# CSCE 435 Group project

## 0. Group number: 13

The team will communicate with discord channel

## 1. Group members:

1. Ravish Shardha
2. Lydia Harding
3. Brack Harmon
4. Jack Hoppe

## 2.Parallel sorting algorithms

### 2a. Brief project description (what algorithms will you be comparing and on what architectures)

All algorithms are going to be written and tested on grace cluster using MPI

- Bitonic Sort:
    - Parallel sorting algorithm that repeatedly sorts segments of a sequence of numbers into bitonic sequences. A bitonic sequence consists of a list of numbers which is first increasing, then decreasing. In the parallel version, for each i round, sorting is done with the partner that differs in the ith bit, alternating between collecting the lower half of all elements, and collecting the higher half of all elements, then sorting in order. Once all rounds have completed, the full array is sorted.
    - Bitonic sort only works on input arrays of size 2^n.
    - Bitonic sort will make the same number of comparisons for any input array, with a complexity of O(log^2n), where n is the number of elements to be sorted.
- Sample Sort:
    - Sample sort is a parallelized version of bucket sort.
    - Each processor takes a chunk of the data and sorts locally.
    - Then each processor takes s samples and those samples get combined into a buffer and sorted.
    - Global splitters or pivots are selected form the sorted samples and define endpoints for buckets.
    - each processor takes its data and filters it into each respective bucket.
    - The buckets are sorted and combined.
    - Finally the endpoints of each bucket are checked to verify the whole dataset is sorted.
    - Uses quicksort to sort partitioned buckets, so works well with random and in-order data.
- Merge Sort:
    - Divide-and-conquer sorting algorithm that splits the array into halves recursively until each sub-array contains a single element, then merges the sub-arrays to produce a sorted array.
    - Best and Worst case time complexity is O(nlogn) and space complexity is O(n).

    - For reverse and random sorted data, merge is a good choice.
    - For sorted and nearly sorted(1% random), merge sort doesn't consider the existing order so might not be the best choice.

- Radix Sort:

    - The Radix sort is a non-comparative integer sorting algorithm that iterates through each digit of the given elements starting from the least significant digit and progressing to the most significant digit. As the algorithm iterates the temporary results are placed in a "bucket" using an algorithm like the "count sort". These buckets are used to create the new ordering of the elements until all digits have been processed.
    - Through the MPI library and utilizing Grace a parallel implementation can be achieved by splitting input data into chunks then distributing them to workers. The workers will sort its chunk of data based on the current digit. After sorting, the bucket data from the count sort is redistributed across the processes. This continues until the most significant digit is reached resulting in sorted data.
    - The Radix sort requires integers or data that can be represented with integers and a fixed range of digits, i.e., (0-9)
    - Larger integers cause more passes and slows the algorithm
    - Uneven distributions of input data can lead to inefficiencies due to some buckets becoming disproportionately large.
    - Runtime: O(d*n) where d = number of digits and n = number of elements

## 2b. Pseudocode for each parallel algorithm

- For MPI programs, include MPI calls you will use to coordinate between processes
- **Merge Sort:**

```
INITIALIZE MPI(MPI_Init)
GET world_rank(MPI_Comm_rank)
GET world_size(MPI_Comm_size)

DIVIDE n by world_size to get chunk_size

CREATE sub_array of size chunk_size

SCATTER original_array to all processes (MPI_Scatter)
EACH PROCESS receives sub_array of size chunk_size

// Local Merge Sort
CALL mergeSort(sub_array, 0, chunk_size - 1)

// Gather sorted sub-arrays at root
GATHER sub_array at root into original_array (MPI_Gather)

IF world_rank == 0 THEN
    CALL mergeSort(original_array, 0, n - 1)  // Final merge at root
    PRINT sorted original_array

//Clean root and the rest of dynamically allocated arrays
```

```
  FREE sub_array and temp_arrays

  FINALIZE MPI
```

Bitonic Sort:

```
////////////////////
// MAIN
// MPI_Init,
// MPI_Comm_size(num_procs)
// MPI_Comm_rank(rank)

// Generate the input array on each processor (rand, in-order, reverse-order, or
perturbed)

// MPI_Barrier

// BITONIC SORT
// dimensions = log2(num_proc)
// For i = 0 to dimensions - 1:
    // For j = i down to 0:
        if (i + 1)st bit of rank == jth bit of rank then
            COMP EXCHANGE MIN (j)
        else
            COMP EXCHANGE MAX (j)
        // MPI_Barrier to ensure steps are in sync


// MPI_Barrier to ensure all sorting is complete

// VERIFY SORT
// Check if array is sorted locally
// Check if array end is less than start of neighbor process's array
// If both true for all processors, array is sorted.

// free array

// MPI Finalize
// RETURN
// END MAIN

////////////////////
// HELPER FUNCTIONS

////////////////////
// COMP EXCHANGE MIN (j)
// partner process = rank XOR (1 << j)
// MPI_Sendrecv array with partner, store in buffer_receive

// Concatenate array and buffer receive, store as temp buffer
// Sort temp
// Set array to be the lower half of temp
```

```
// free buffers
// RETURN
// END COMP EXCHANGE MIN
////////////////////

////////////////////
// COMP EXCHANGE MAX (j)
// partner process = rank XOR (1 << j)
// MPI_Sendrecv array with partner, store in buffer_receive

// Concatenate array and buffer receive, store as temp
// Sort temp
// Set array to be the higher half of temp

// free buffers
// RETURN
// END COMP EXCHANGE MAX
////////////////////
```

Sample Sort:

```
    // Starting MPI commands
    MPI_Init
    MPI_Comm_rank
    MPI_Comm_world
    etc.

    // Set Number of elements per processor
    data_size = total_size / num_processors

    // Fill local data with random integers
    for each element in data from rank * data_size to (rank + 1) * data_size:
    data[i] = random int from 1 to 999

    // Here is where we start timing the Samplesort algorithm

    // Step 1: Sort the local data
    sort local_data

    // Step 2: Select local data samples
    set samples_list empty
    samples_list[i] = sample an index from local_data

    // Step 3: Gather all the samples at rank 0 (rank 0 handles the samples)
    call MPI_Gather with sample data to fill gathered_samples


    // Step 4: Rank 0 sorts the samples and selects splitters
    if(rank == 0)
        sort gathered_samples
        for each sample
```

```
            splitters[i] = sample from gathered_samples


    // Step 5: Broadcast the splitters to all processes
    call MPI_Bcast with splitters and num_samples

    // Step 6: Partition the local data based on the splitters

    create send_counts, send_offsets, and partitioned_data arrays
    populate partitioned_data with info from data based on offsets and send counts

    // Step 7: Send partitioned data to respective processor buckets
    use MPI_Alltoall with send_counts and recv_counts
    create recv_data array and populate with received information using
MPI_Alltoallv

    // Step 8: Sort the received data locally
    sort recv_data

    // Here is where we end timing the Samplesort algorithm

    // print sorted data (optional)

    // Ending MPI commands (MPI_Finalize)
```

Radix Sort:

```
1. Initialize MPI, get rank, and size
INITIALIZE MPI(MPI_Init)
GET world_rank(MPI_Comm_rank)
GET world_size(MPI_Comm_size)

2. Generate different types of inputs listed in 2c.
    Create input arrays of different sizes etc
    Define length of data

3. Distribute data to processes with MPI_Scatter
    rank = 0:
        divide n by world size to calculate chunck size
        create arrays with chunk size and then fill
        send the chunks out to the processes
        MPI_Scatter(chunks)

4. Radix sort for worker
    -iterate through each digit:
        -each process performs local count sort:

        -initialize count array representing numbers 0-9
         array count[10] = {0};

        -Count occurence of each digit from local chunk
         Extract current digit with modulo (%)
```

```
           count[current digit]++;

          -Gather count data from each process and combine
           MPI_Allreduce(total_count)

          -Caluclate Cumulative count
          cumulative[i] = cumulative[i-1] + total_count[i]

          -Redistribute the elements based on the combined count
          Then place into processes depending on that calculation
          MPI_Alltoall

5. Use MPI_Gather to collect sorted data
    After processing all digits gather data
    MPI_Gather(sorted data)

6. Finalize MPI
MPI_Finalize()
```

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

- Input sizes, Input types:
  - The input will consist of multiple arrays of numerical data, with sizes increasing by powers of 2. These arrays will be tested on varying numbers of processors, which will also increase by powers of 2. Furthermore, there would be four main input types: sorted, nearly sorted, random, and reverse sorted.
  - These would be the inputs:
    - For input_size's:
      - 2^16, 2^18, 2^20, 2^22, 2^24, 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
- Strong scaling (same problem size, increasing number of processors/nodes)
  - We will keep the problem size fixed (e.g., a 2^18-sized array) and increase the number of processors/nodes. For each configuration, we will graph the computation time. This analysis will be done for all four input types: sorted, nearly sorted, random, and reverse sorted. Overall, there would be a total of 7 plots, one for each input size with 4 lines per plot representing each input type.
- Weak scaling (increase problem size, increase number of processors)
  - We will increase the problem size while also increasing the number of processors (e.g., 2, 4, 8, etc.). The computation time for each array size will be graphed across varying numbers of processors. This analysis will also be conducted for all four input types: sorted, nearly sorted, random, and reverse sorted. There would be four plots for each input type for weak scaling.

## 3a. Caliper instrumentation

Please use the caliper build `/scratch/group/csce435-f24/Caliper/caliper/share/cmake/caliper` (same as lab2 build.sh) to collect caliper files for each experiment you run.

Your Caliper annotations should result in the following calltree (use `Thicket.tree()` to see the calltree):

```
main
|_ data_init_X      # X = runtime OR io
|_ comm
|    |_ comm_small
|    |_ comm_large
|_ comp
|    |_ comp_small
|    |_ comp_large
|_ correctness_check
```

Required region annotations:

- `main` - top-level main function.
  - `data_init_X` - the function where input data is generated or read in from file. Use *data_init_runtime* if you are generating the data during the program, and *data_init_io* if you are reading the data from a file.
  - `correctness_check` - function for checking the correctness of the algorithm output (e.g., checking if the resulting data is sorted).
  - `comm` - All communication-related functions in your algorithm should be nested under the `comm` region.
    - Inside the `comm` region, you should create regions to indicate how much data you are communicating (i.e., `comm_small` if you are sending or broadcasting a few values, `comm_large` if you are sending all of your local values).
    - Notice that auxillary functions like MPI_init are not under here.
  - `comp` - All computation functions within your algorithm should be nested under the `comp` region.
    - Inside the `comp` region, you should create regions to indicate how much data you are computing on (i.e., `comp_small` if you are sorting a few values like the splitters, `comp_large` if you are sorting values in the array).
    - Notice that auxillary functions like data_init are not under here.
  - `MPI_X` - You will also see MPI regions in the calltree if using the appropriate MPI profiling configuration (see **Builds/**). Examples shown below.

All functions will be called from `main` and most will be grouped under either `comm` or `comp` regions, representing communication and computation, respectively. You should be timing as many significant functions in your code as possible. **Do not** time print statements or other insignificant operations that may skew the performance measurements.

