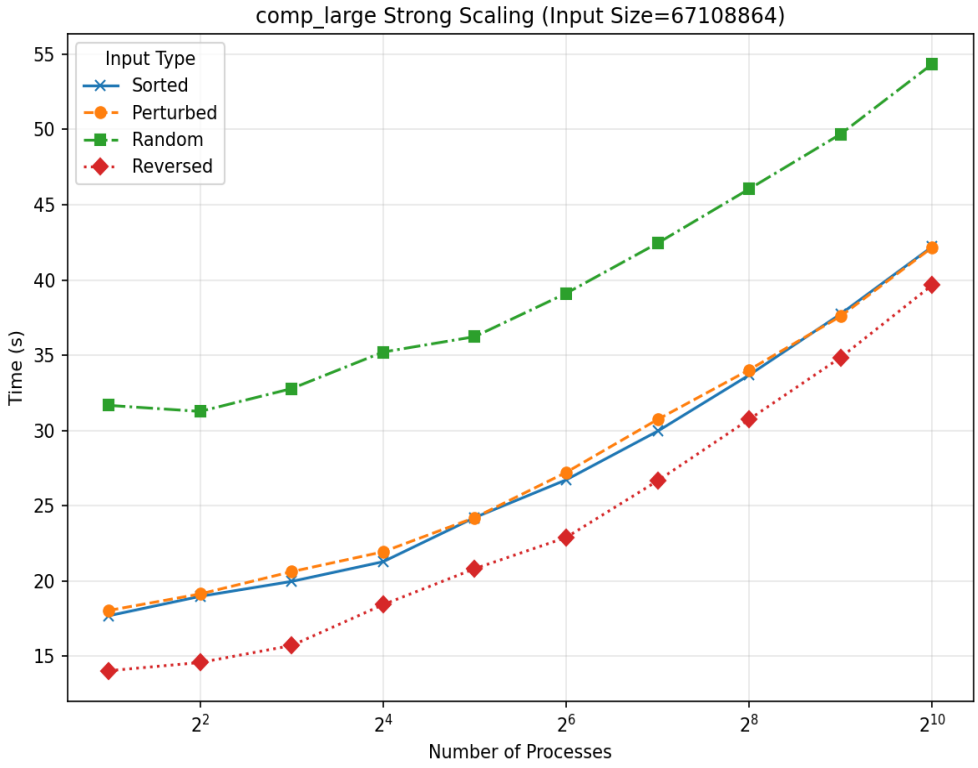
- For each of comp_large, comm, and main:
  - Strong scaling plots for each input_size with lines for input_type (7 plots - 4 lines each)
  - Strong scaling speedup plot for each input_type (4 plots)
  - Weak scaling plots for each input_type (4 plots)

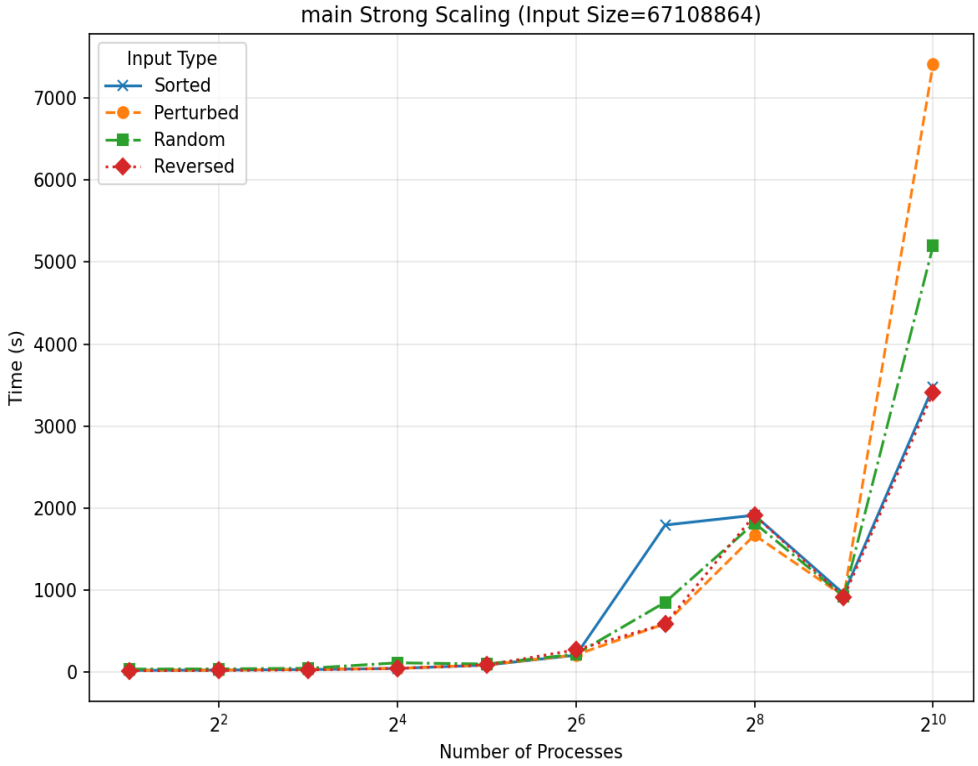
Analyze these plots and choose a subset to present and explain in your presentation.

Bitonic Sort

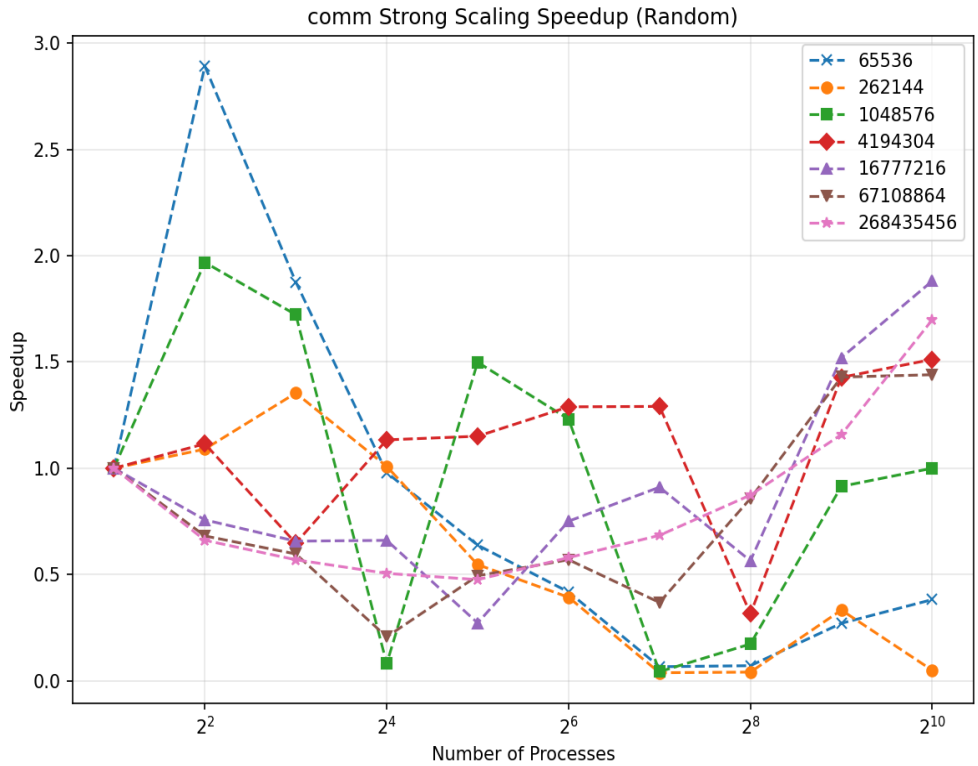
Strong Scaling Plots

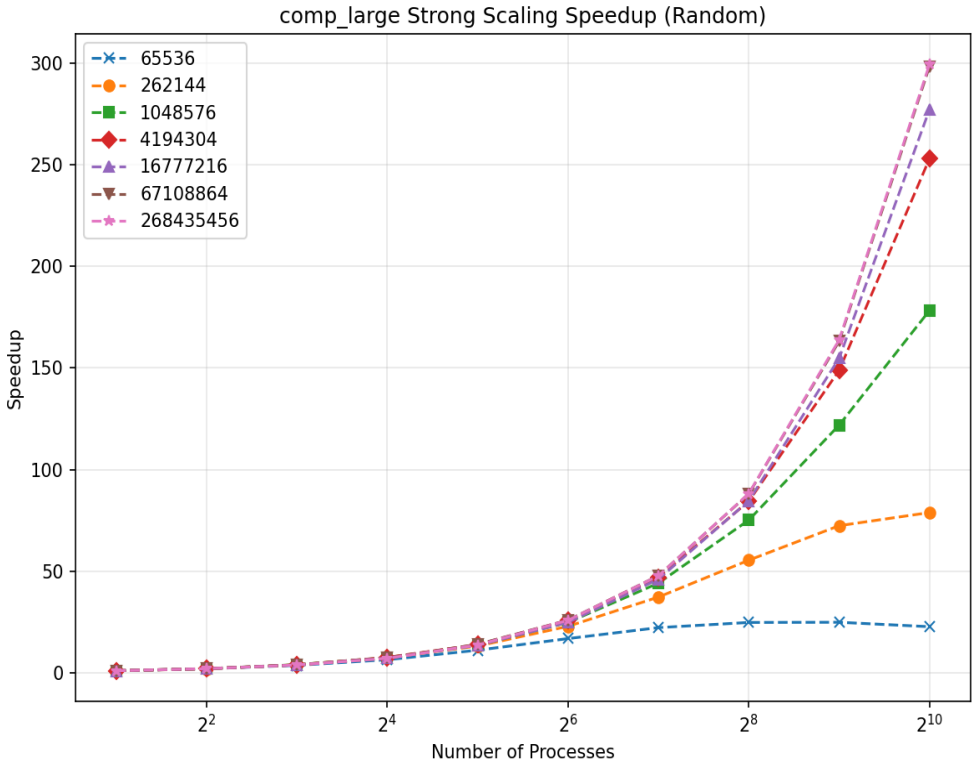


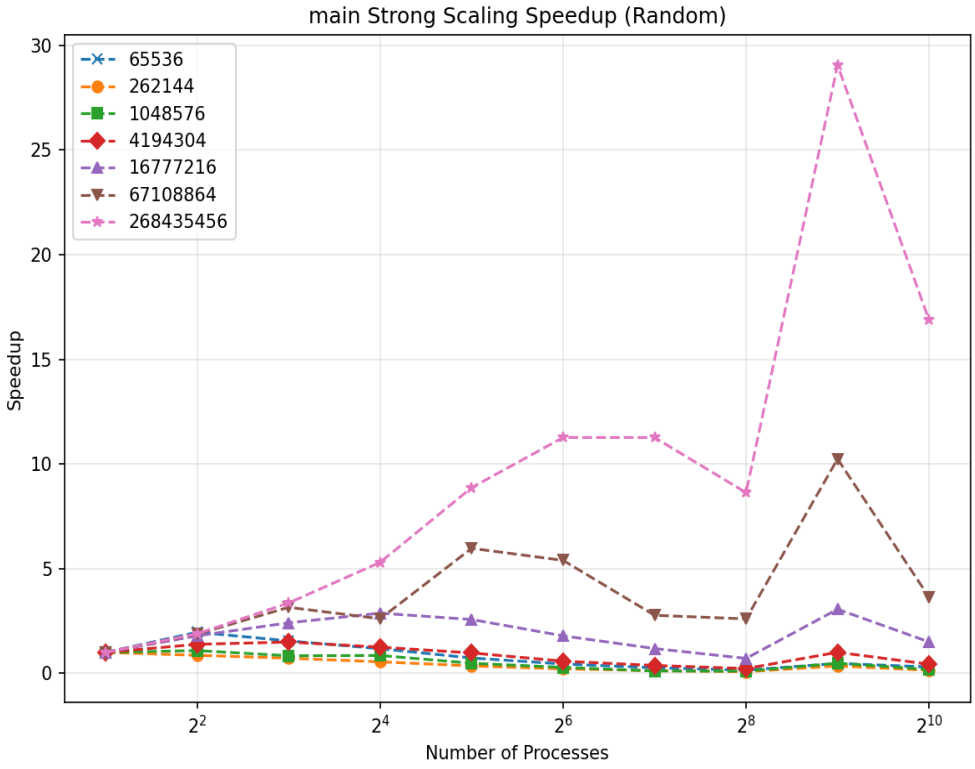




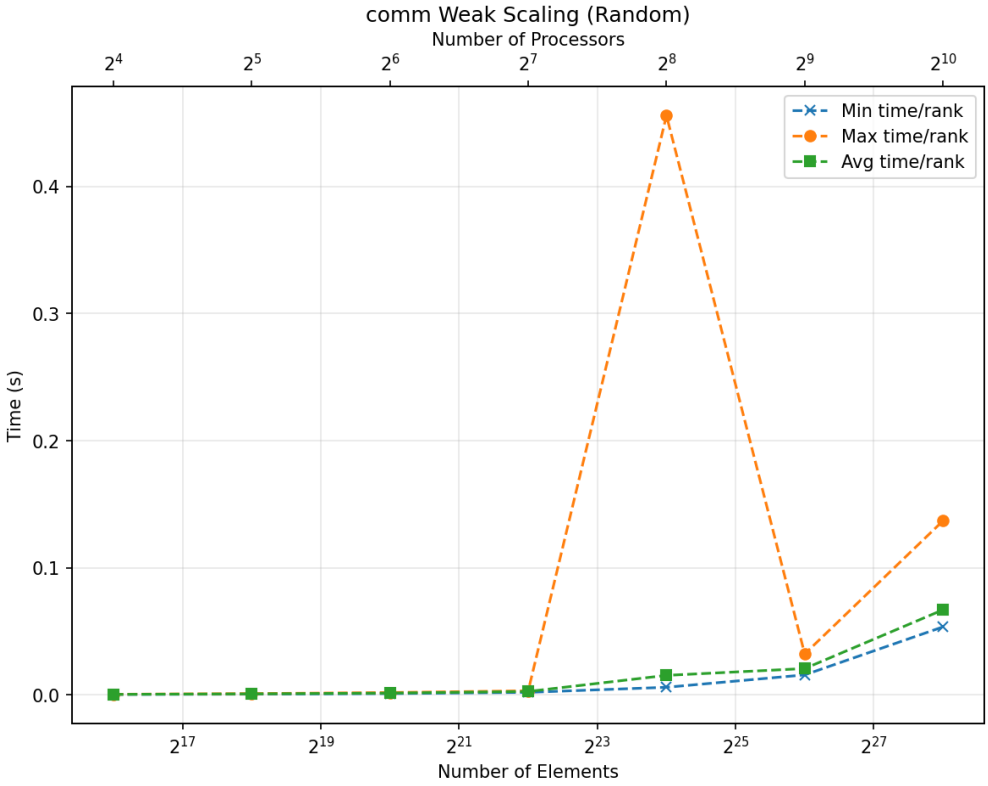
Strong Scaling Speedup Plots

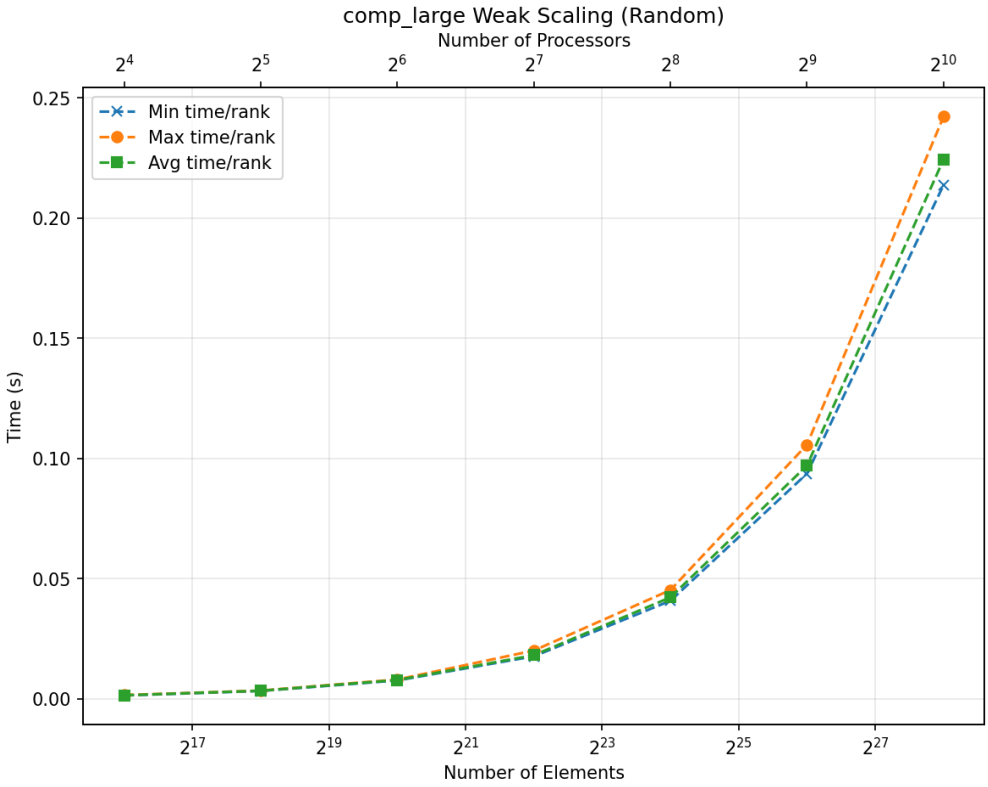


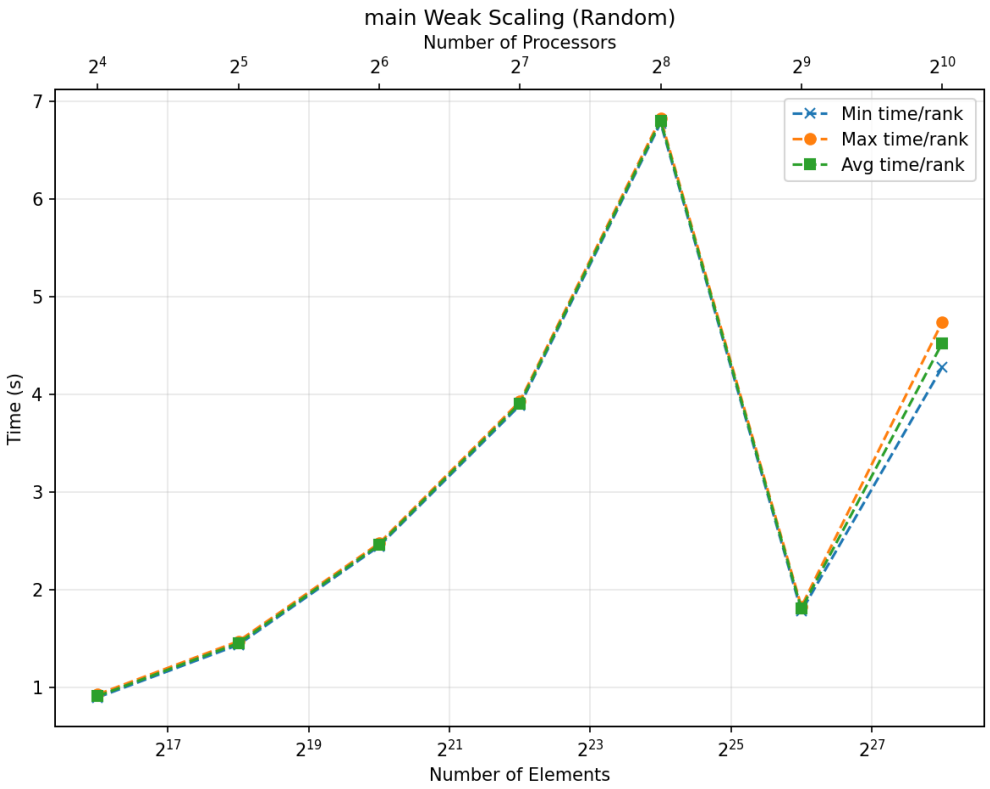




Weak Scaling Plots


comm Weak Scaling (Random)
Number of Processors
2^4 2^5 2^6 2^7 2^8 2^9 2^{10}
Min time/rank
Max time/rank
Avg time/rank
Time (s)
0.0 0.1 0.2 0.3 0.4
2^{17} 2^{19} 2^{21} 2^{23} 2^{25} 2^{27}
Number of Elements


comp_large Weak Scaling (Random)
Number of Processors
2^4 2^5 2^6 2^7 2^8 2^9 2^{10}
Min time/rank
Max time/rank
Avg time/rank
Time (s)
0.00 0.05 0.10 0.15 0.20 0.25
2^{17} 2^{19} 2^{21} 2^{23} 2^{25} 2^{27}
Number of Elements

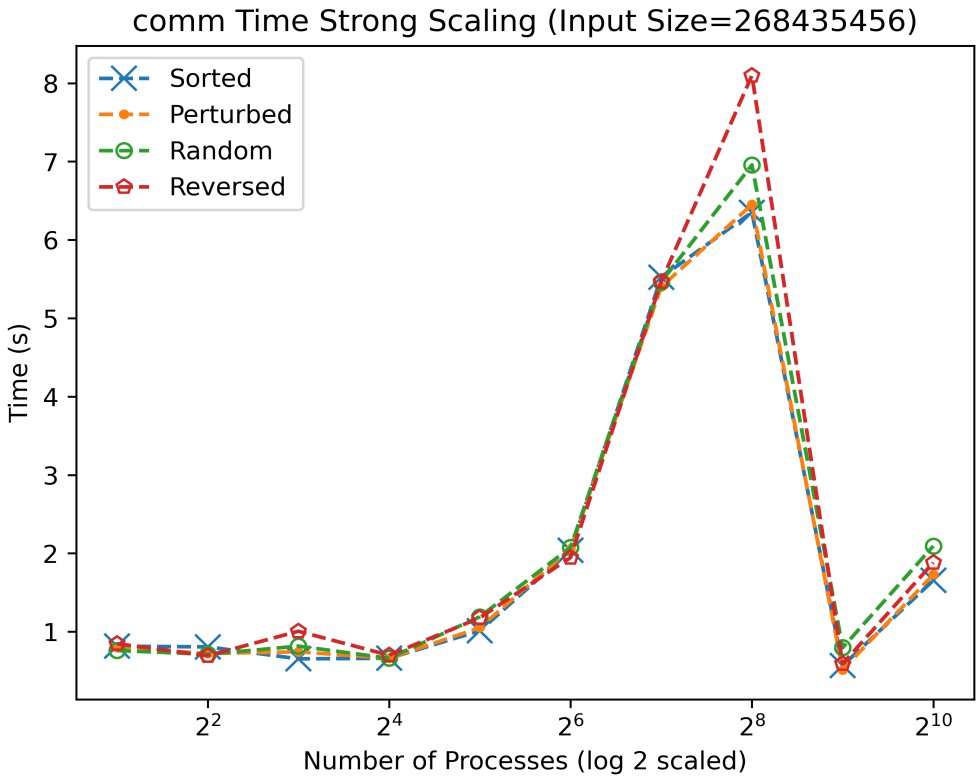


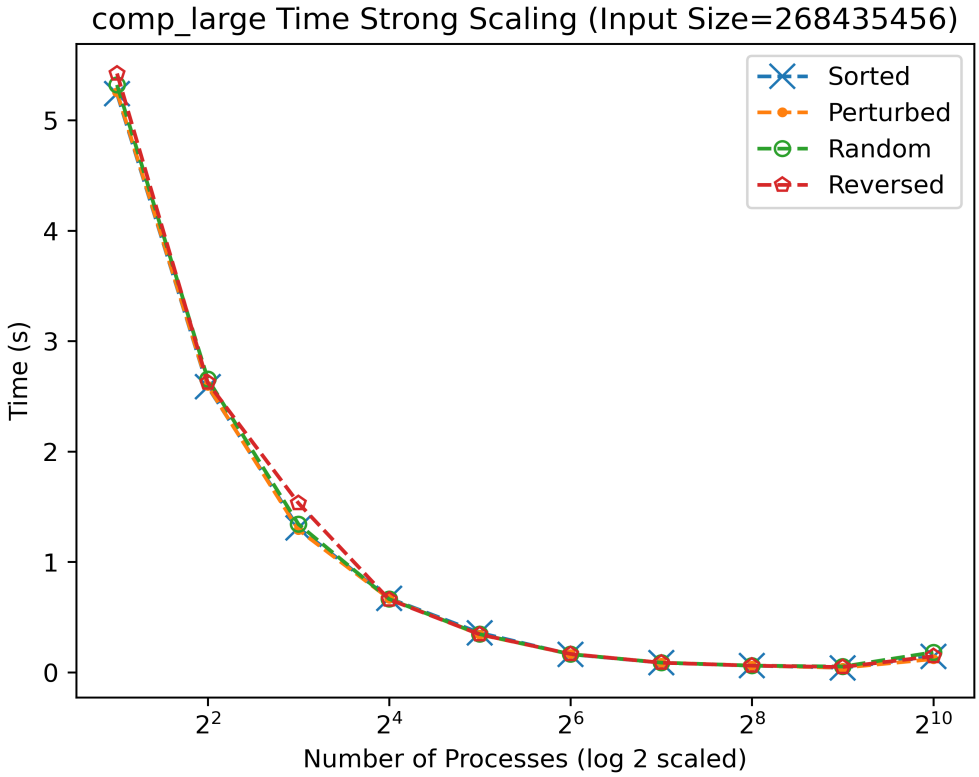
Analysis Bitonic Sort

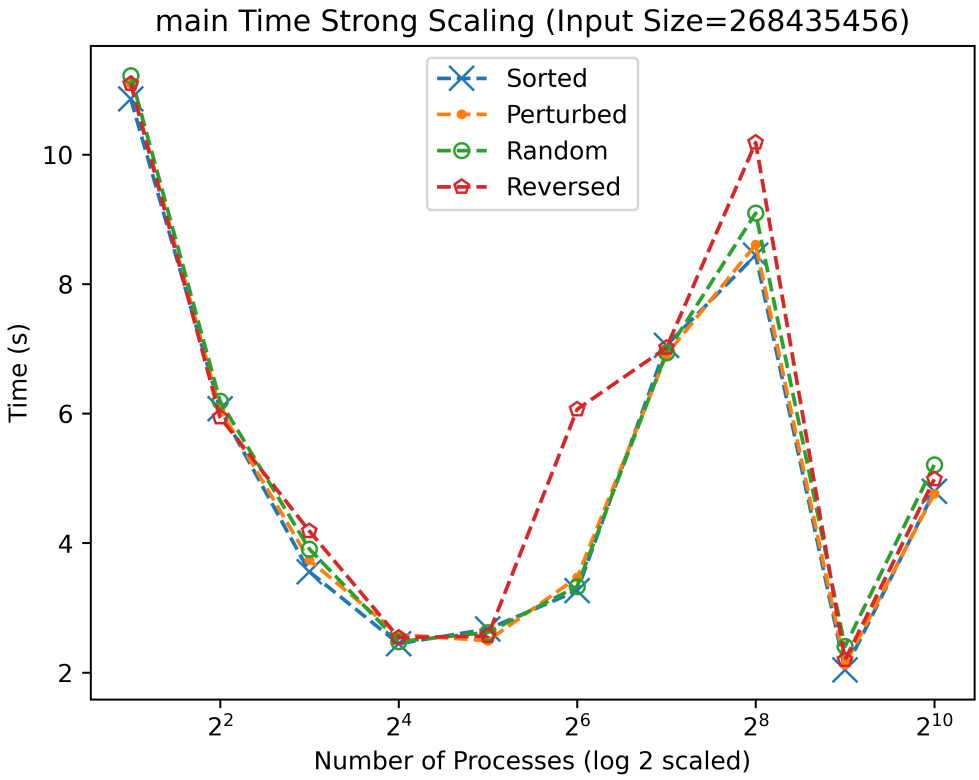
What's going on is pretty simple: the bitonic sort has a lot of communication phases. In each phase, every process sends its chunk to a partner, gets their chunk back, and then compares the two chunks to keep the half it's supposed to keep. We do that over and over as we scale up. So even when you add more processes, each one still has to go through all of those group steps, and each step touches its whole local array. That's why the runtime doesn't fall off the way you'd hope when you throw more ranks at it. Once we hit around 128–256 ranks the extra stages and the busy network start to dominate and the strong scaling curves flatten or even go up. The communication plots pop at the same points, especially for random and reverse inputs, because those inputs force real data movement in the compare split instead of letting us "win" quickly. Sorted input is always the cleanest line for that reason. The main time is just comp plus comm, so it shows both effects at once. In weak scaling we still have to run all the bitonic stages as P grows, so the time drifts upward even though the local problem size stays about the same. In short, adding more processors does not inherantly increase performance. Communication between processes takes time, and with enough processors, communication between them takes longer than computation itself.

Radix Sort

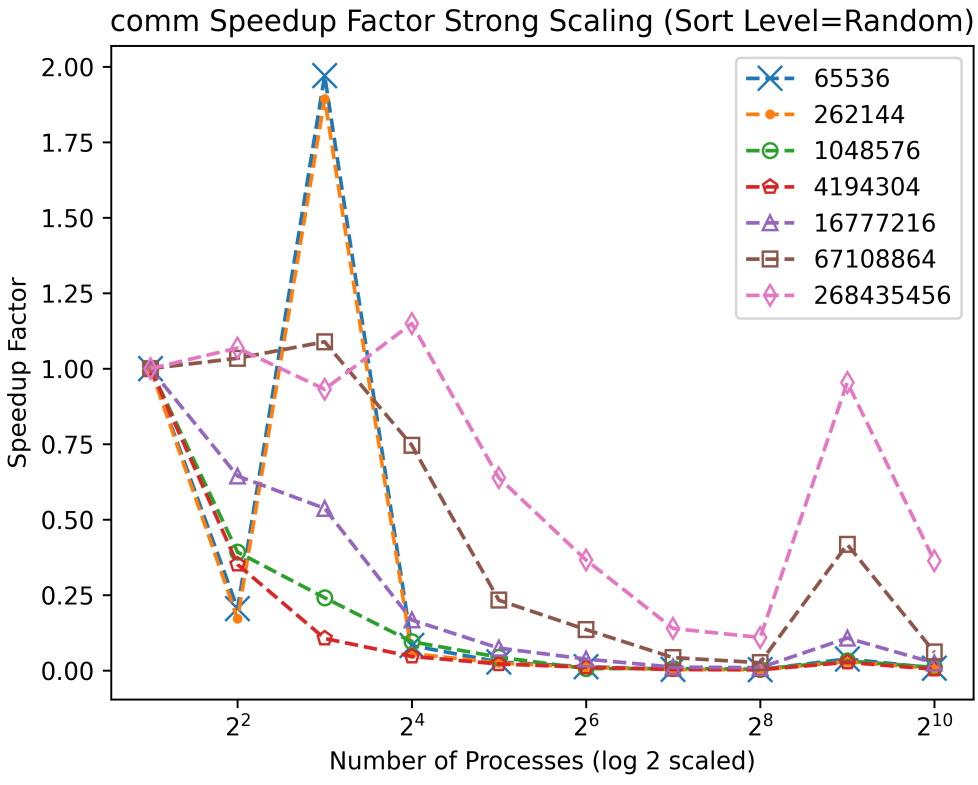
Strong Scaling Plots

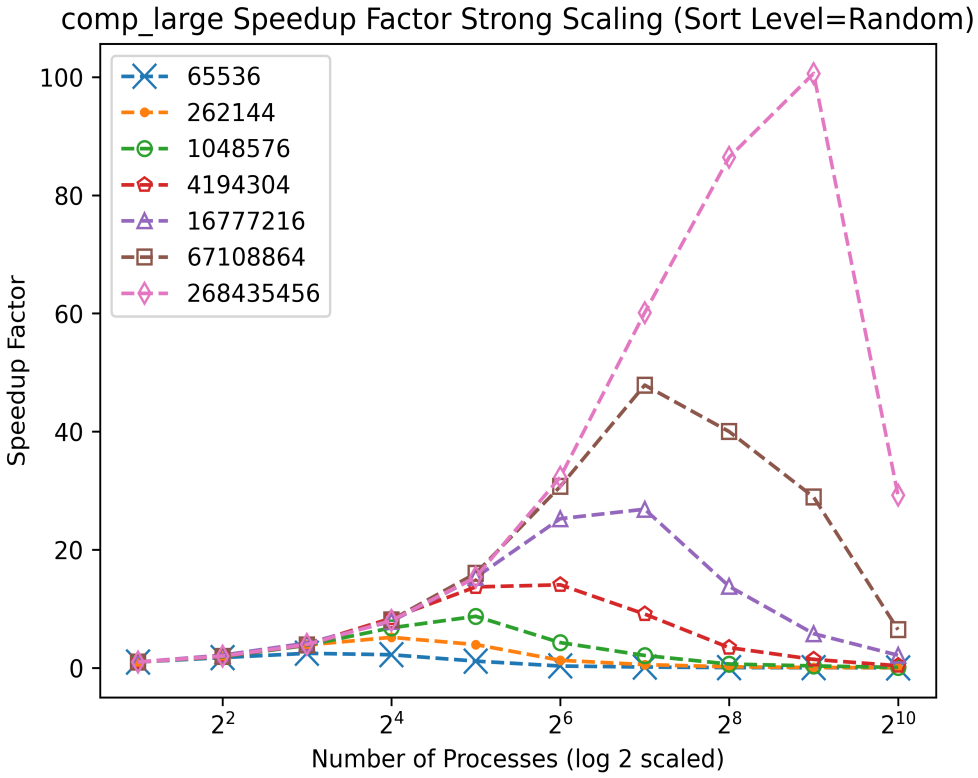

comm Time Strong Scaling (Input Size=268435456)
Sorted
Perturbed
Random
Reversed
Time (s)
Number of Processes (log 2 scaled)