### **Nesting Code Regions Example** - all computation code regions should be nested in the "comp" parent code region as following:

```
CALI_MARK_BEGIN("comp");
CALI_MARK_BEGIN("comp_small");
sort_pivots(pivot_arr);
CALI_MARK_END("comp_small");
CALI_MARK_END("comp");
```

```
# Other non-computation code
...

CALI_MARK_BEGIN("comp");
CALI_MARK_BEGIN("comp_large");
sort_values(arr);
CALI_MARK_END("comp_large");
CALI_MARK_END("comp");
```

**Calltree Example**:

```
# MPI Mergesort
4.695 main
├─ 0.001 MPI_Comm_dup
├─ 0.000 MPI_Finalize
├─ 0.000 MPI_Finalized
├─ 0.000 MPI_Init
├─ 0.000 MPI_Initialized
├─ 2.599 comm
│  ├─ 2.572 MPI_Barrier
│  └─ 0.027 comm_large
│     ├─ 0.011 MPI_Gather
│     └─ 0.016 MPI_Scatter
├─ 0.910 comp
│  └─ 0.909 comp_large
├─ 0.201 data_init_runtime
└─ 0.440 correctness_check
```

**I) Calltree For Merge Sort:**

```
1.552 main
├─ 0.000 MPI_Init
├─ 0.006 data_init_runtime
├─ 0.899 comm
│  ├─ 0.432 comm_large
│  │  ├─ 0.431 MPI_Scatter
│  │  └─ 0.001 MPI_Gather
│  └─ 0.467 MPI_Barrier
├─ 0.011 comp
│  ├─ 0.006 comp_small
│  └─ 0.036 comp_large
├─ 0.000 MPI_Finalize
├─ 0.001 correctness_check
├─ 0.000 MPI_Initialized
├─ 0.000 MPI_Finalized
└─ 0.004 MPI_Comm_dup
```

**II) Calltree For Sample Sort:**

**III) Calltree for Bitonic Sort:**

**VI) Calltree for Radix Sort:**

## 3b. 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.

We have included the metadata code mentioned in 3b in our algorithms. The values that we are getting in the metadata are the following:

- Launch date of the job
- Libraries used in our algorithm
- Name of the cluster

- Name of the algorithm you are using
- The programming model used for communication which in our case is MPI
- The datatype of input elements, which in our case is int
- The sizeof(datatype) of input(int) elements in bytes
- The number of elements in the input dataset/n - 2^16, 2^18, 2^20, 2^22, 2^24, 2^26, 2^28
- The input type used for sorting("Sorted", "ReverseSorted", "Random", "1_perc_perturbed").
- The number of processors used(MPI ranks) - 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024
- The scalability of your algorithm. Either if we are doing strong or weak
- Our group number, which is 13
- The place where we got the source code of your algorithm

# 4. Performance evaluation

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

## 4a. Vary the following parameters

For input_size's:

- 2^16, 2^18, 2^20, 2^22, 2^24, 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.

## 4b. 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).

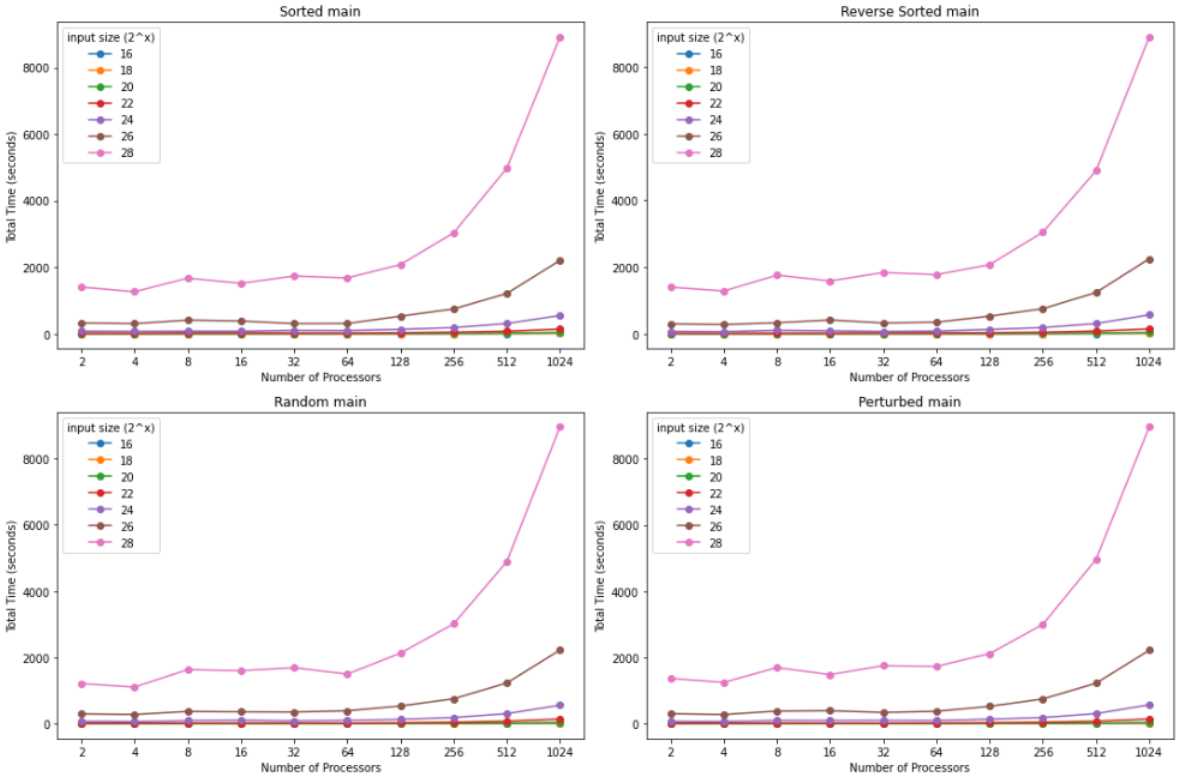
## 4c. You should measure the following performance metrics

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

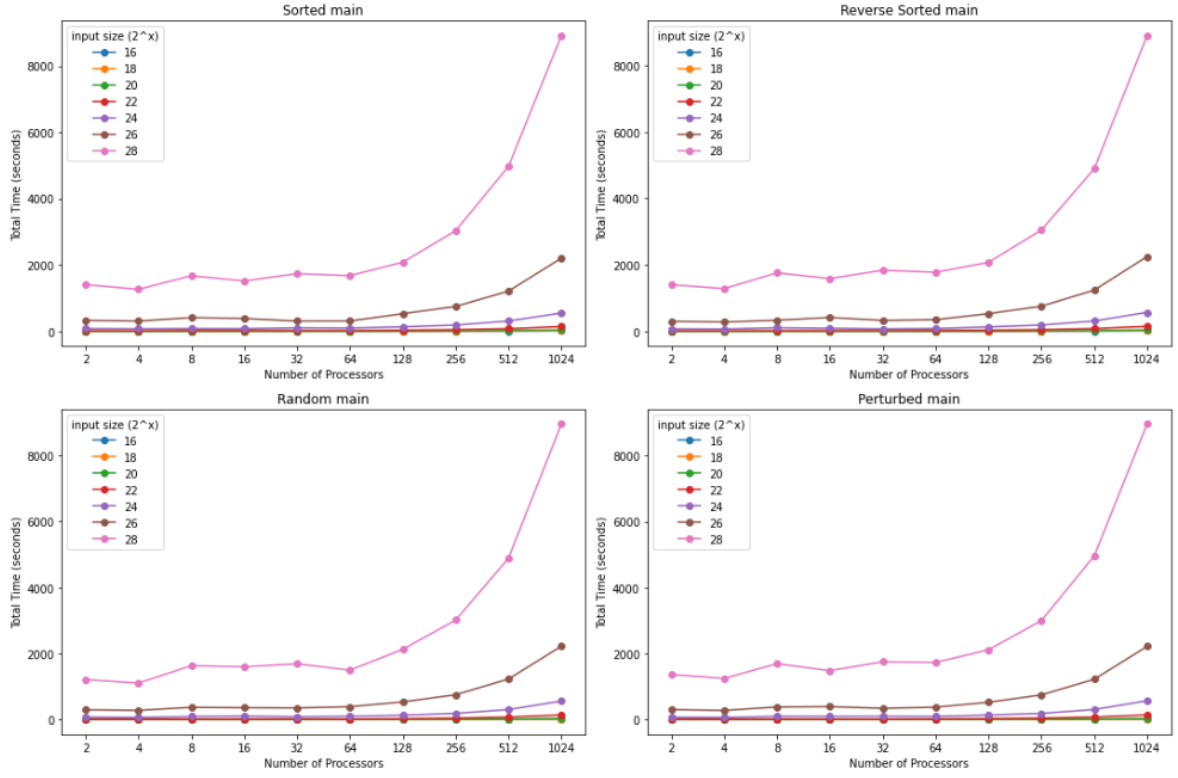
- Total time
- Variance time/rank

## I) Merge Sort

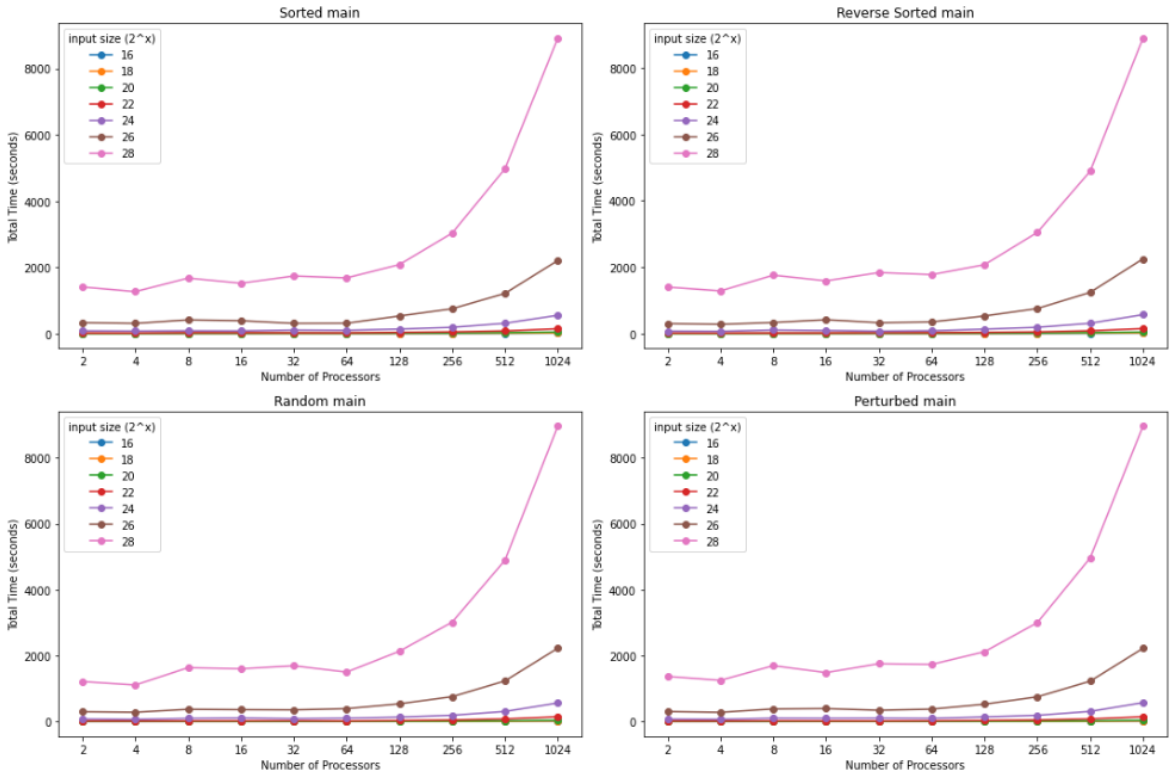
- **main:**
  - Avg time/Rank:



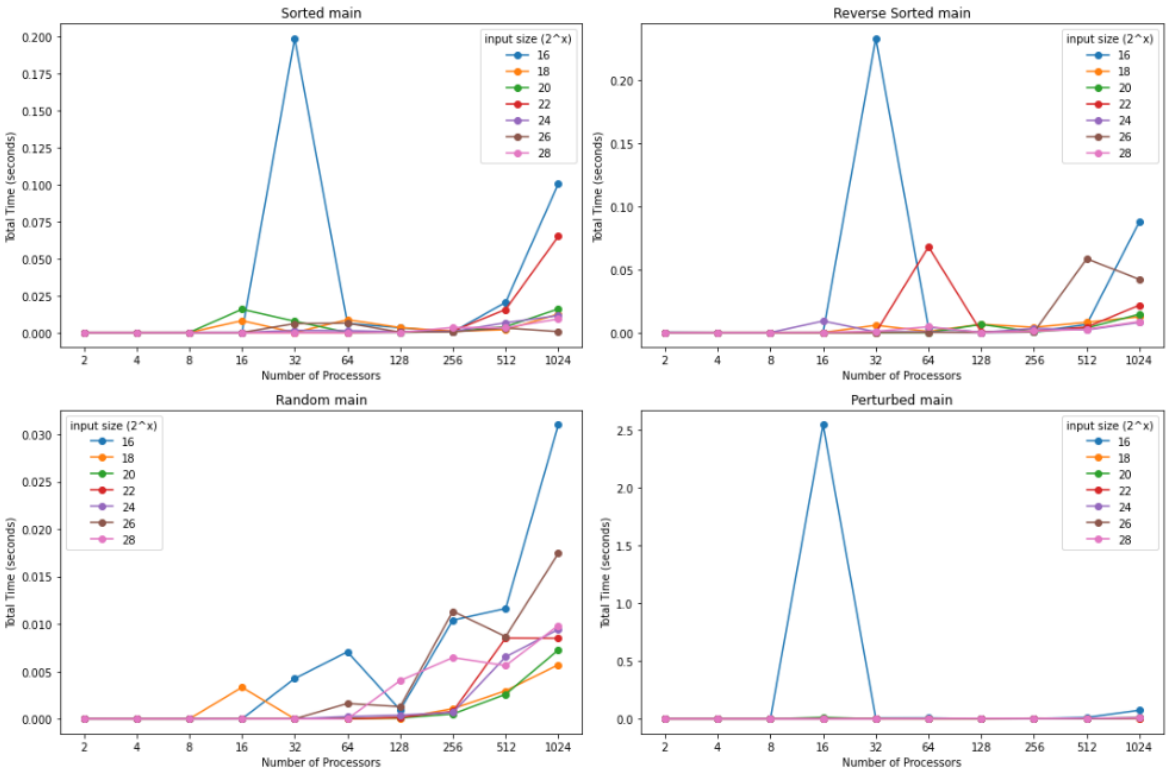

  - Min time/Rank:



- Max time/Rank:



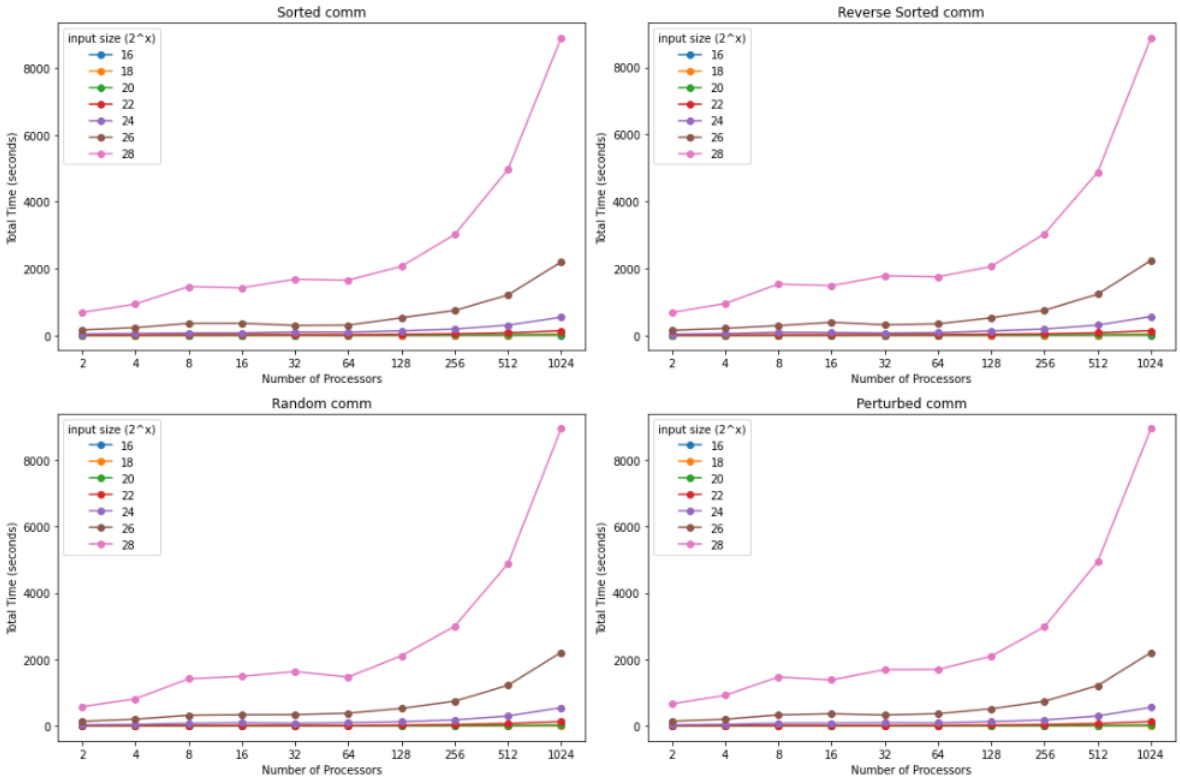

- Variance time/Rank:



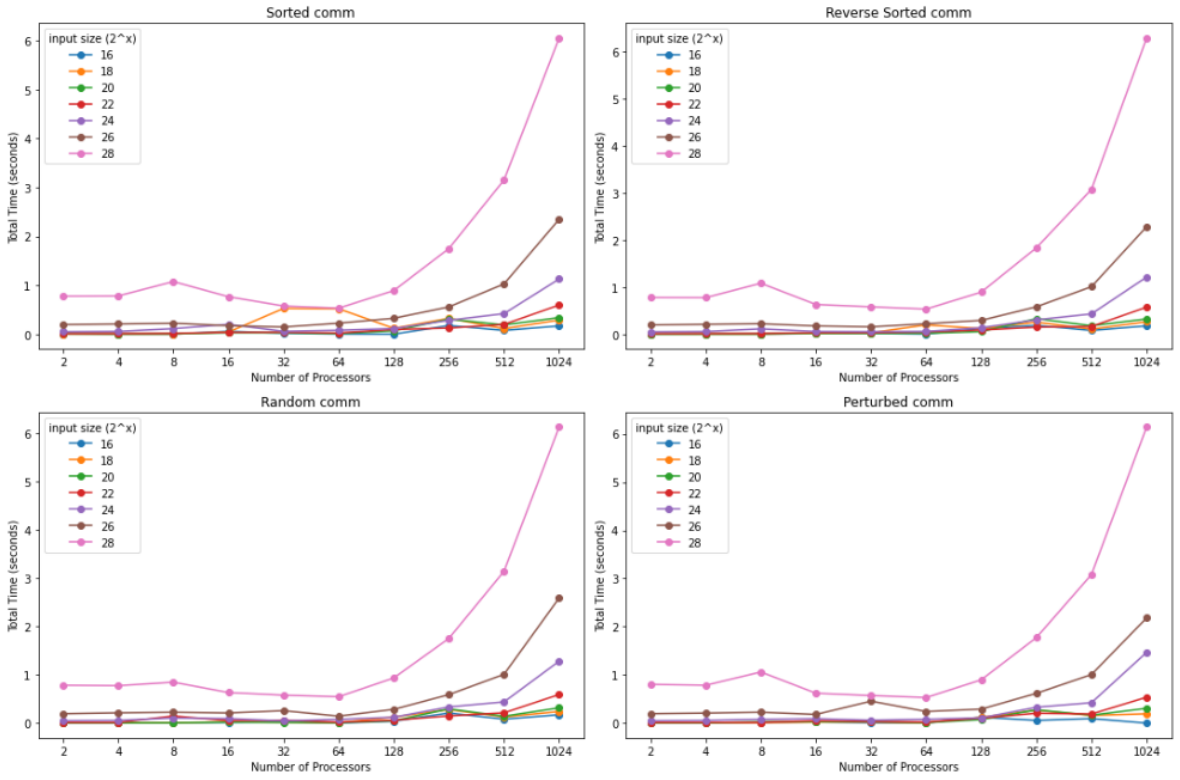

For all the input types, as the array size increases, we can see the avg, max, and min time for the main also increases. This is because, with more data, the algorithm needs to make more computations and communication. The stability of up to 64 processors suggests good parallelization. The sharp increase beyond 128 processors is likely because there's a point where adding more processors becomes counterproductive. The problem size becomes too small for each processor making the overhead of parallelization outweigh its benefits. All input types have similar data points/trends due to the way merge sort works. Regardless of what

input type is used, merge sort still divides and conquers the data and does almost the same number of computations - with random and 1% pert doing a bit more computation. This makes the complexity O(nlogn)

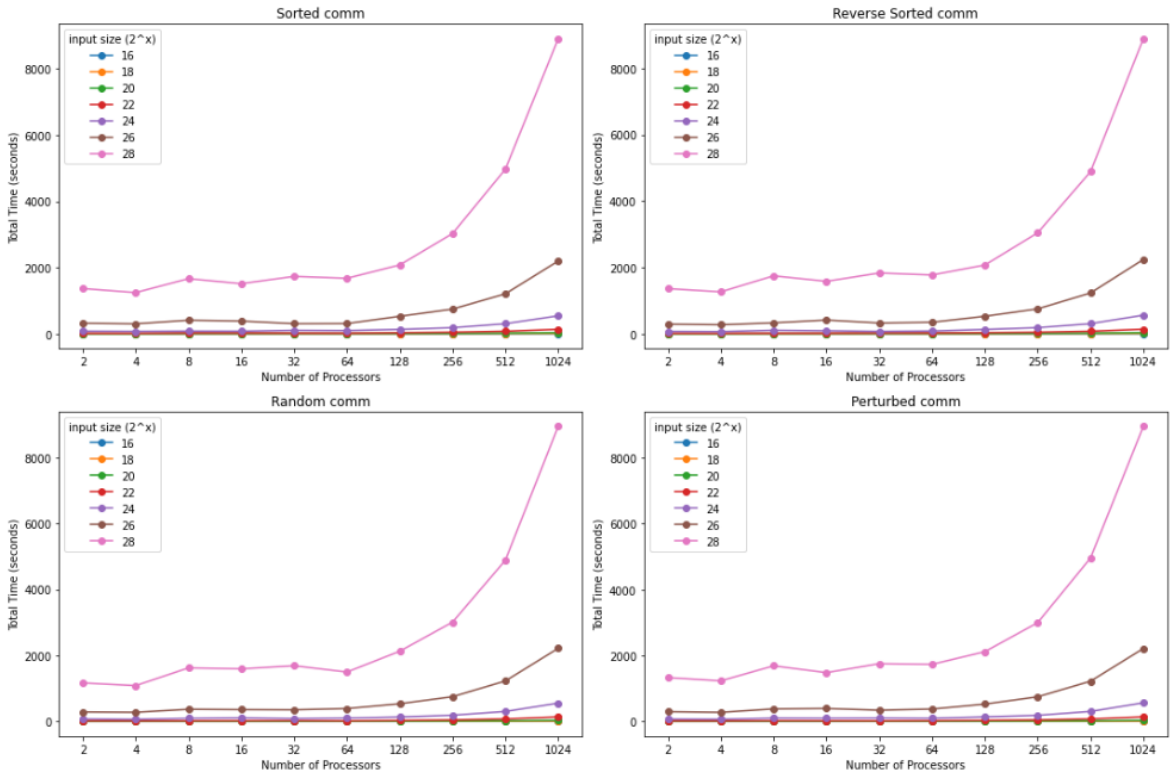
- **comm:**
  - Avg time/Rank:



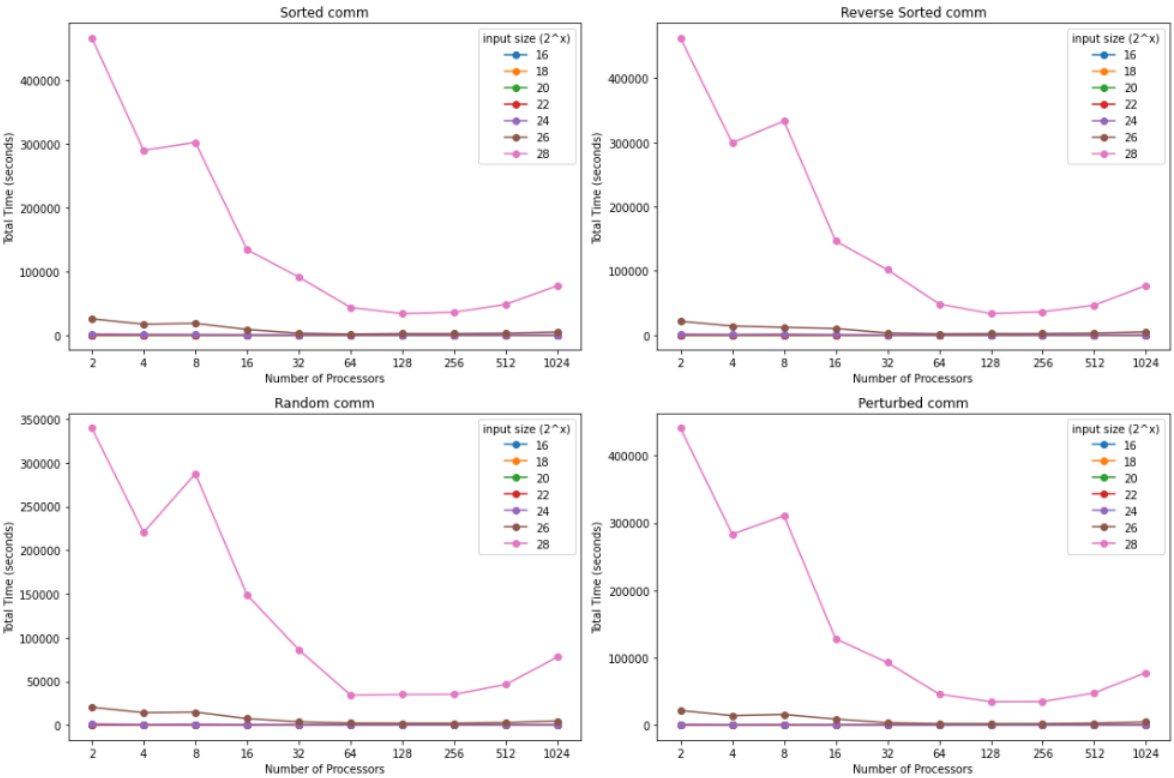

  - Min time/Rank:



- Max time/Rank:



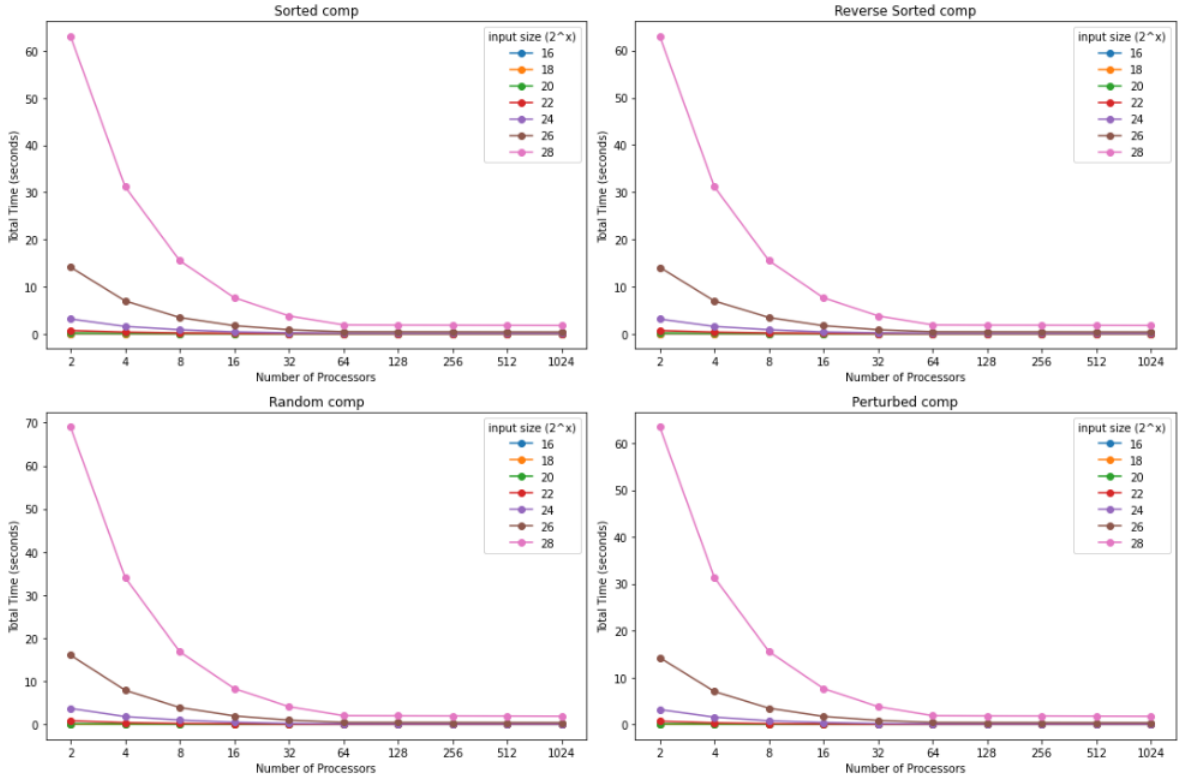

- Variance time/Rank:



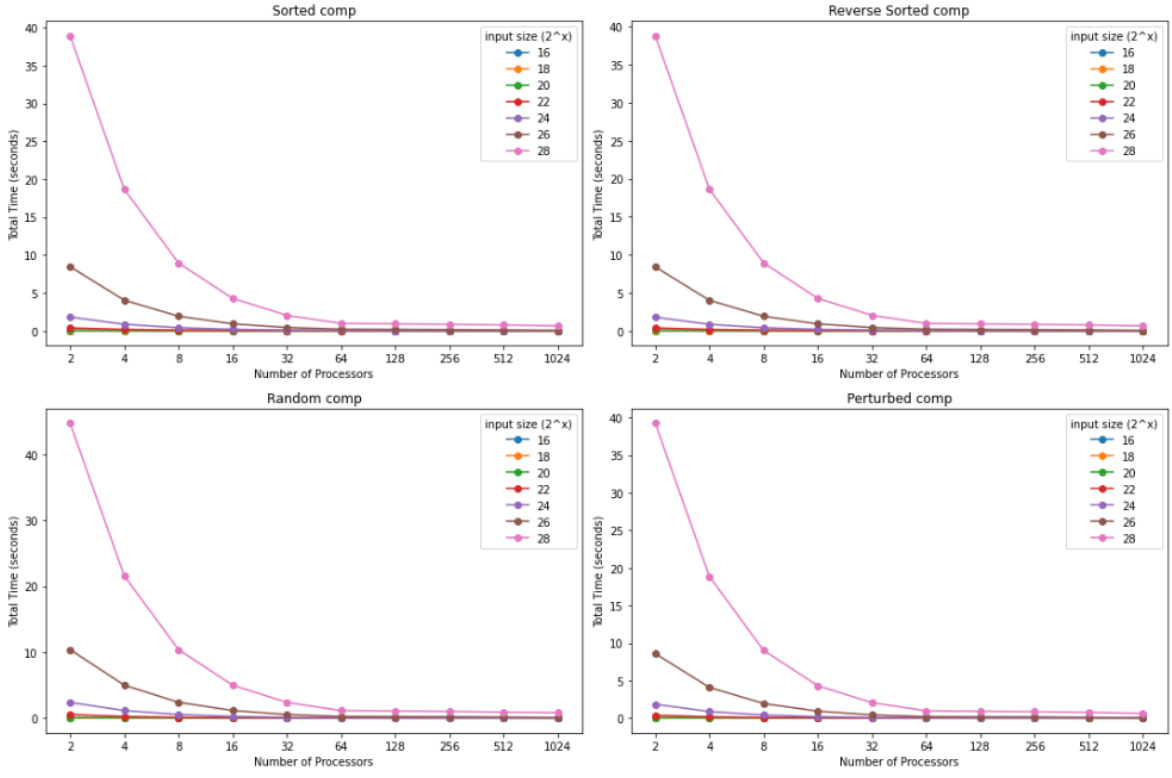

For all communication patterns, the max, min, and avg increase as the input size increases. This is expected as the algorithm has more work to do with larger datasets. So more data to scatter, gather, and wait for due to MPI barrier. This consistency makes sense because the communication costs are dependent on data size, not data arrangement. The code shows the same communication pattern (Scatter/Gather) regardless of input type. There's a notable uptick in communication time as the number of processors increases beyond 128 which can be caused by Network congestion with more communicating nodes, the overhead of coordinating

larger numbers of processes or MPI_Scatter/MPI_Gather operations becoming more expensive with more participants.