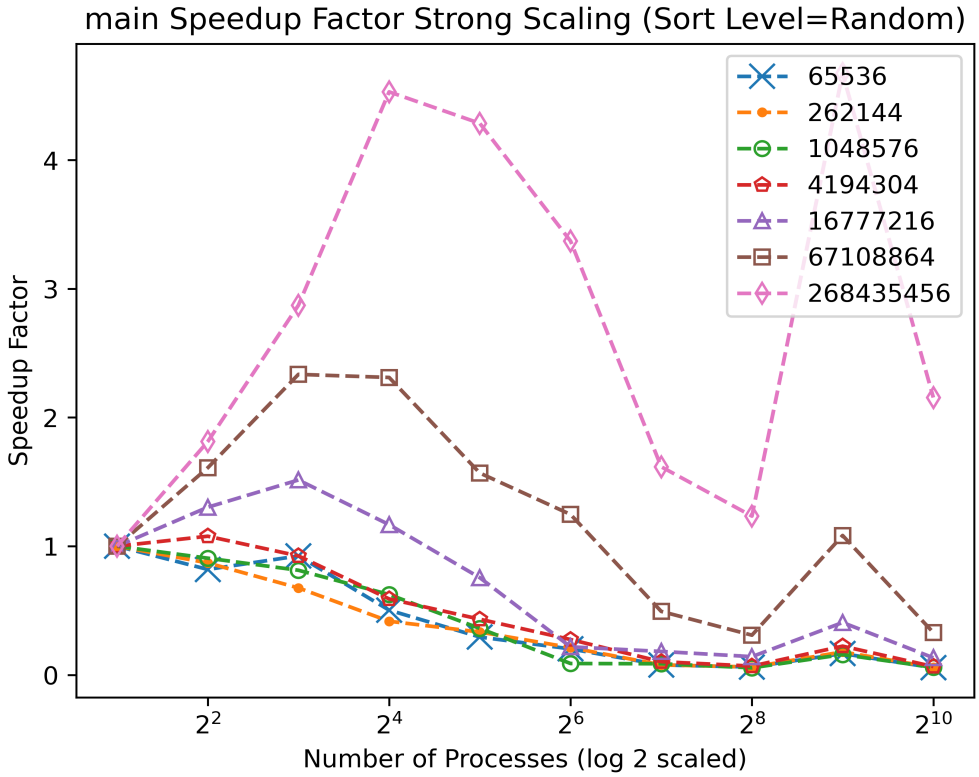

comp_large Time Strong Scaling (Input Size=268435456)
Sorted
Perturbed
Random
Reversed
Time (s)
Number of Processes (log 2 scaled)



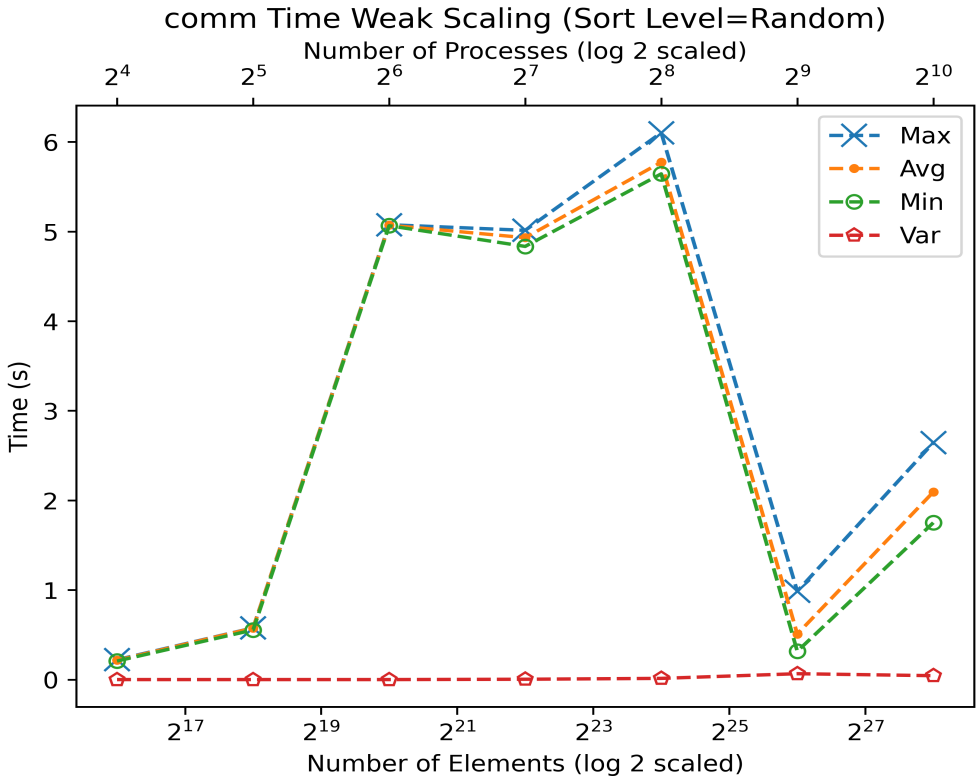
Strong Scaling Speedup Plots

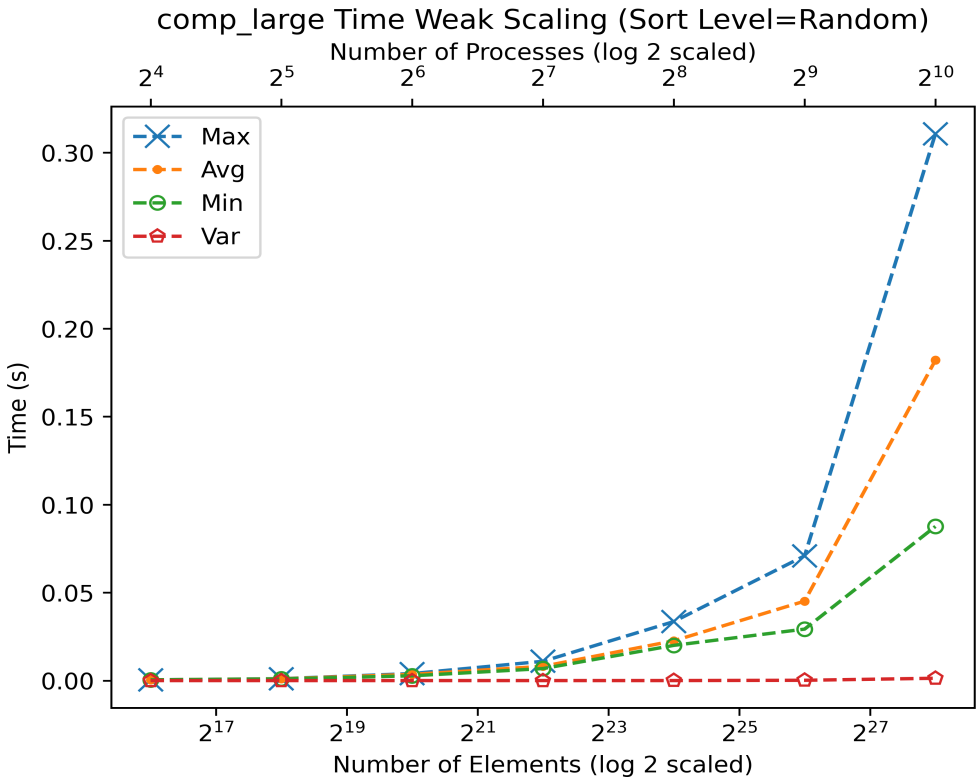


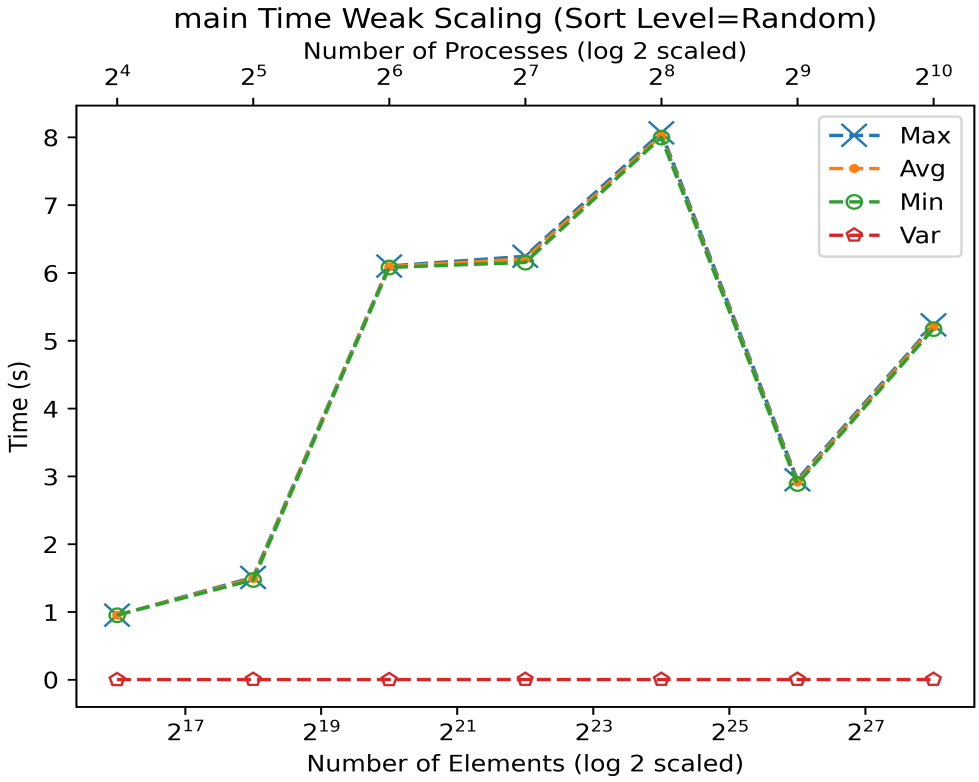




Weak Scaling Plots


comm Time Weak Scaling (Sort Level=Random)
Number of Processes (log 2 scaled)
2^4 2^5 2^6 2^7 2^8 2^9 2^{10}
Max
Avg
Min
Var
Time (s)
0 1 2 3 4 5 6
2^{17} 2^{19} 2^{21} 2^{23} 2^{25} 2^{27}
Number of Elements (log 2 scaled)


comp_large Time Weak Scaling (Sort Level=Random)
Number of Processes (log 2 scaled)
2^4 2^5 2^6 2^7 2^8 2^9 2^{10}
Max
Avg
Min
Var
Time (s)
0.00 0.05 0.10 0.15 0.20 0.25 0.30
2^{17} 2^{19} 2^{21} 2^{23} 2^{25} 2^{27}
Number of Elements (log 2 scaled)

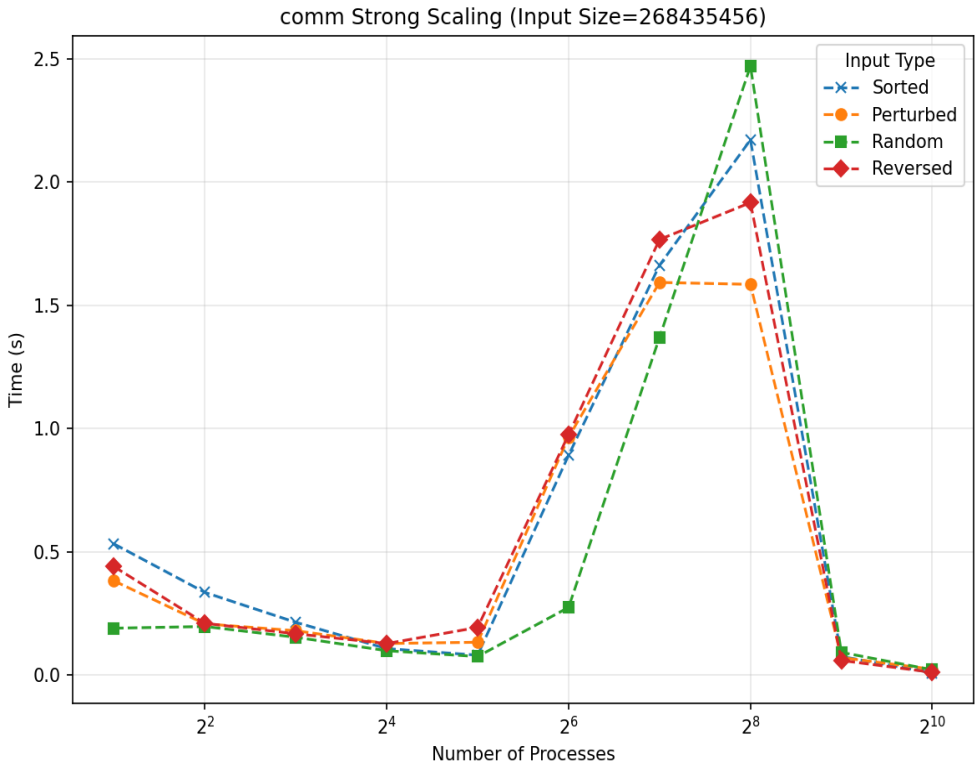


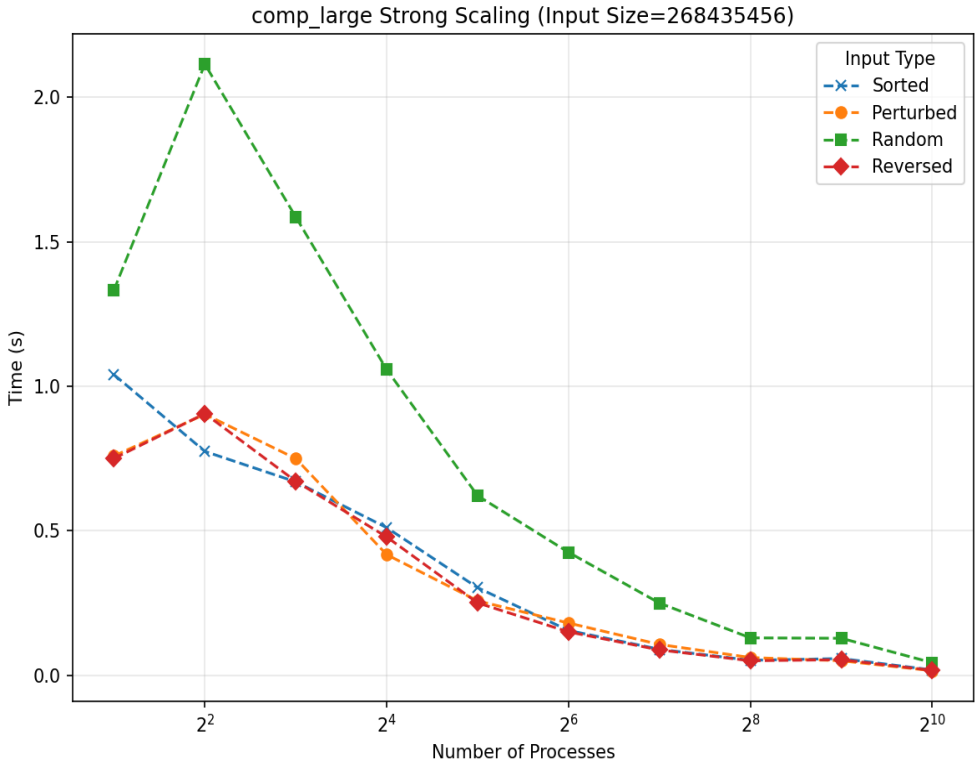
Analysis Radix Sort

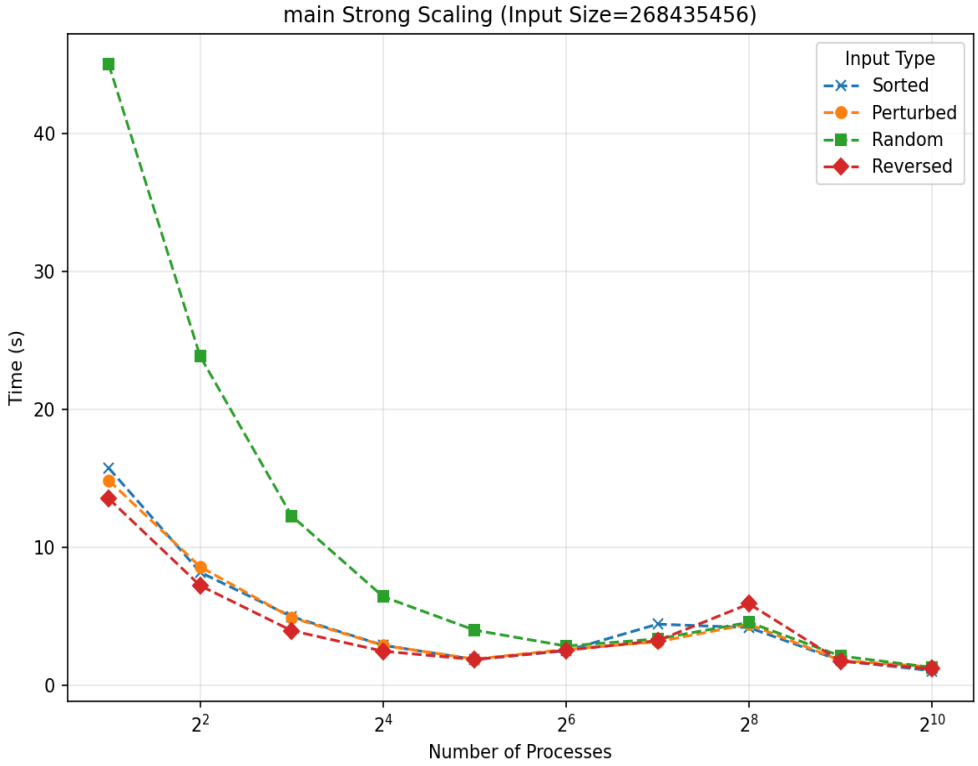
The speedup factor doesn't vary much between input type (sorted, perturbed, etc.) because radix sort doesn't have an inbuilt mechanism to check the sorting level of the array. So, each input, even if it's already sorted, is sorted radix by radix. The communication speedup curve generally decays because increasing the number of processes causes increased communication among the processes. This is especially true for my radix sort implementation since each process could possibly send elements to every other process. Interestingly, 2^9 and above processes seem to perform well in many plots, and even have communication speedup, suggesting that grace has a more optimal resource allocation strategy after some threshold. The large computation speedup generally increases with increased process count before hitting a plateau and decreasing. Larger input sizes seem to have increased speedup compared to smaller input sizes as indicated by sharper curves. This increased speedup can be attributed to each process having to locally sort less elements since the input size is fixed. The strong scaling plot especially supports this as increasing the process count for fixed input sizes shows a decaying exponential trend. Since part of the large computation time involves computing histograms of every process, plateaus are unsurprising since computing histograms for more processes takes longer. The total or main speedup behaves similarly to the large computation time as it increases before hitting a plateau and decreasing. Increasing process count only seems to benefit input sizes of 2^24 and above as the other input sizes seem to mainly have negative speedup (i.e. slowdown). While the large computation speedup is as large as ~120, the total speedup is much more modest (~4). This is because increased communication costs cancel out the benefits of quicker computation, leading to more modest efficiency gains. The weak scaling plots especially support this as the main time looks like a replica of the comm time on a different scale. The strong scaling plots also show this balance quite well as the first half of the main time closely resembles a decaying computation time while the second half resembles an increased communication time. In other words, the strong scaling plot looks like the comp_large plot superimposed on the comm_time plot.