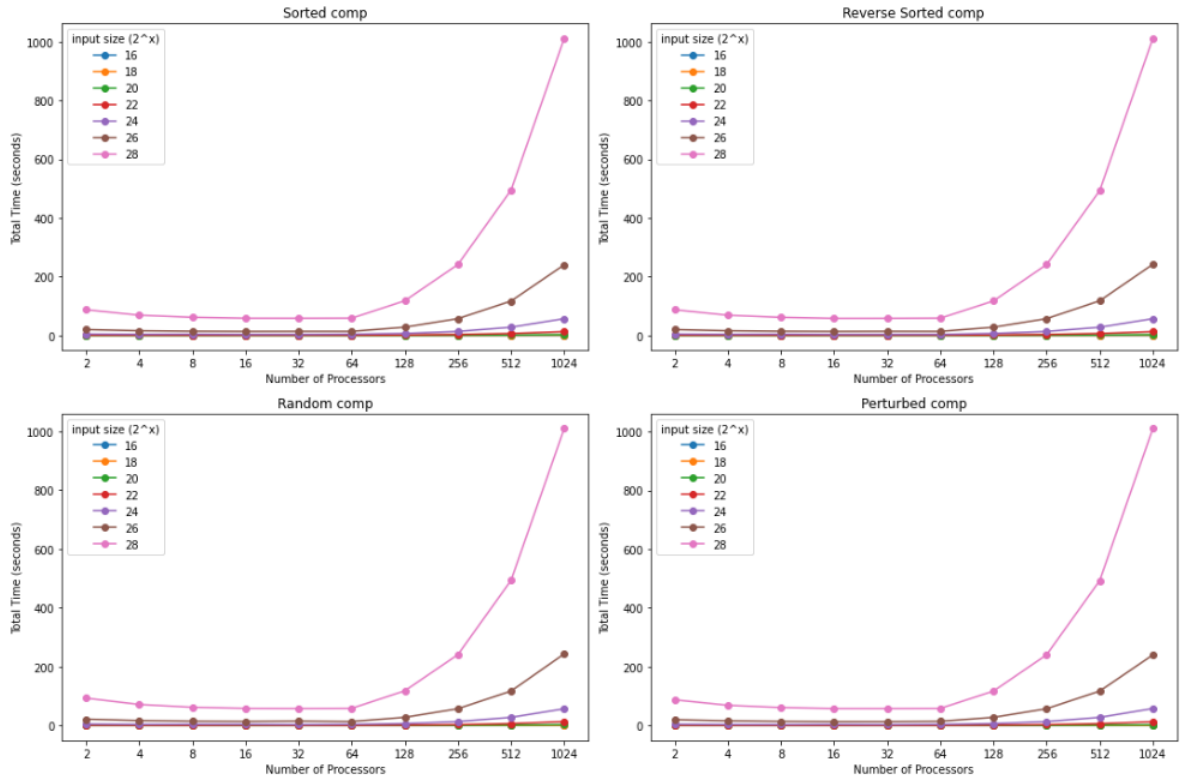
- **comp:**
  - Avg time/Rank:



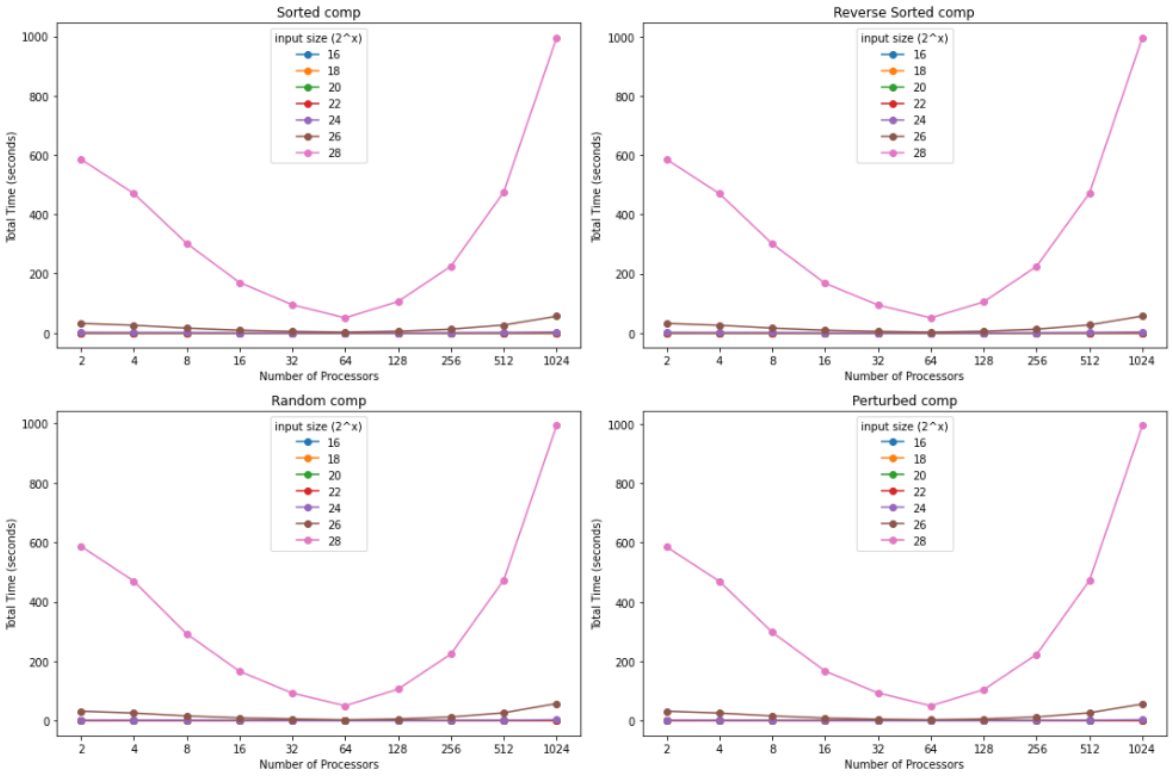

  - Min time/Rank:



- Max time/Rank:



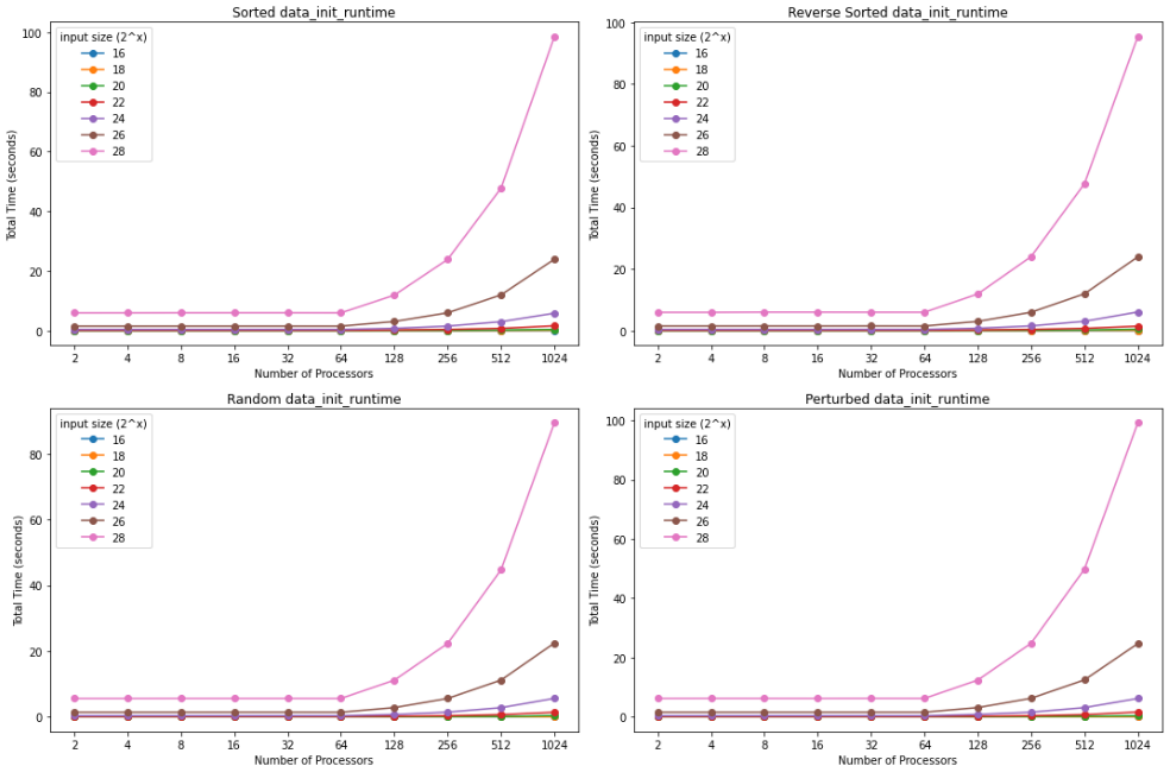

- Variance time/Rank:



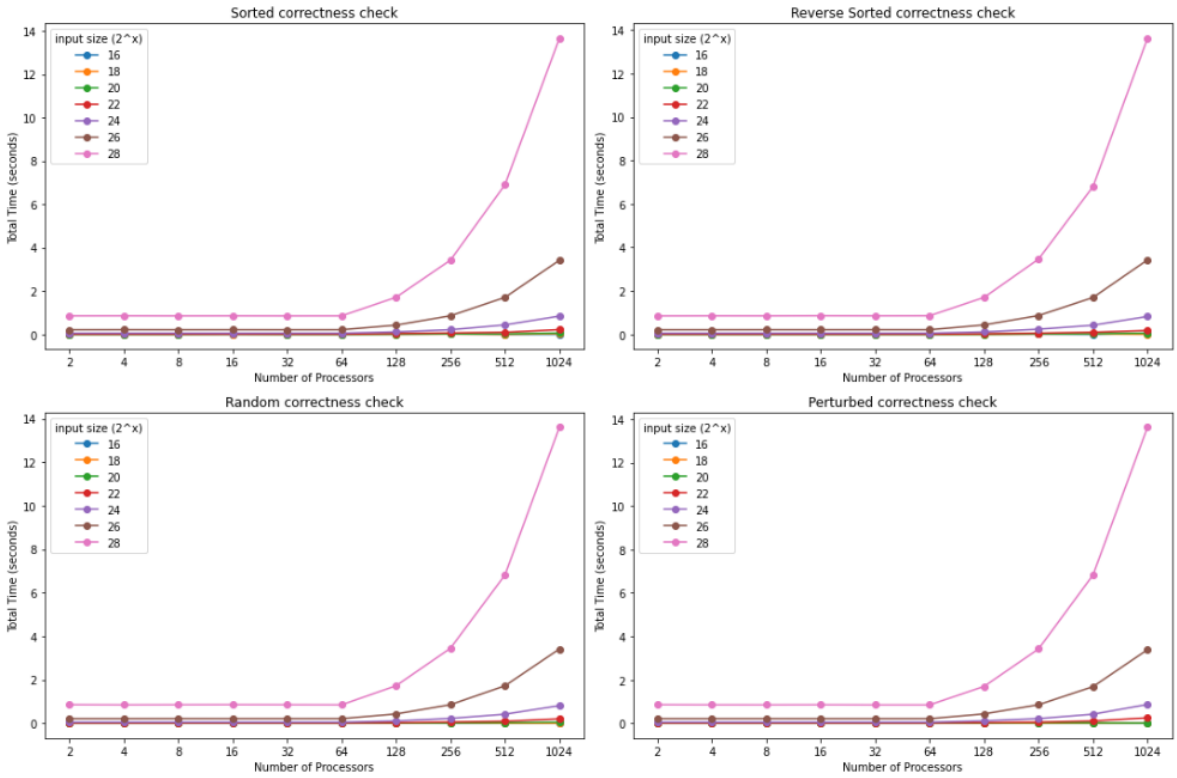

All four input types show a clear trend of decreasing avg computation time as the number of processors increases This is because the mergeSort function is being parallelized, with each processor handling a smaller subset (size = n/world_size) of the data. That is why 2^28 with 2 processors shows the highest time because that data is divided on only 2 processors. As the number of processors increases, the data is being uniformly divided into more processors so less time spent on computation for the smaller chunk. The computation time

curves flatten out after around 64-128 processors, this suggests that each processor's local mergeSort operation becomes too small to benefit from further division.