Merge Sort

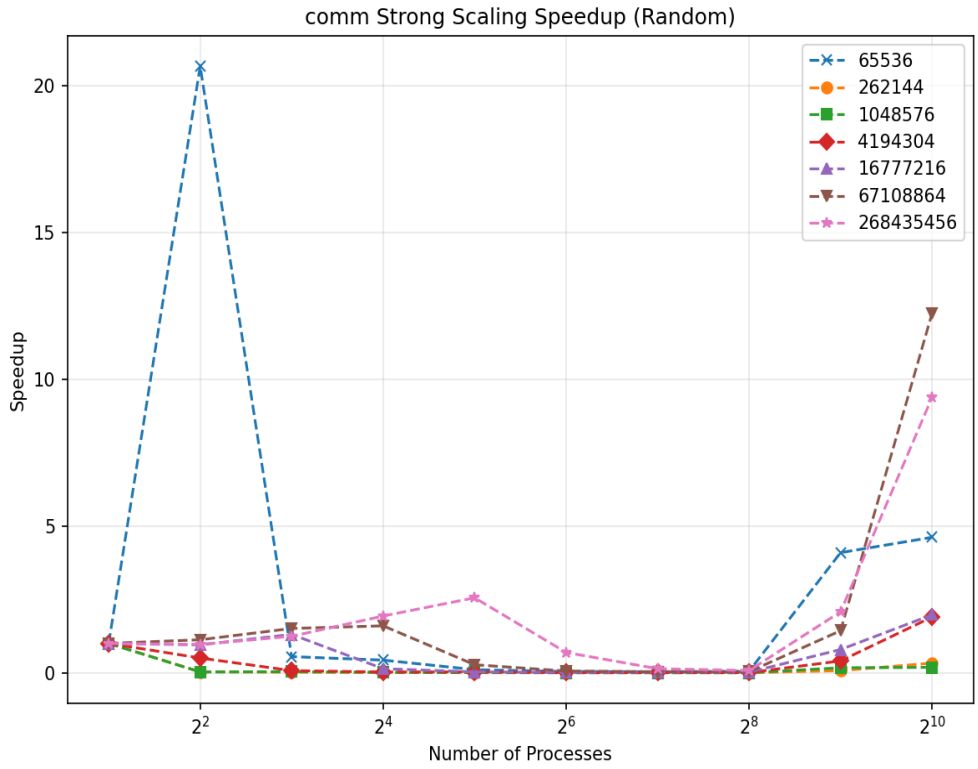
Strong Scaling Plots

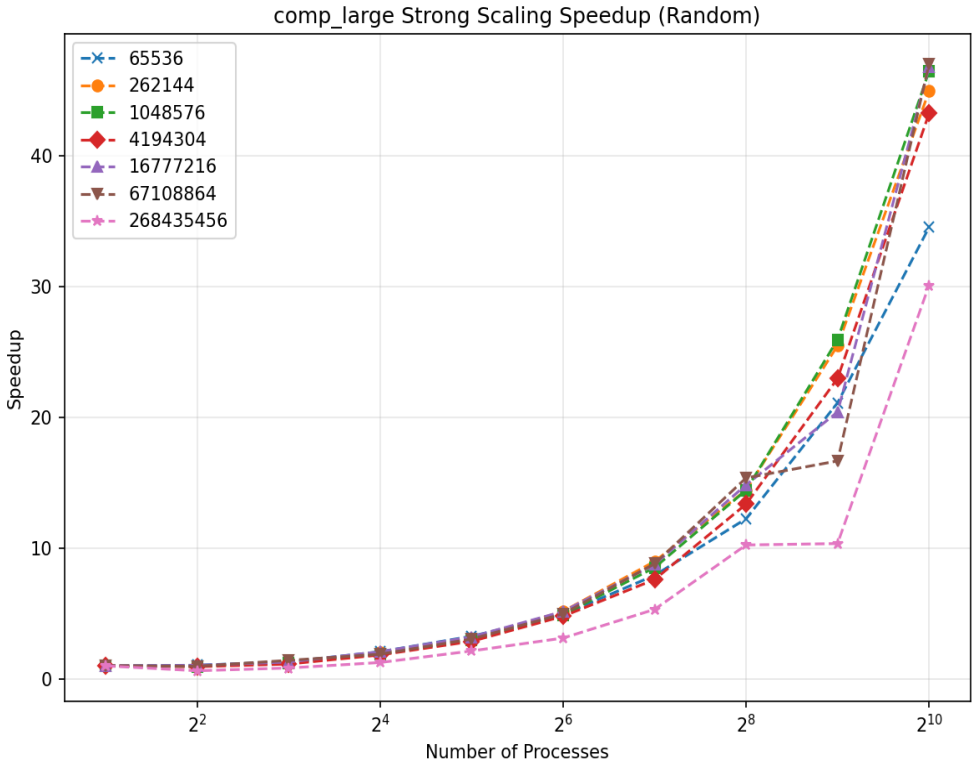


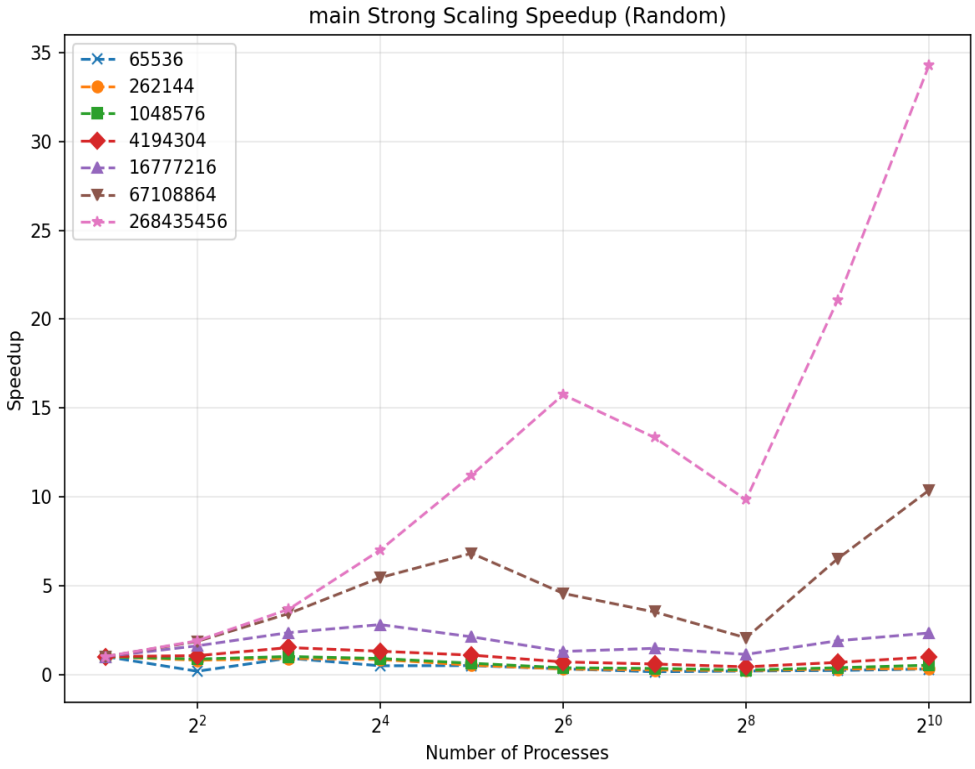




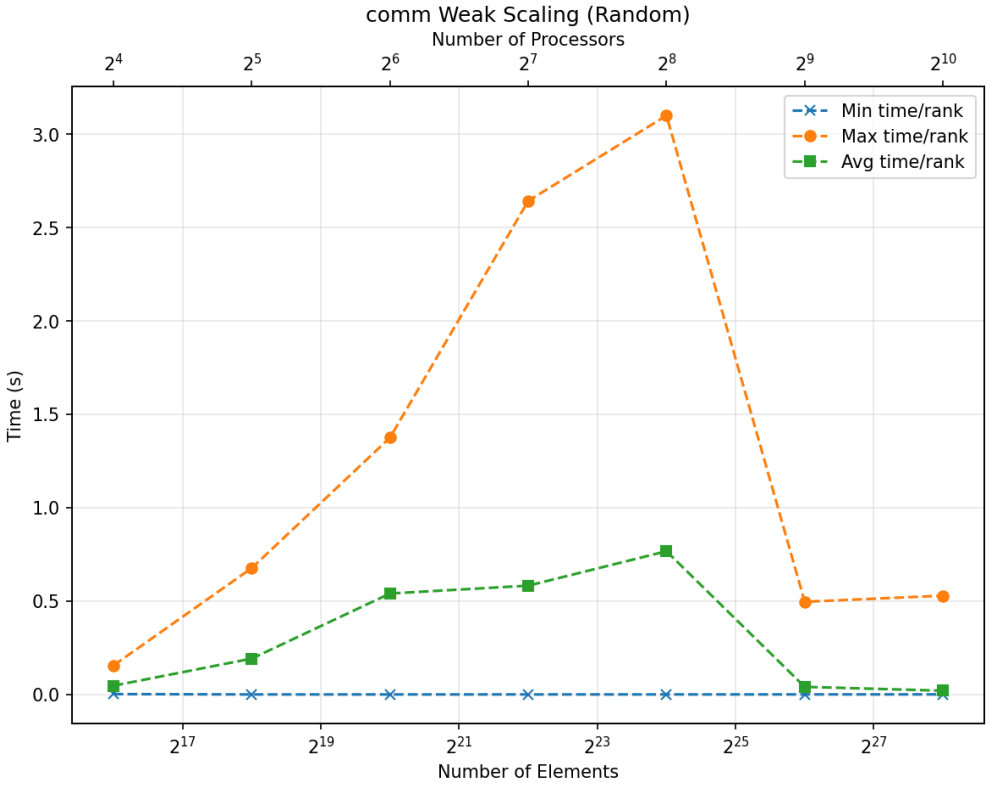
Strong Scaling Speedup Plots

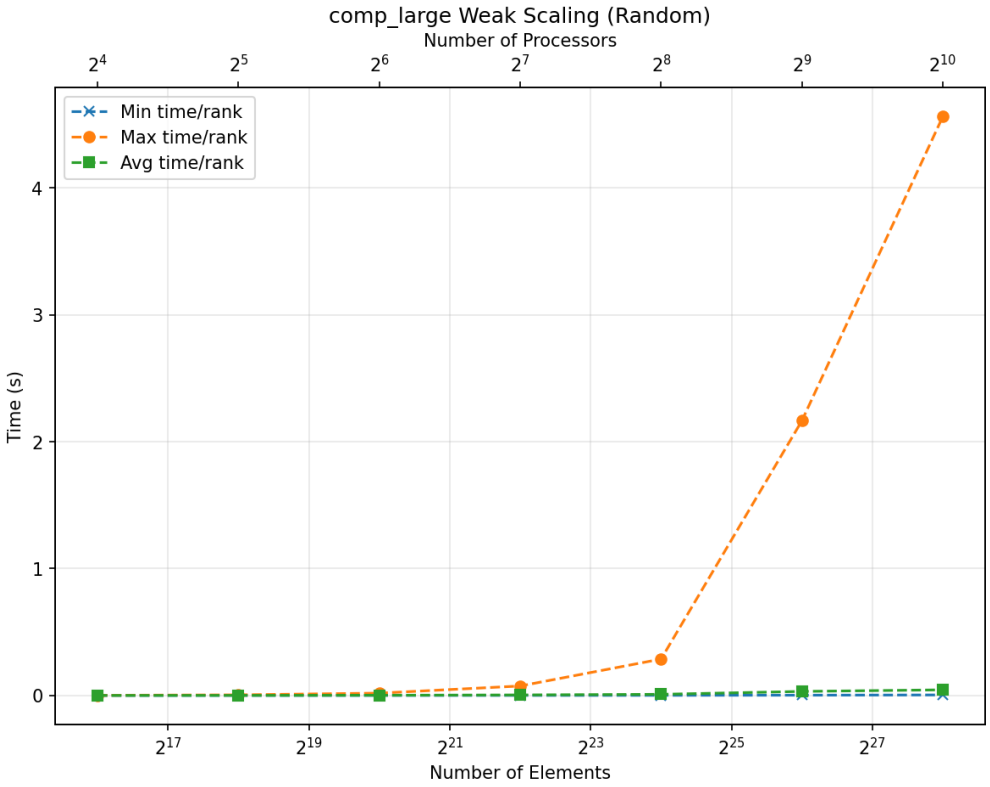


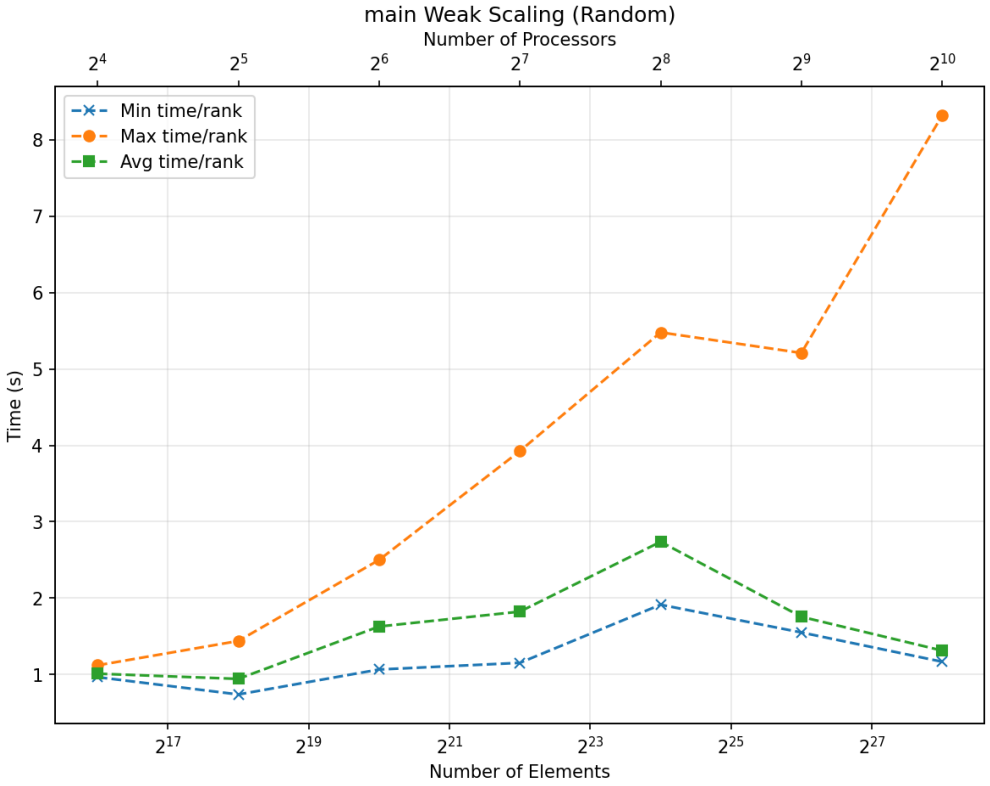




Weak Scaling Plots


comm Weak Scaling (Random)
Number of Processors
2^4 2^5 2^6 2^7 2^8 2^9 2^{10}
Min time/rank
Max time/rank
Avg time/rank
Time (s)
0.0 0.5 1.0 1.5 2.0 2.5 3.0
2^{17} 2^{19} 2^{21} 2^{23} 2^{25} 2^{27}
Number of Elements


comp_large Weak Scaling (Random)
Number of Processors
2^4 2^5 2^6 2^7 2^8 2^9 2^{10}
Min time/rank
Max time/rank
Avg time/rank
Time (s)
0 1 2 3 4
2^{17} 2^{19} 2^{21} 2^{23} 2^{25} 2^{27}
Number of Elements



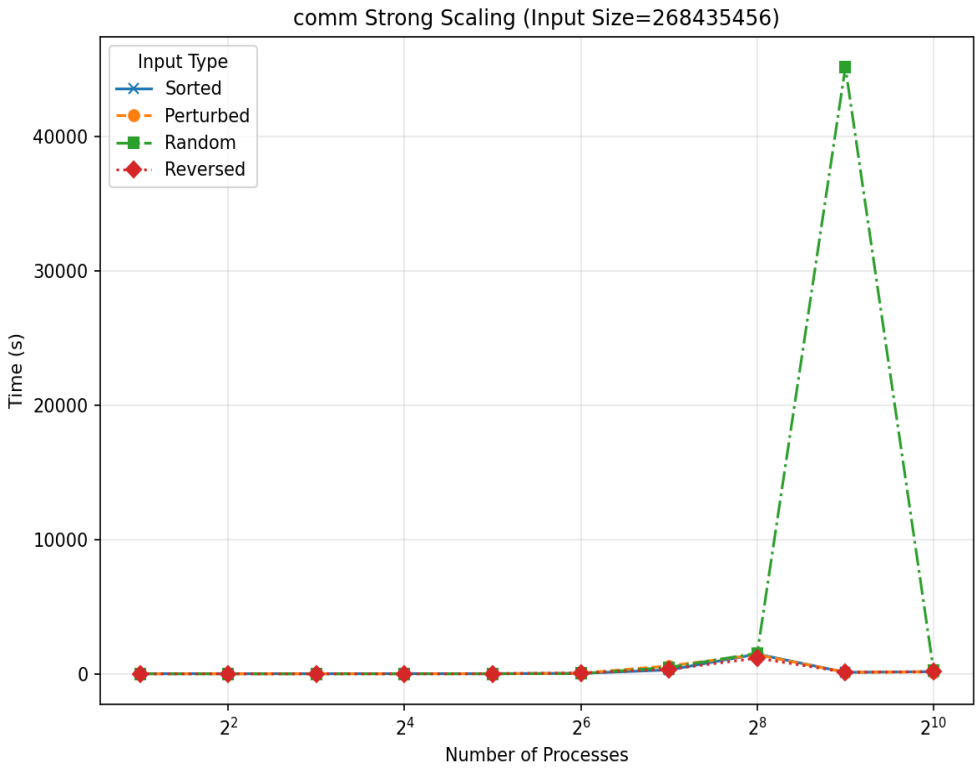
Analysis - Merge Sort

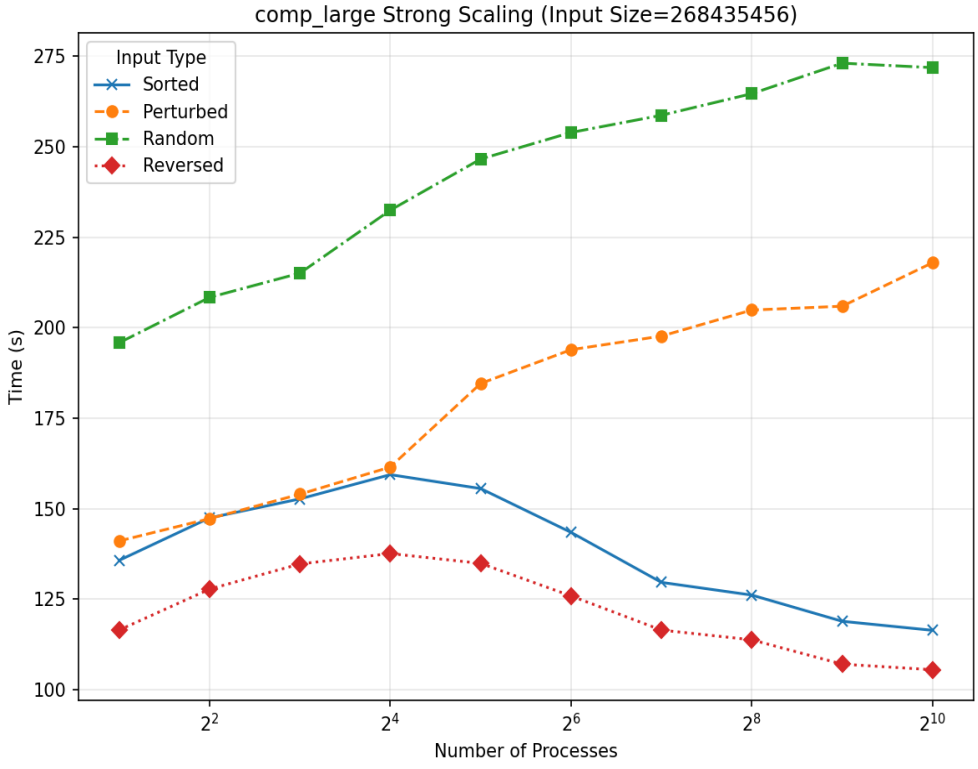
For comm strong scaling, increasing the number of processor means more communication being done by each processor. As we increase the number of processors, we start by sending and reciving smaller arrays which cascade into sending larger and reciving larger arrays. This could be causing the increase in communication, because we have large amounts of processors sending and reciving larger and larger arrays. The dropoff at the end could be due to merge sort working best when the number of processors is large, such that each processor is getting small enough arrays so that its very quick initally to send and recives before it starts to getting to sending larger arrays, where it starts to slow down. This make sense since with less processors we spend more time in comp_large, but as we spend less time in comp_large, we are limited by the communcation between processors. The reason our comp_large drops at 1024 processors, but our comm also drops, could be because our array sizes are so small, that it takes very little time to merge and to send. For comp large strong scaling, we are measure the time it takes to merge arrays. Since we increase the processor count, each sorted array for each processor gets smaller, so we are merging smaller sorted arrays, which takes less time as we increase the processor count. Each processor starts with the smaller array, so merging initially is quite fast.Our main has a similar shape to our comp_large, illustrating that our comp_large mostly domiantes the main function. Comp large seems to scale well with all input sizes and input types, which makes sense since each processor is getting a smaller array to merge together as we increase the processor count. Our comm seems to scale well with only large input sizes, which tells us that only with large input sizes are we able to give each processor enough work to do, and send data efficiently. Because of this, our comm domiantes the main speed up, since comm is only scaling well with large input sizes. Our comp_large max is really high since we are reciving and sending larger and larger arrays as we increase processor count. Eventually, a rank will be sending/reciving a large array, which increase the amount of time that rank spends merging. The min and avg are low since we start with small arrays that we merge and build up to larger arrays, and most of the processors end up working with the smaller arrays. For comm we again see the dropoff for 1024 processors at a large input size, which tells us that we are only scaling well with large input sizes and large amounts of processors. Which again could be that each array is small enough

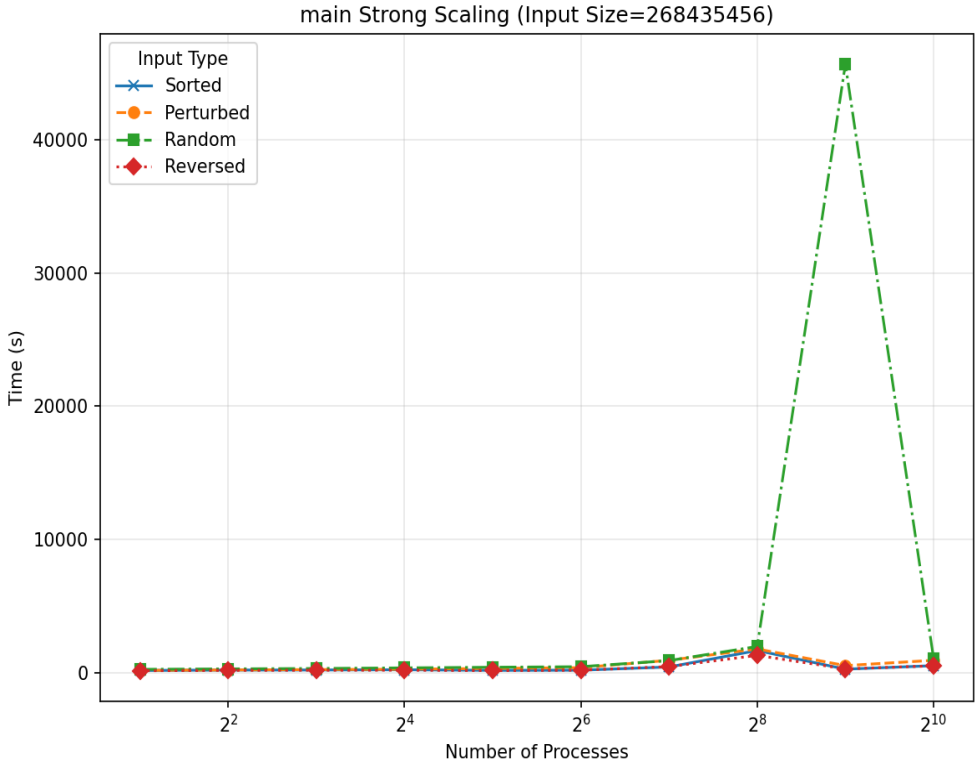
such that merging does not take long, and the array is small enough so that sending does not take long. The communication increase makes sense. Our main is mostly dominated by our comm for large input sizes and large processor counts