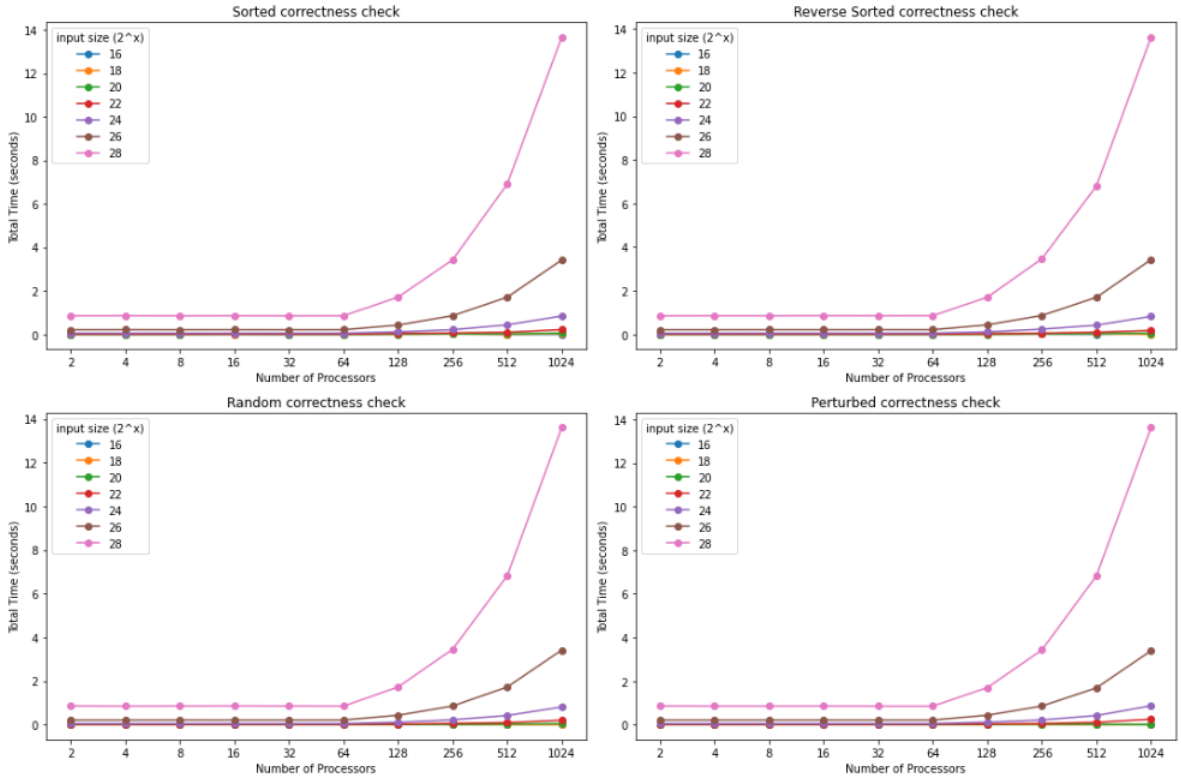
- **Data generation:**
  - Avg time/Rank:



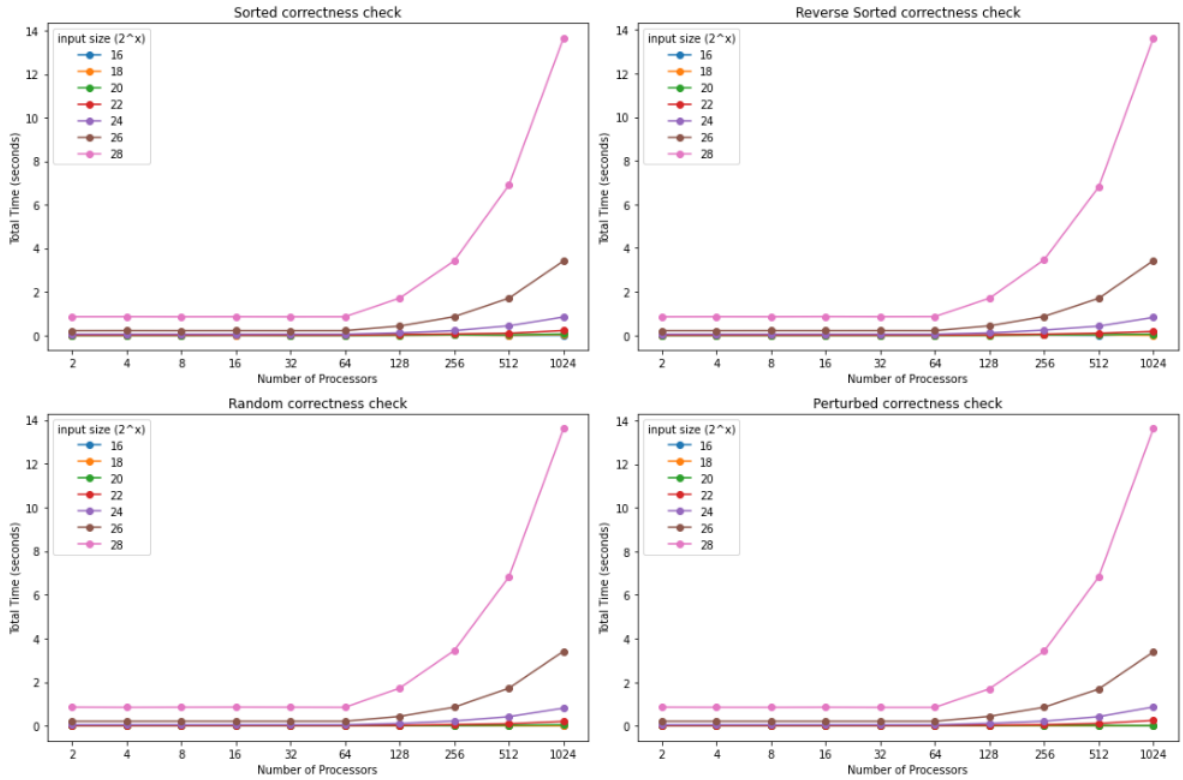

- **Correctness check:**
  - Avg time/Rank:



- Min time/Rank:




- Max time/Rank:



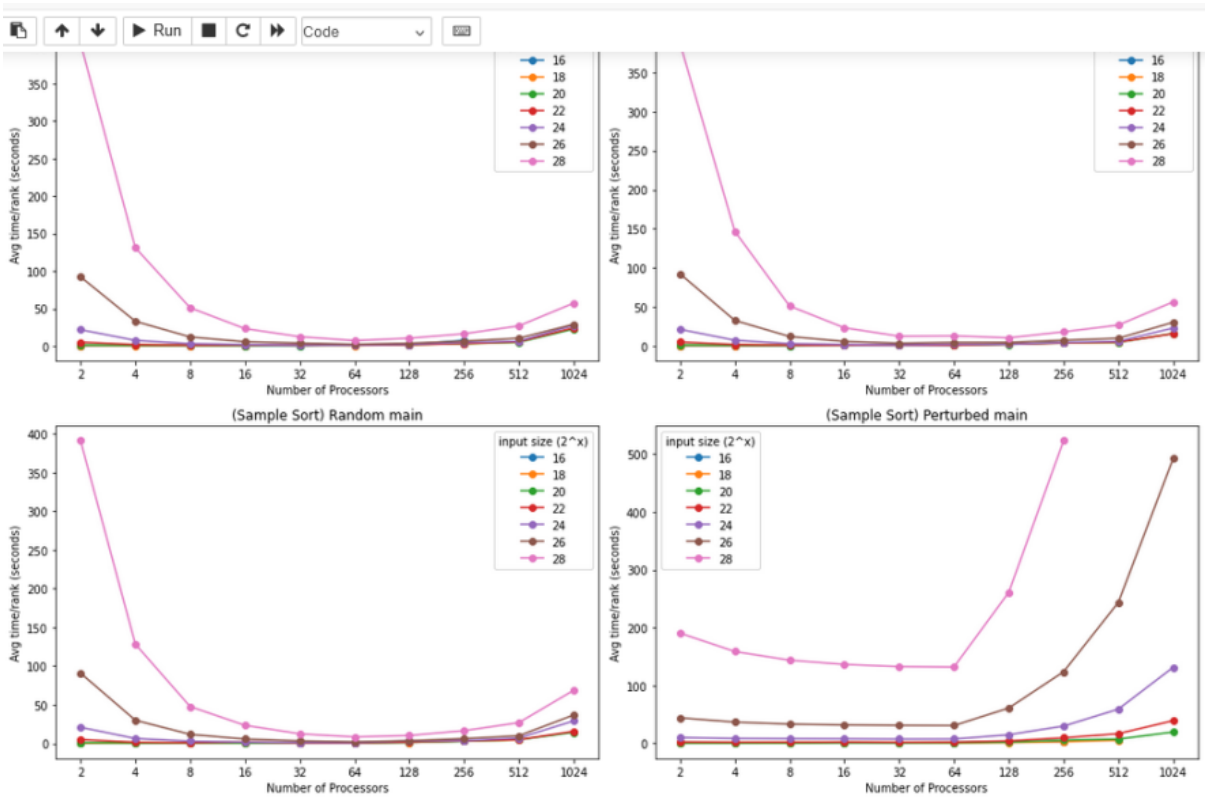

Data initialization and correctness check time increases significantly with input size for all input types because there is more data to generate, traverse, and check. The root process (rank 0) handles all data generation (generate_array function) and traversing and checking if the array is sorted. This means that larger processor counts create more overhead in managing the distributed environment.
Both metrics show:

- Exponential growth with input size

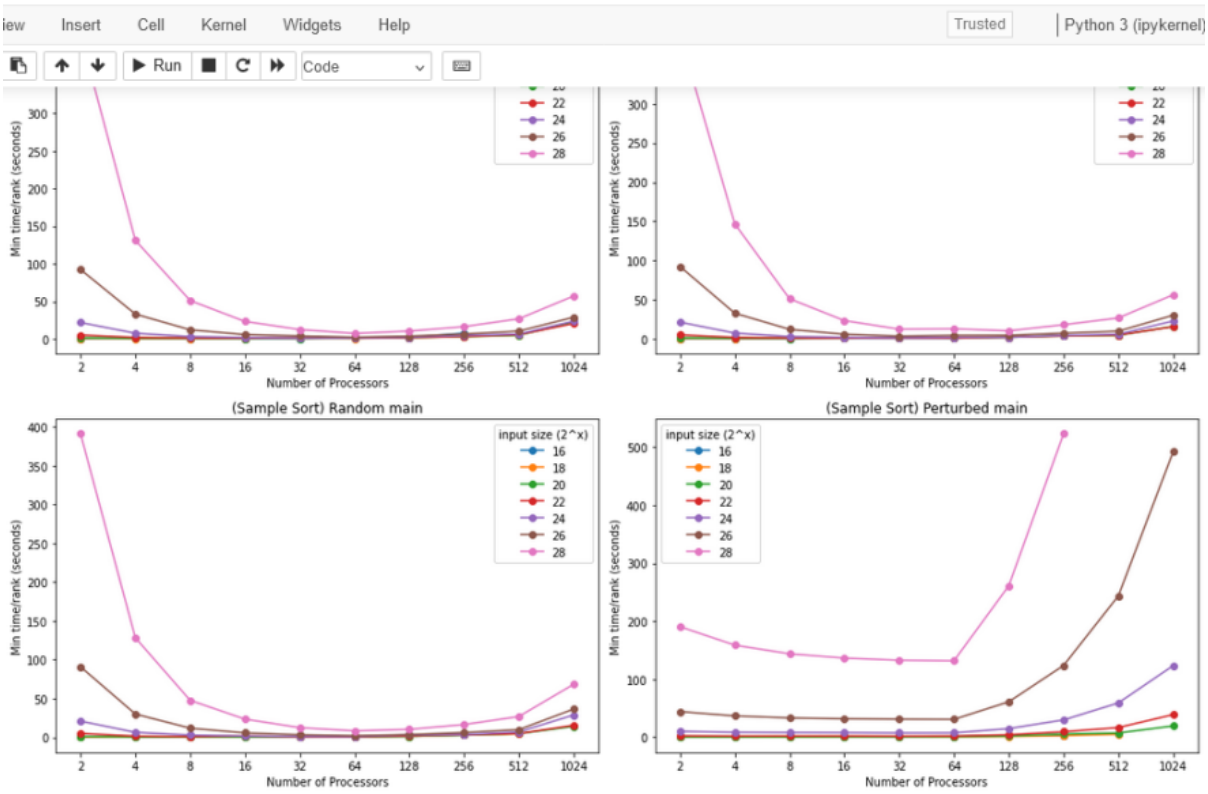
- Similar behavior across different input types
- Performance degradation at high processor counts (>256)
  This similarity makes sense because both operations are sequential and performed only by the root process. This sequential process is the main bottleneck for the calculation time.