Sample Sort

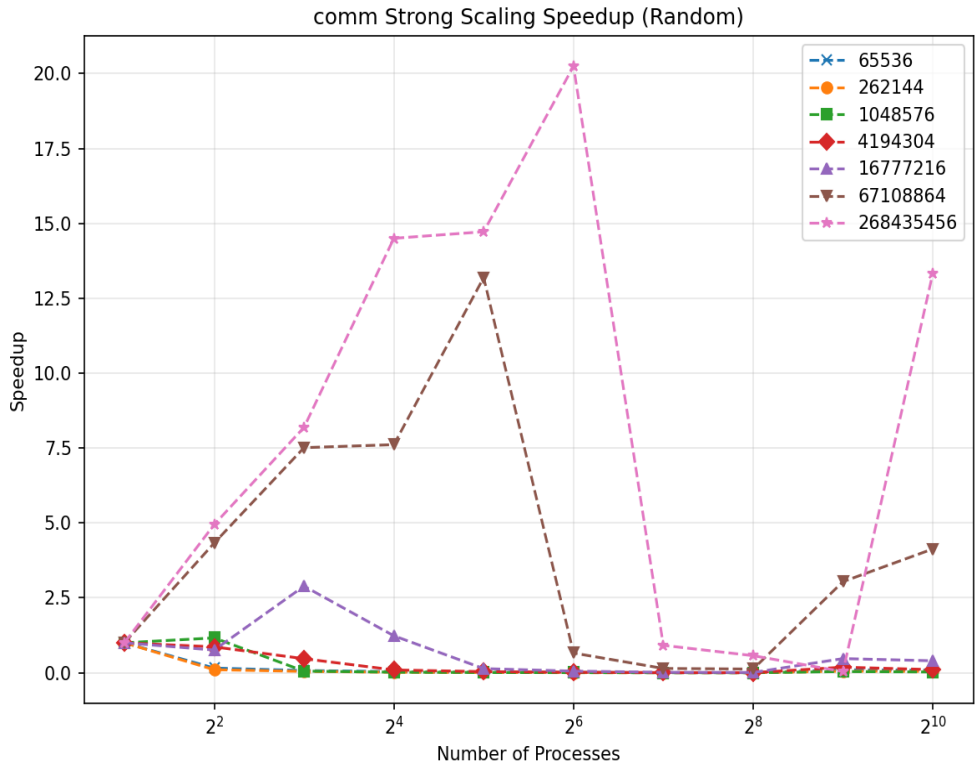
Strong Scaling Plots

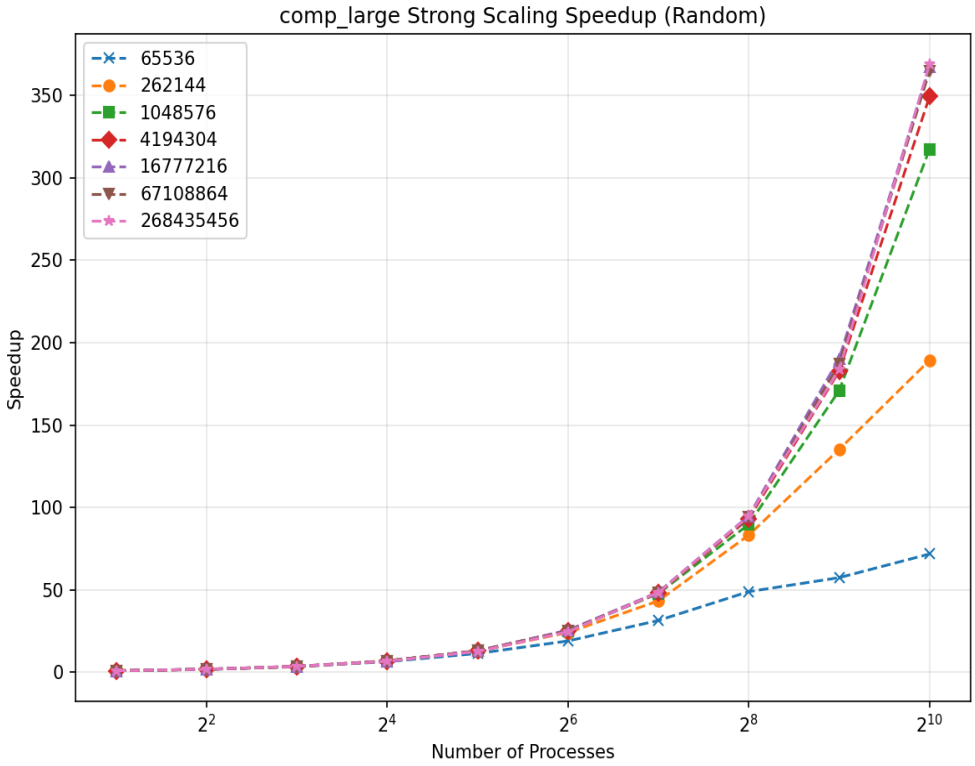


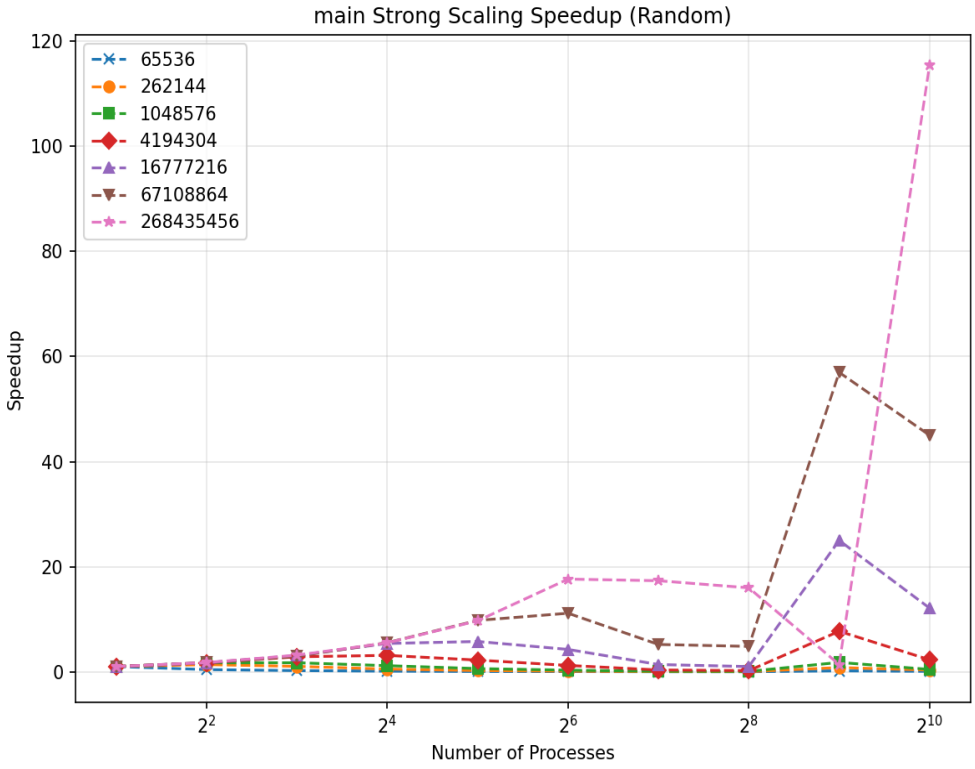




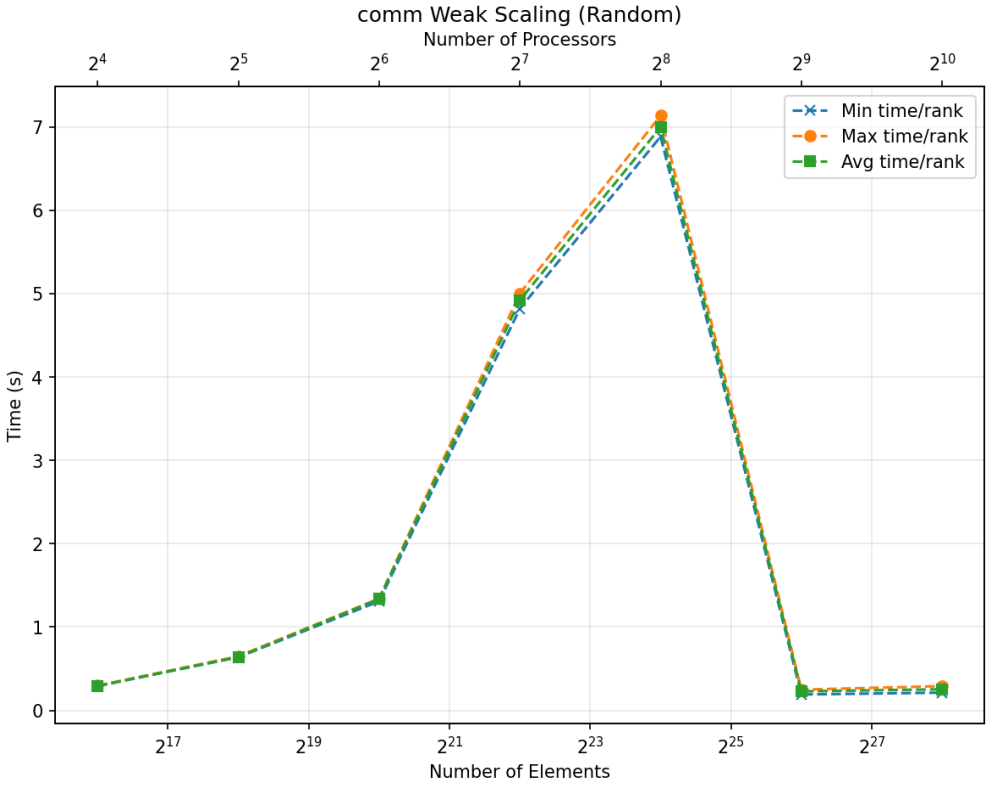
Strong Scaling Speedup Plots

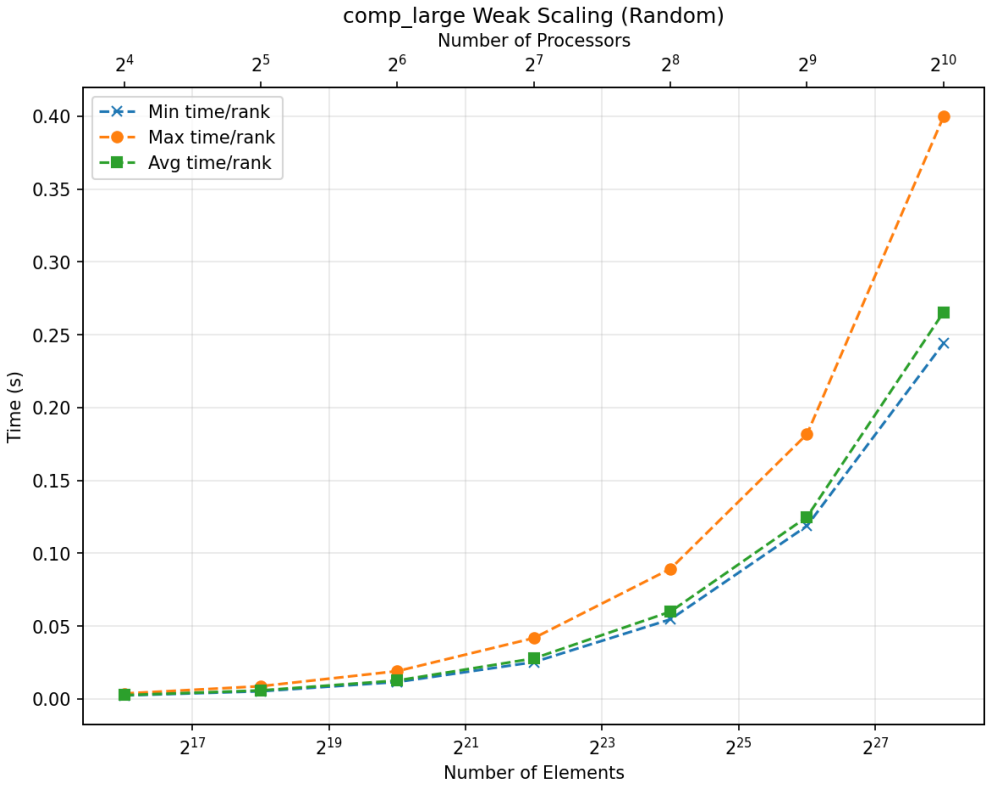


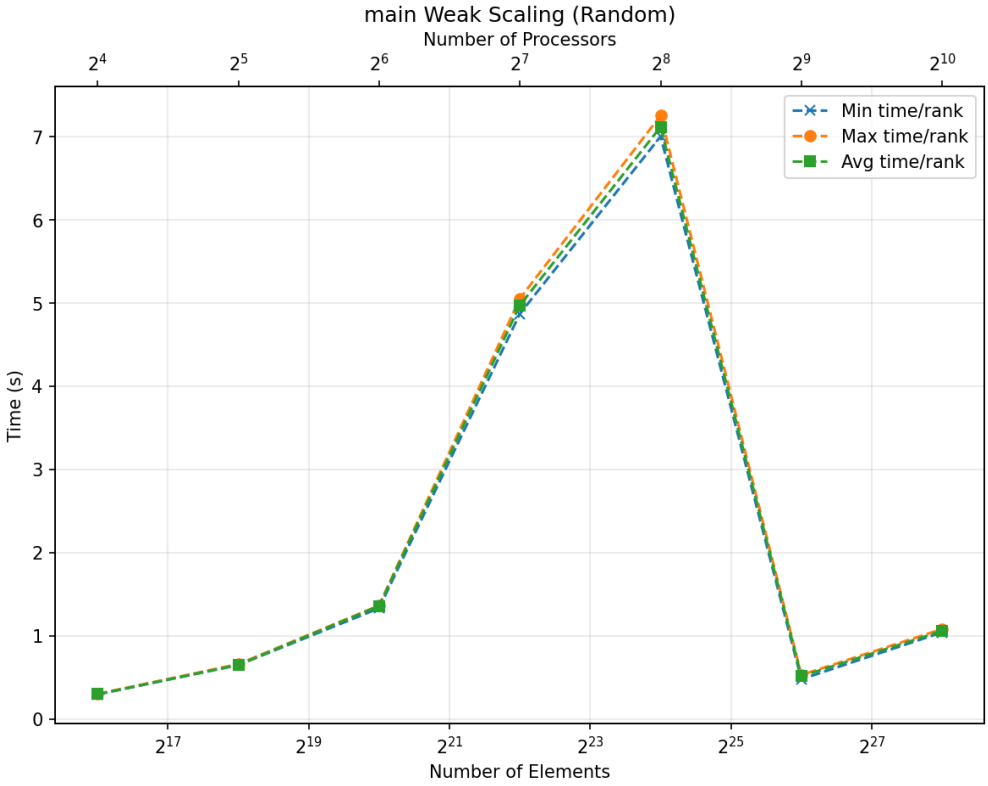




Weak Scaling Plots


comm Weak Scaling (Random)
Number of Processors
2^4 2^5 2^6 2^7 2^8 2^9 2^{10}
Min time/rank
Max time/rank
Avg time/rank
Time (s)
0 1 2 3 4 5 6 7
2^{17} 2^{19} 2^{21} 2^{23} 2^{25} 2^{27}
Number of Elements


comp_large Weak Scaling (Random)
Number of Processors
2^4 2^5 2^6 2^7 2^8 2^9 2^{10}
Min time/rank
Max time/rank
Avg time/rank
Time (s)
0.00 0.05 0.10 0.15 0.20 0.25 0.30 0.35 0.40
2^{17} 2^{19} 2^{21} 2^{23} 2^{25} 2^{27}
Number of Elements



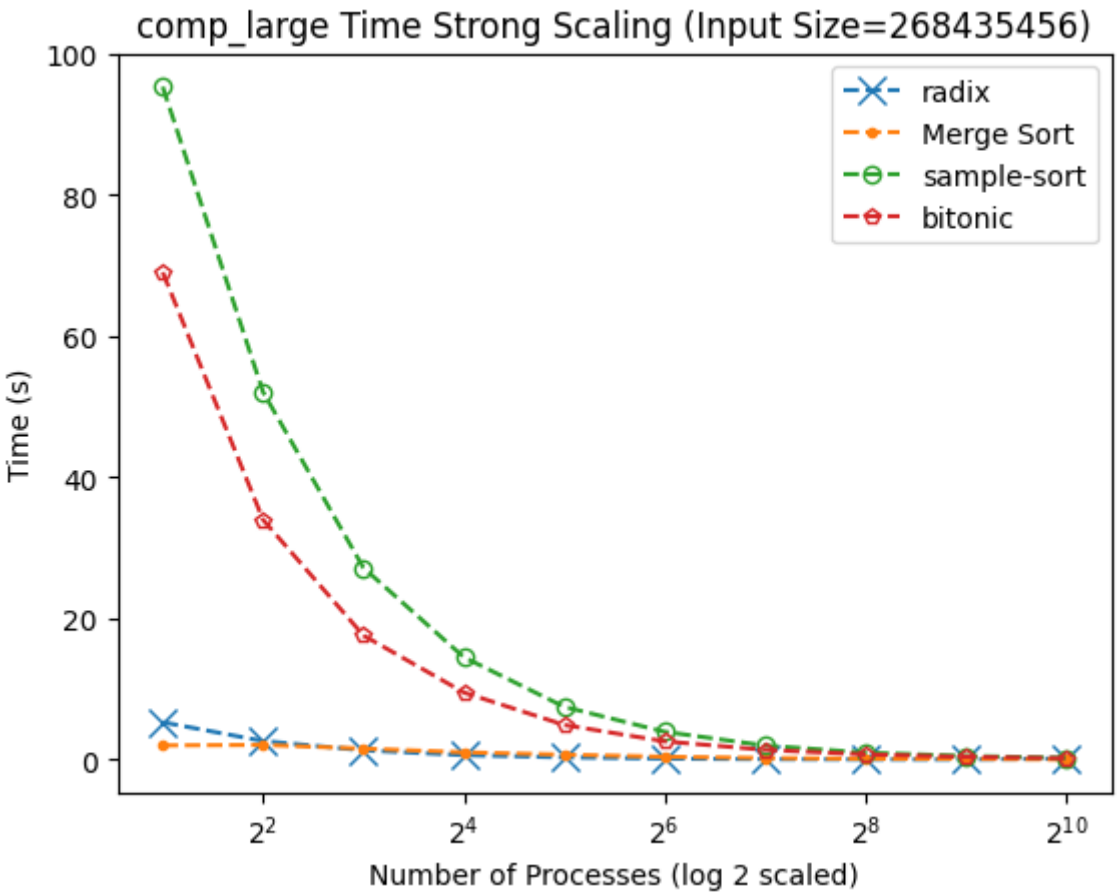
Analysis - Sample Sort

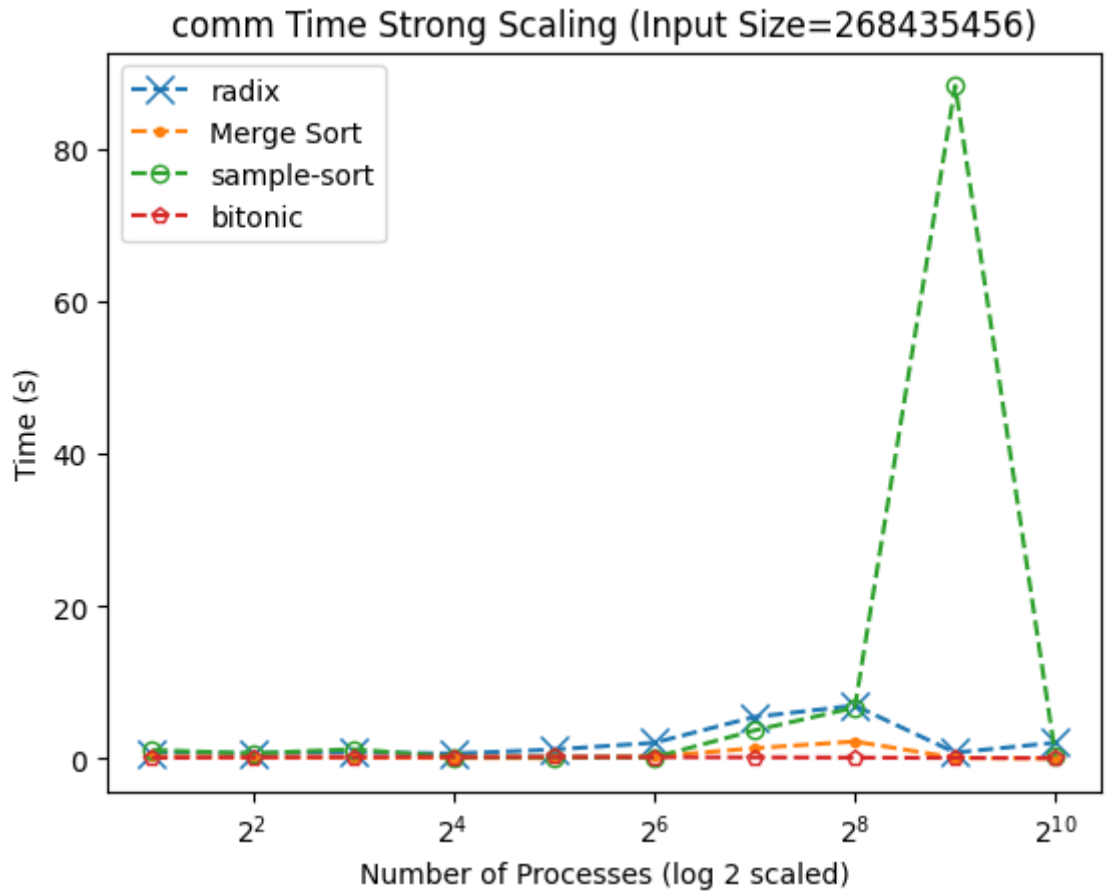
Looking at the strong scaling performance in Sample Sort, follows an upward trend with high irregularity, particularly for the random input case. Most input types (sorted, perturbed, reversed) remain relatively stable across increasing processor counts, but random input spikes dramatically near 256 processors before dropping sharply at around 512 and 1024 processors. This behavior can be caused by load imbalance and communication bottlenecks triggered by uneven partitioning in the local data for each processor. With random input, samples are not uniformly distributed across buckets, leading to heavy communication overheads during the Alltoallv redistribution. The comp_small exhibits a more predictable behavior trend, which increases as the processor count increases, which indicates that that there is a degradation of computational efficiency due to uneven data distribution presumably by poor splitter selection. This behavior is shared with the main input type, which suggests that the load imbalance is the main bottleneck in the sorting algorithm. When looking at main, it seems to indicate that the communication overhead is the main bottleneck in this algorithm. Looking at the weak scaling graphs, starting with comp_large, the time increases nearly linearly as both problem size and processor count grows. This is expected behavior as each rank must handle a proportionally similar local problem, which its local sorting cost grows as the array size grows. The main phase shows behavior consistent with the combination of both rising overall runtime with increasing scale but without major outliers. The lack of severe spikes suggests that weak scaling mitigates imbalance issues, as each rank's data size grows uniformly. This is especially apparent in the sorted and reversed respectively. Looking at the Strong Scaling Speedup graph, starting with the comm speedup, the plot shows that communication efficiency varies drastically across input sizes, which presumably reflects the network contention and nonuniform workload balance under random partitioning, where small changes in the splitter distribution can drastically affect which ranks communicate most heavily. On te other hand, comp_large speedup is nearly ideal, as we increase the number of processors, speedup grows exponentially across all input sizes. This reflects that the local sort operation is close to being embarassingly parallel, which each processor gets to sort a smaller subset of a bigger array straight off the bat, decreasing nearly perfectly with additional processor. The main speedup graph is the middle ground. The graph suggests that while

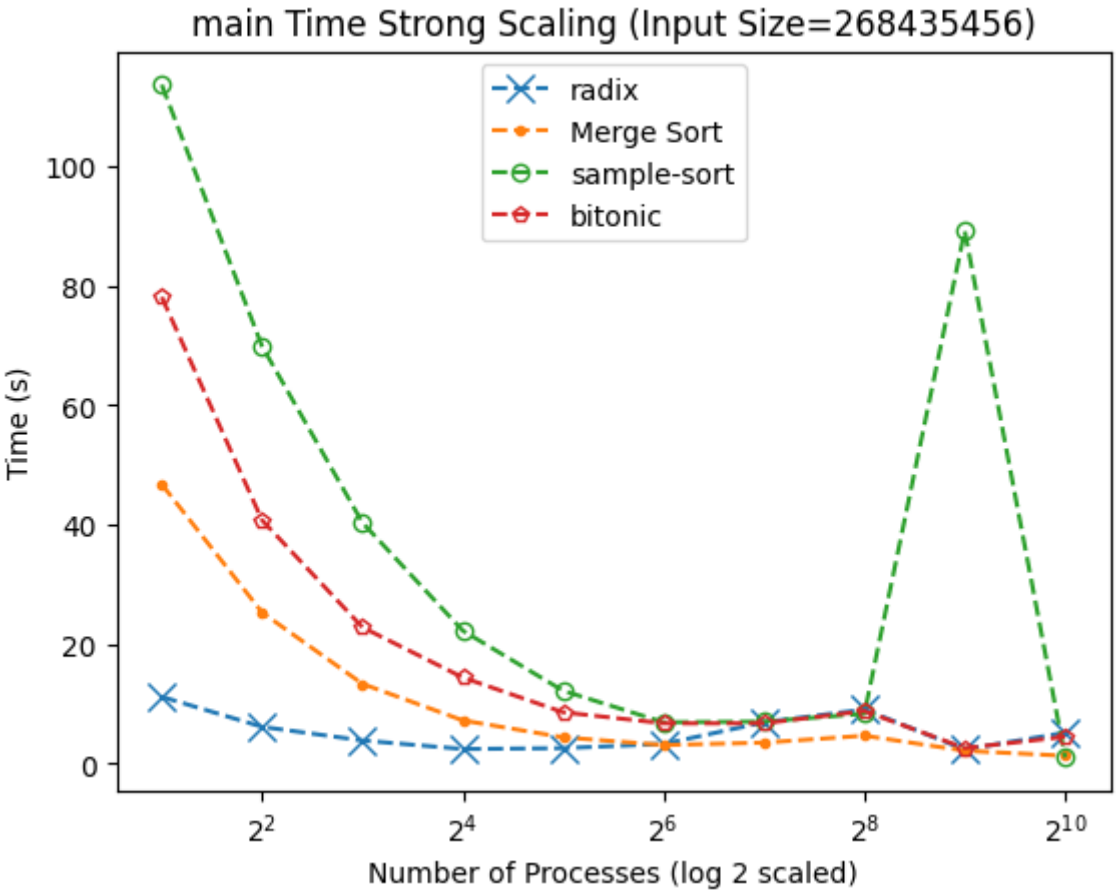
computation scales well, communication dominates the total runtime at scale, limiting the total efficiency of the algorithm.

Comparisons

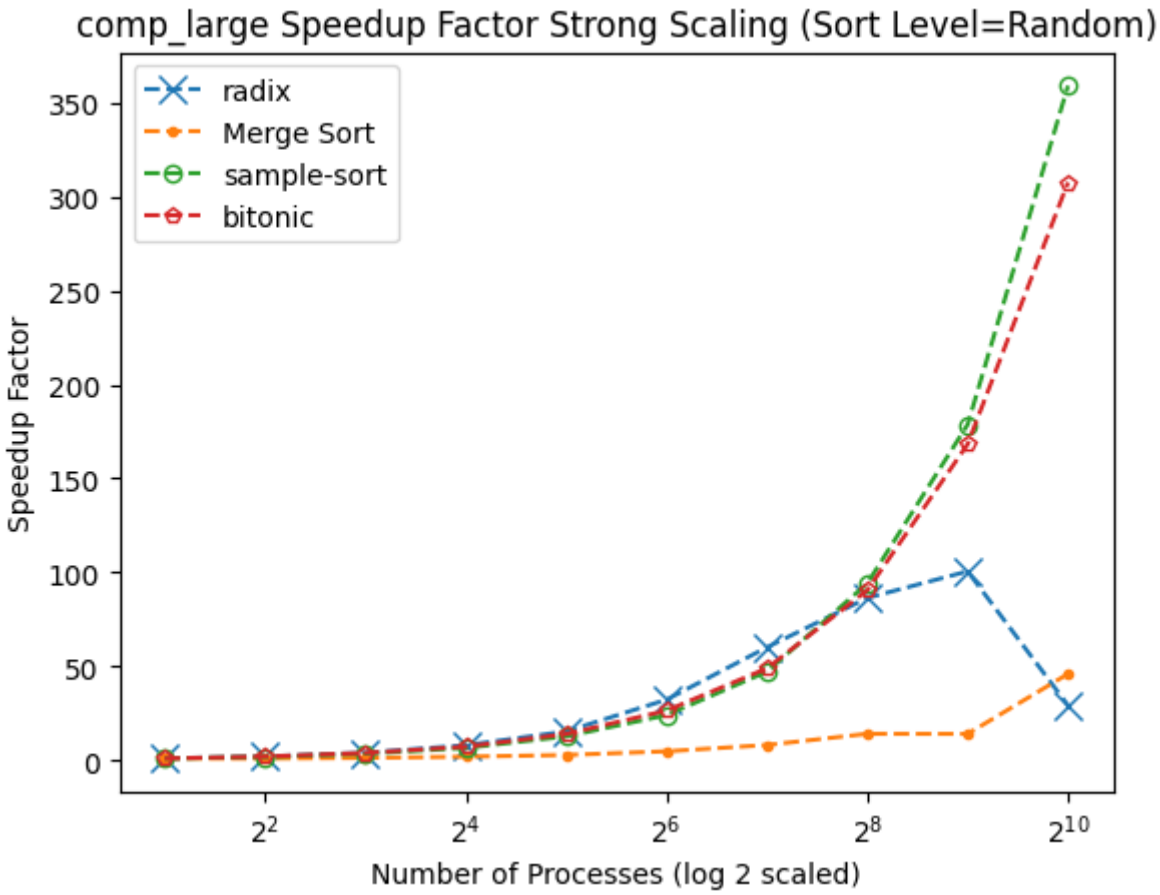
Strong Scaling Plots

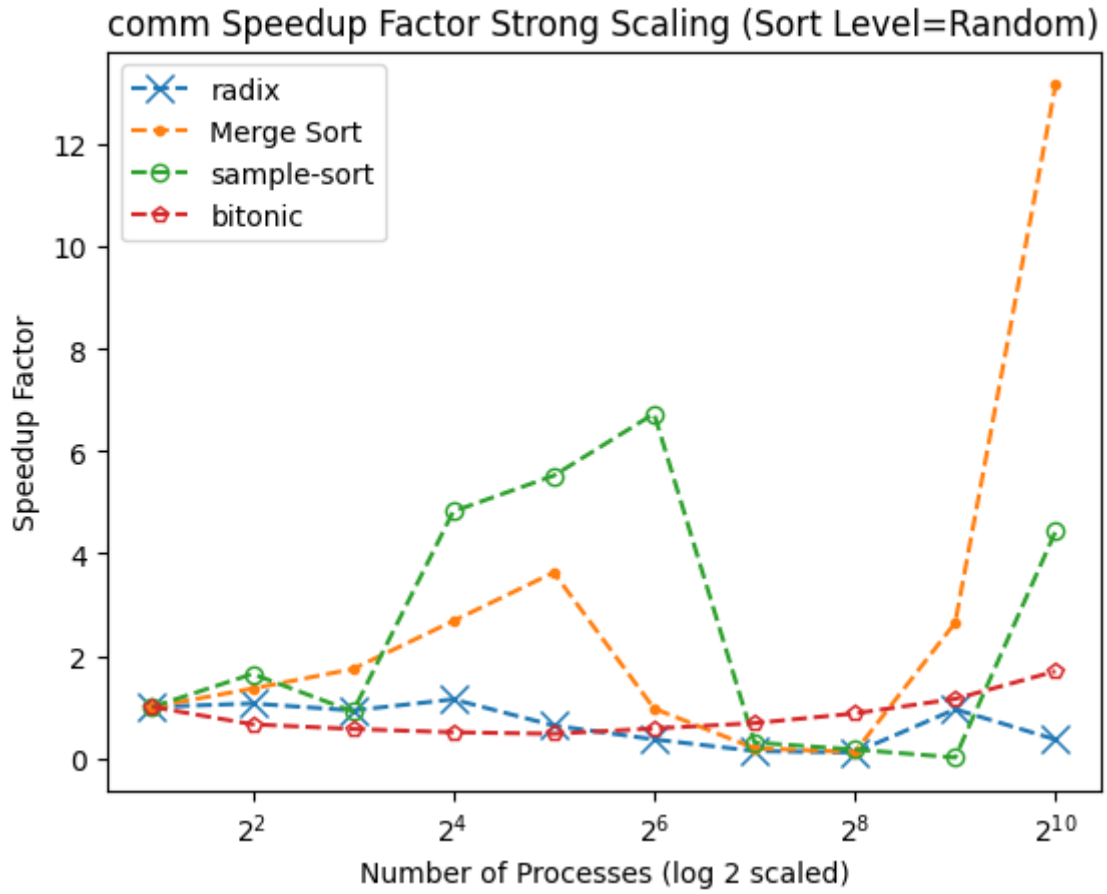


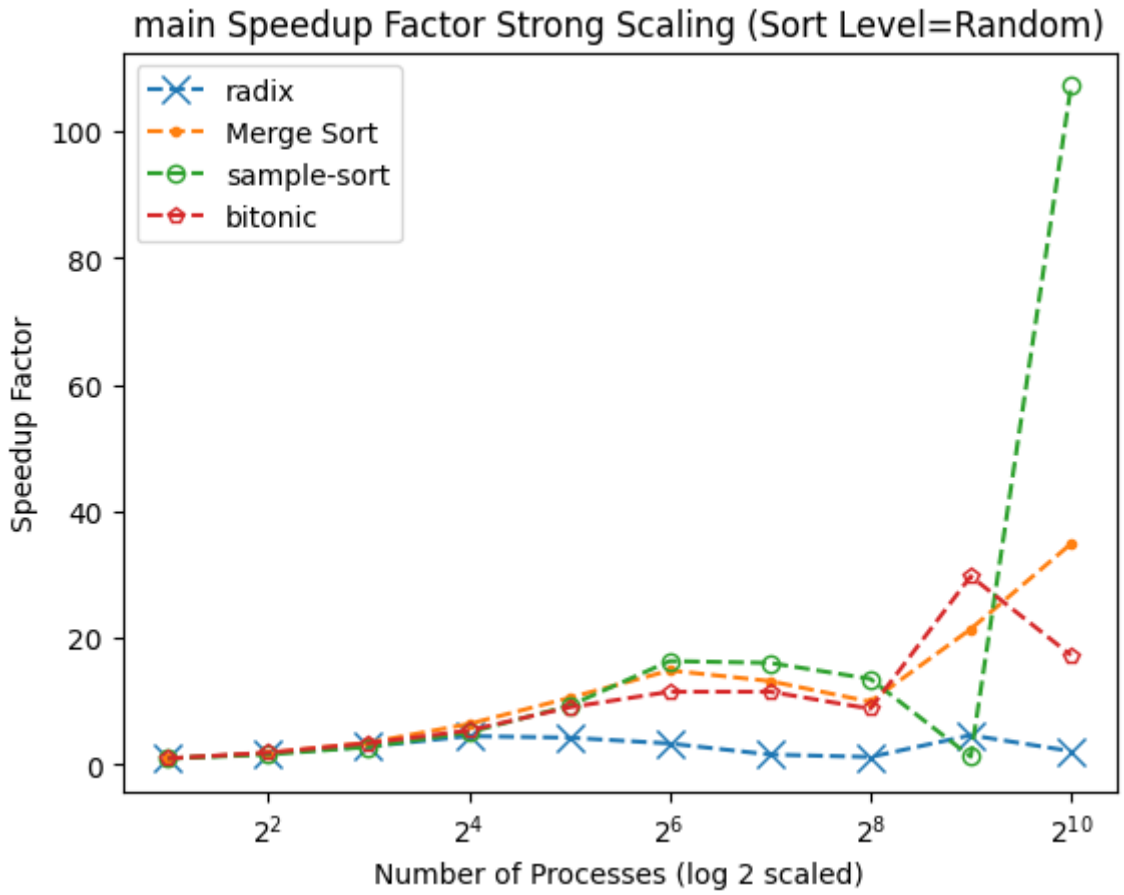




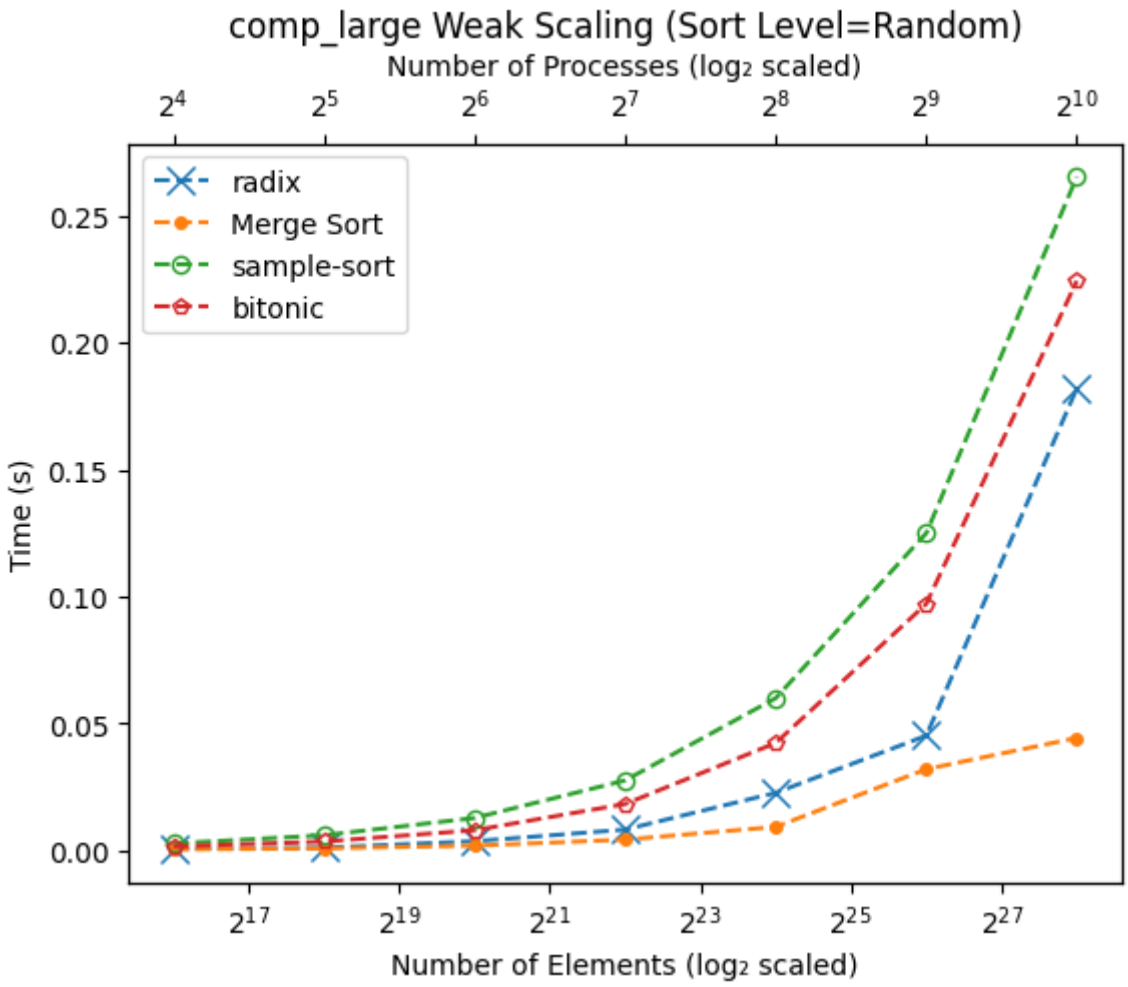
Strong Scaling Speedup Plots

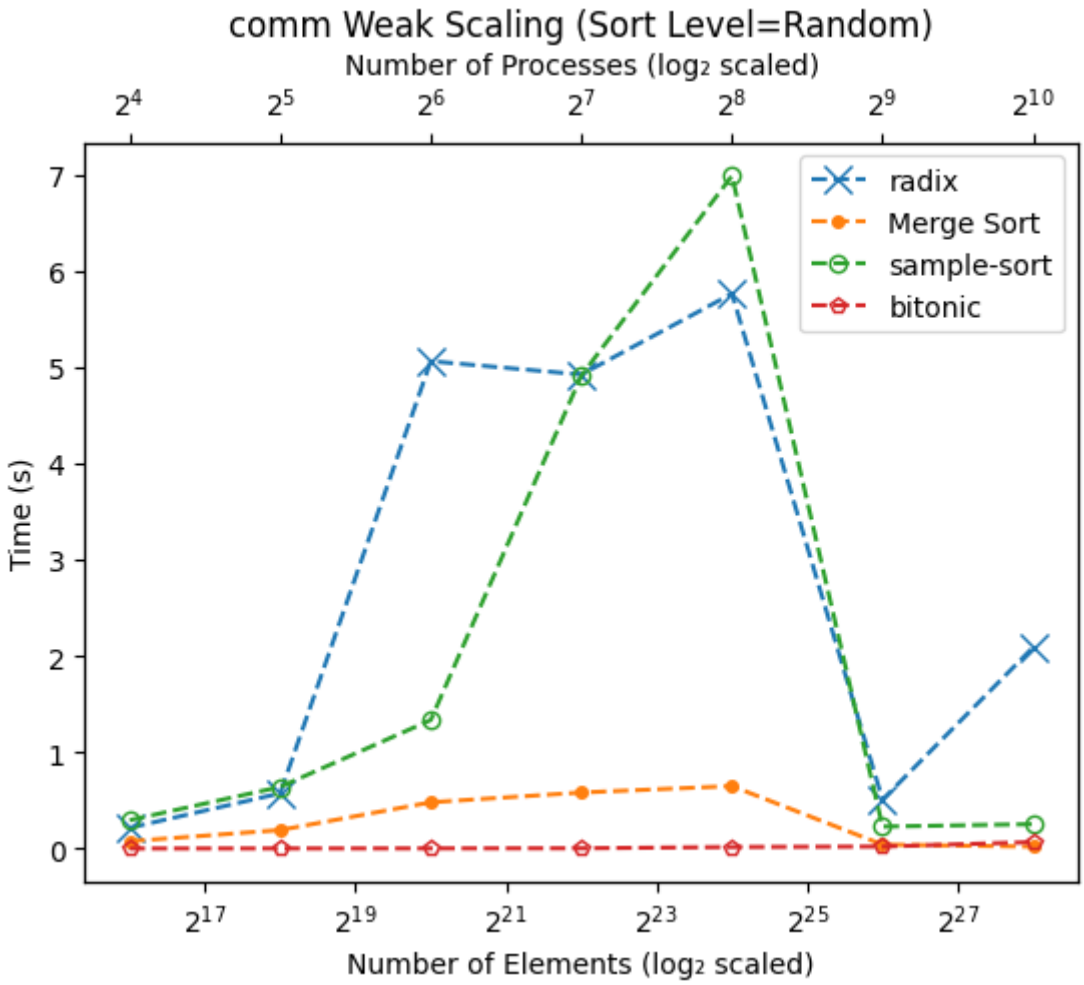

comp_large Speedup Factor Strong Scaling (Sort Level=Random)
radix
Merge Sort
sample-sort
bitonic
Speedup Factor
0
50
100
150
200
250
300
350
2^2
2^4
2^6
2^8
2^{10}
Number of Processes (log 2 scaled)

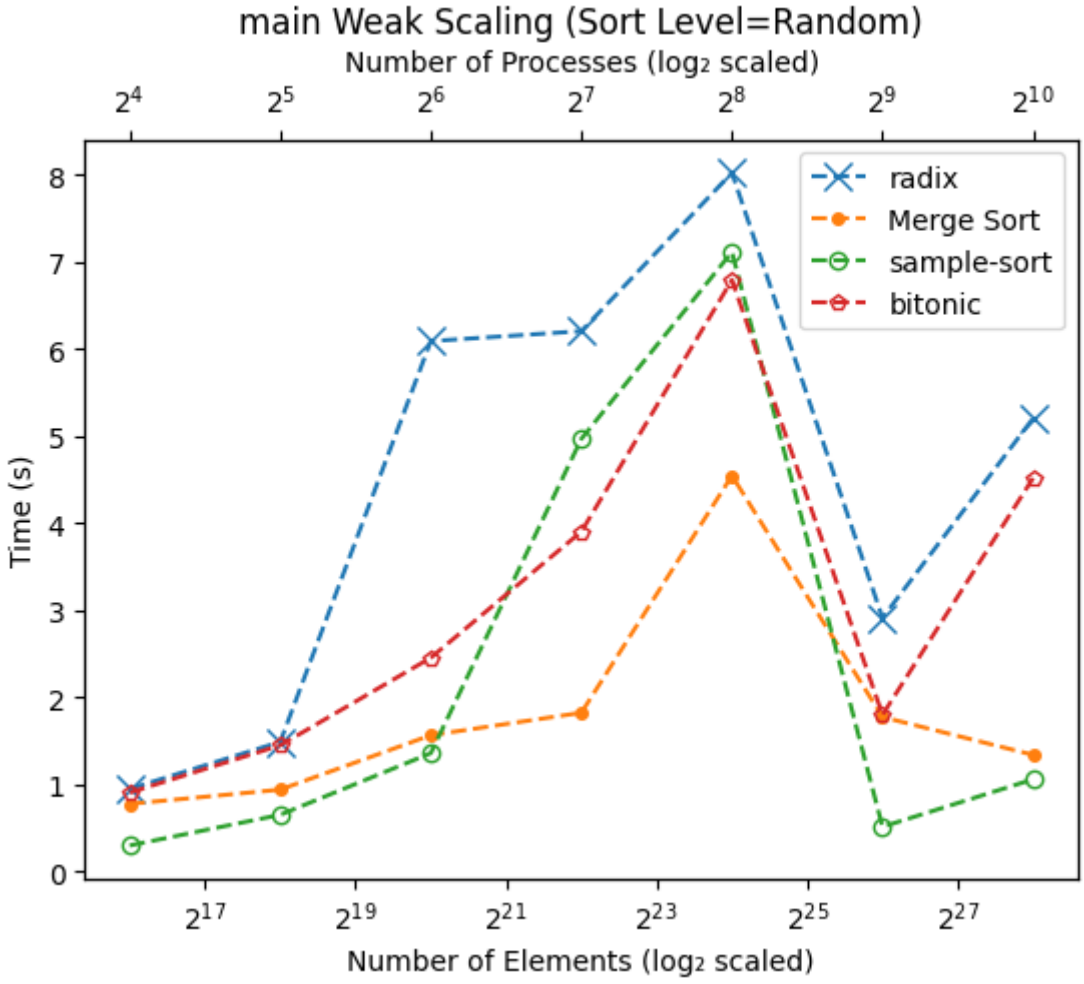

comm Speedup Factor Strong Scaling (Sort Level=Random)
radix
Merge Sort
sample-sort
bitonic
Speedup Factor
0
2
4
6
8
10
12
2^2
2^4
2^6
2^8
2^{10}
Number of Processes (log 2 scaled)



Weak Scaling Plots




comm Weak Scaling (Sort Level=Random)
Number of Processes (log₂ scaled)
2⁴ 2⁵ 2⁶ 2⁷ 2⁸ 2⁹ 2¹⁰
Time (s)
0 1 2 3 4 5 6 7
2¹⁷ 2¹⁹ 2²¹ 2²³ 2²⁵ 2²⁷
Number of Elements (log₂ scaled)
radix
Merge Sort
sample-sort
bitonic


main Weak Scaling (Sort Level=Random)
Number of Processes (log₂ scaled)
2⁴ 2⁵ 2⁶ 2⁷ 2⁸ 2⁹ 2¹⁰
Time (s)
0 1 2 3 4 5 6 7 8
2¹⁷ 2¹⁹ 2²¹ 2²³ 2²⁵ 2²⁷
Number of Elements (log₂ scaled)
radix
Merge Sort
sample-sort
bitonic

8. Final Report

Submit a zip named `TeamX.zip` where `X` is your team number. The zip should contain the following files:

- Algorithms: Directory of source code of your algorithms.
- Data: All `.cali` files used to generate the plots separated by algorithm/implementation.
- Jupyter notebook: The Jupyter notebook(s) used to generate the plots for the report.
- Report.md