## II) Sample Sort

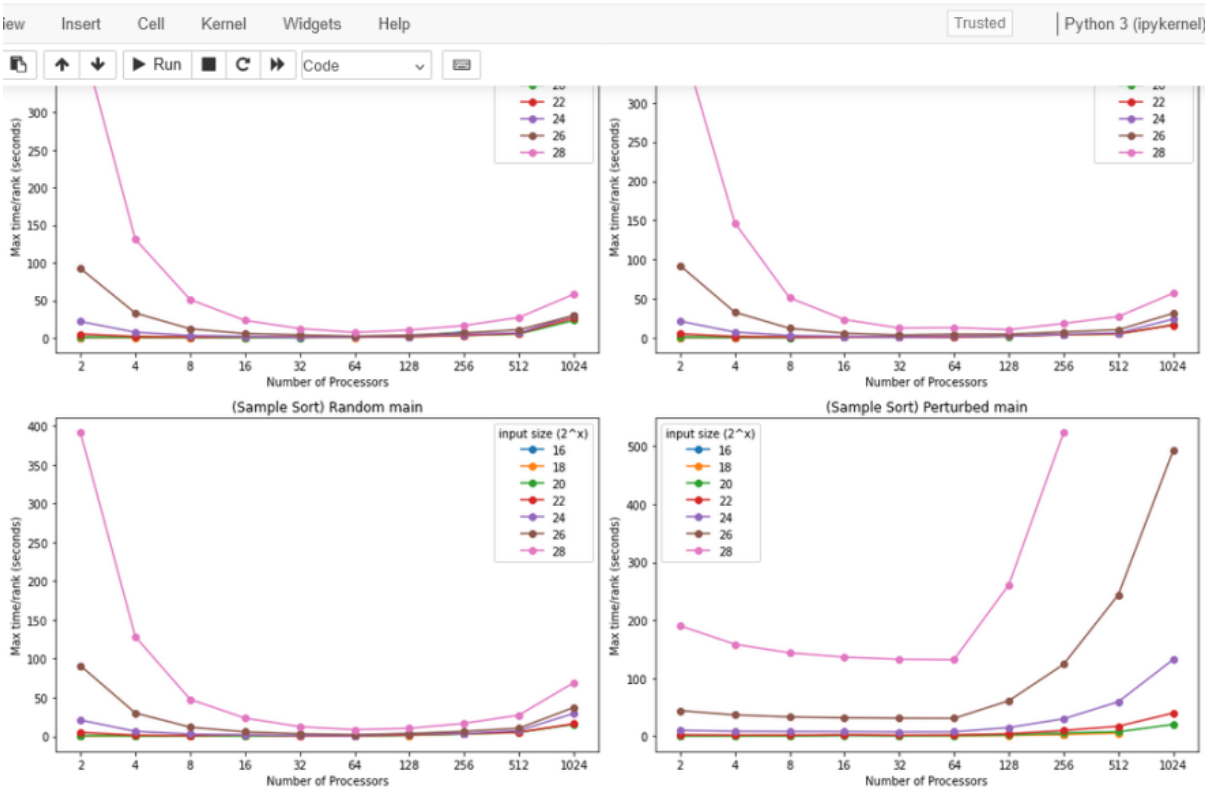
- **main:**
  - Avg time/Rank:



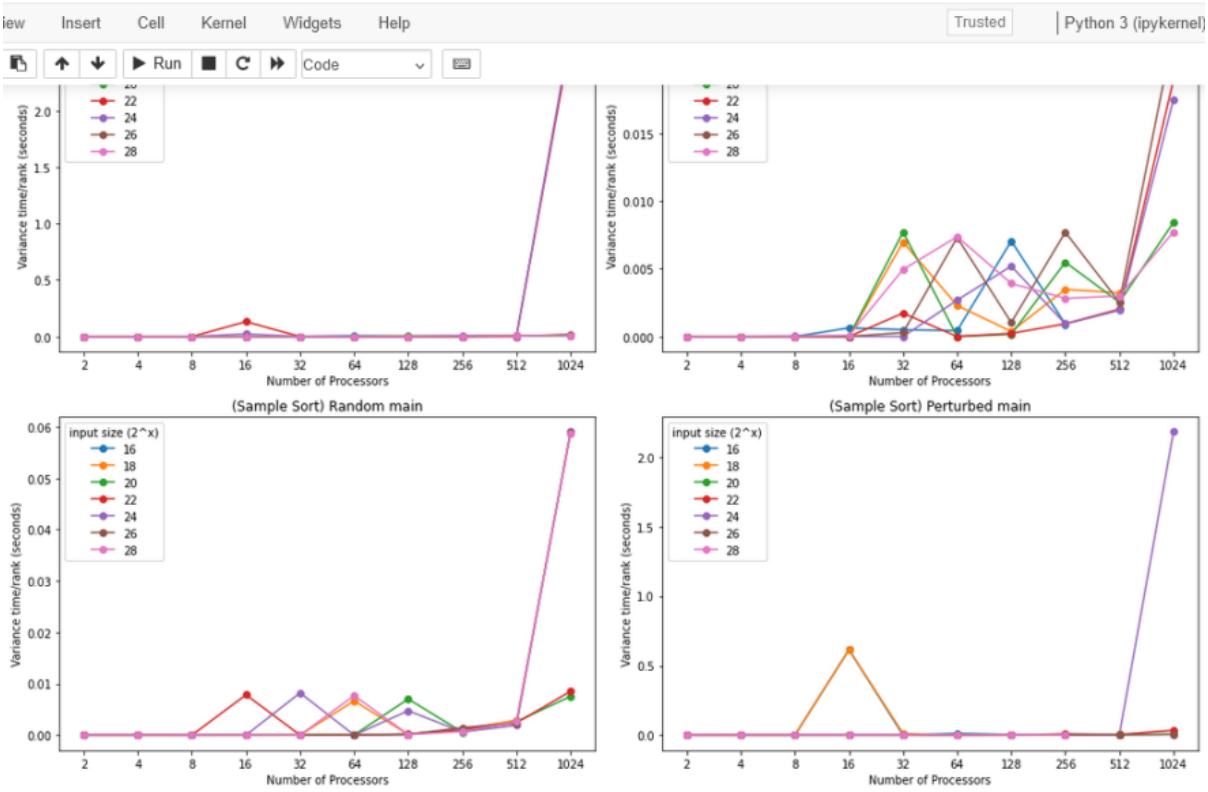
- Min time/Rank:



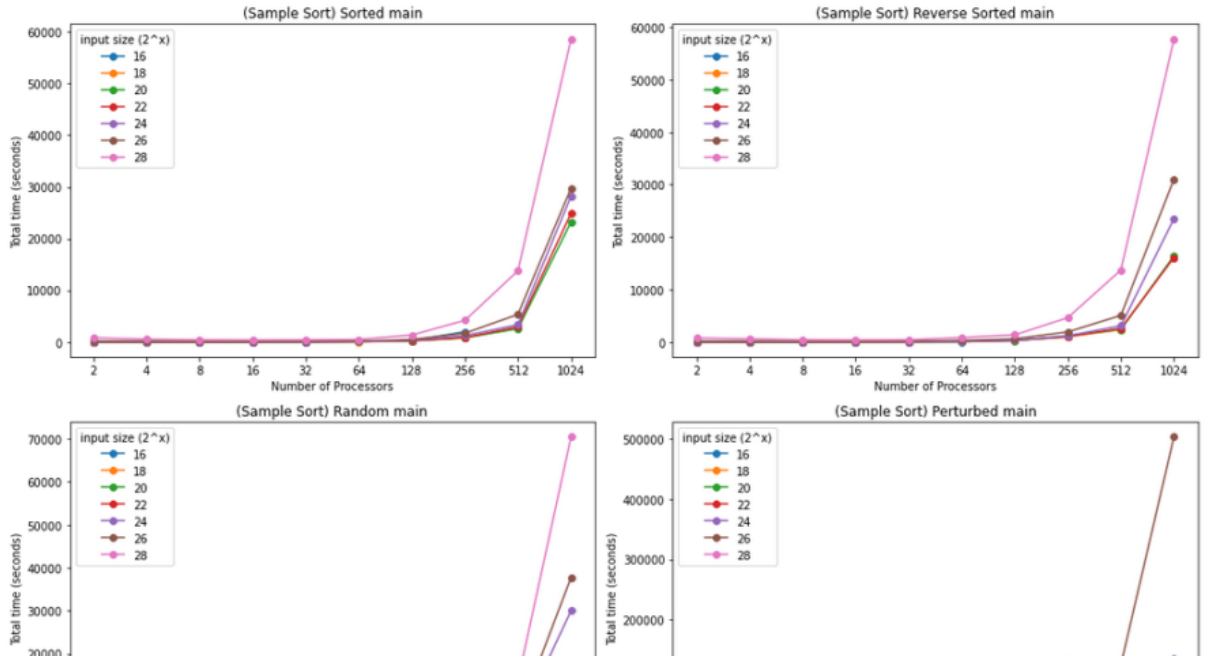

- Max time/Rank:



- Variance time/Rank:



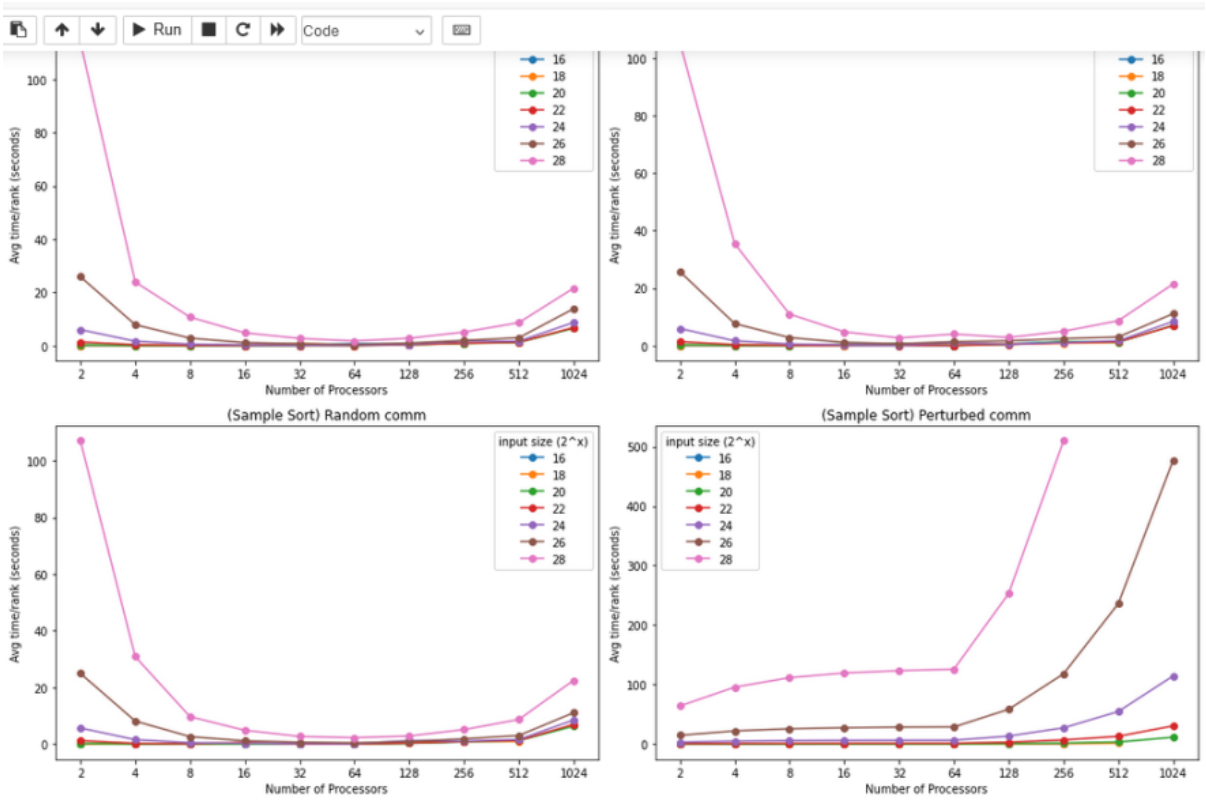

- Total time:



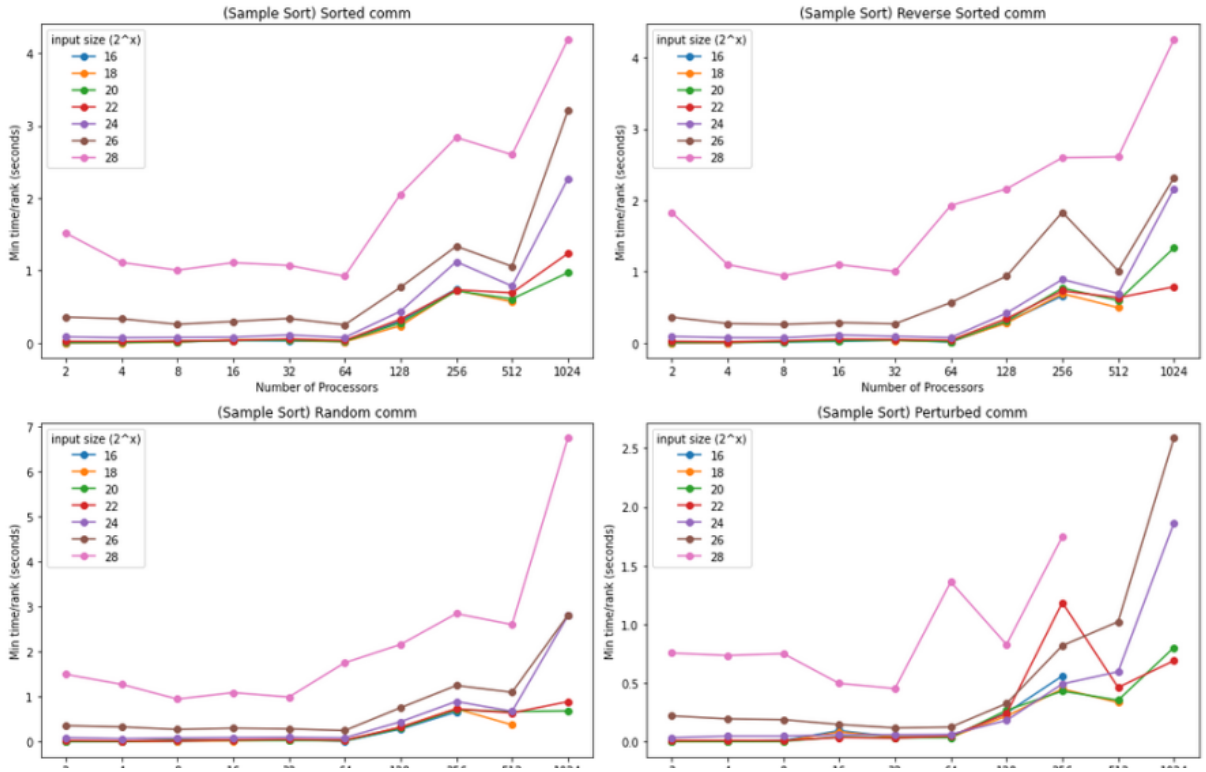

Overall, for main as the input size increases, the longer sample sort takes to sort the array across all input types as shown by the higher input size dots are higher than the lower input size dots on average time per processor. before 64 processors, runtime descrases as the number of processors increase likely due to the data getting separated into more buckets and sorted quicker locally. After 64 processors, runtime increases slightly as the number of processors increase likely due to increases in communication overhead (MPI_Alltoall, MPI_Gather).

- **comm:**

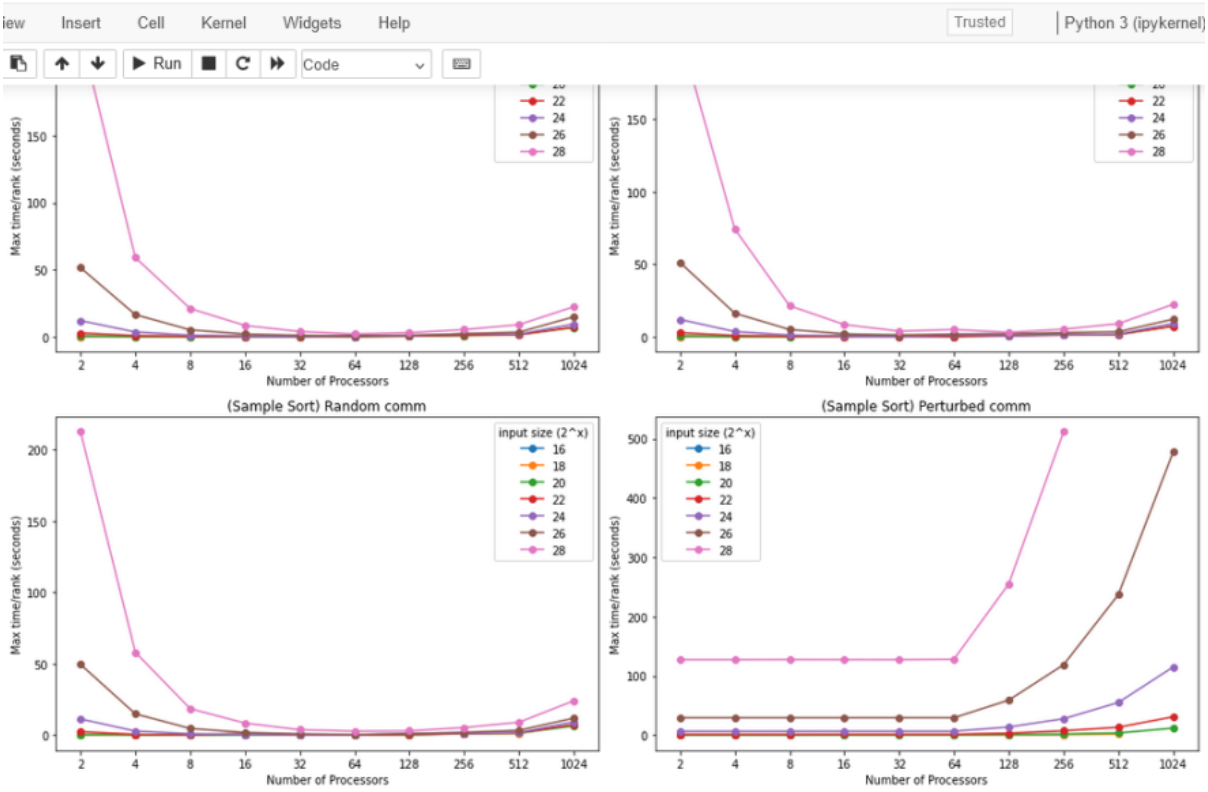
- Avg time/Rank:



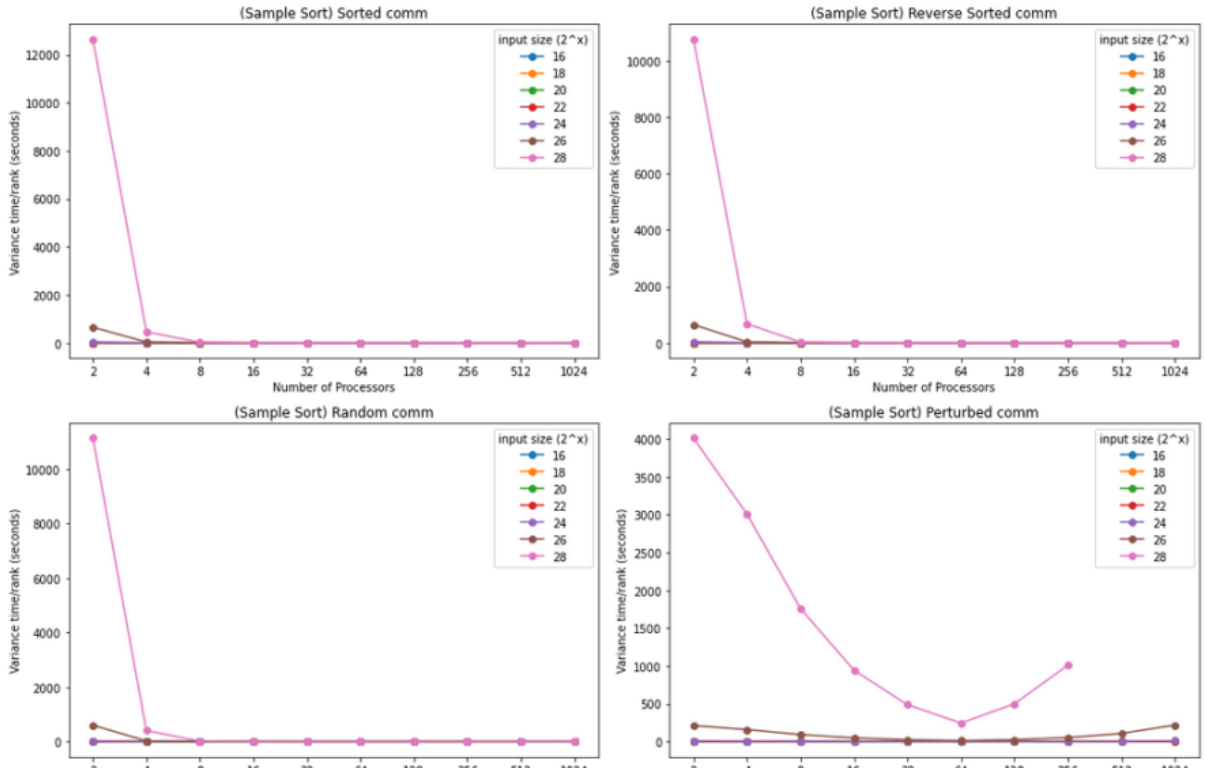

- Min time/Rank:



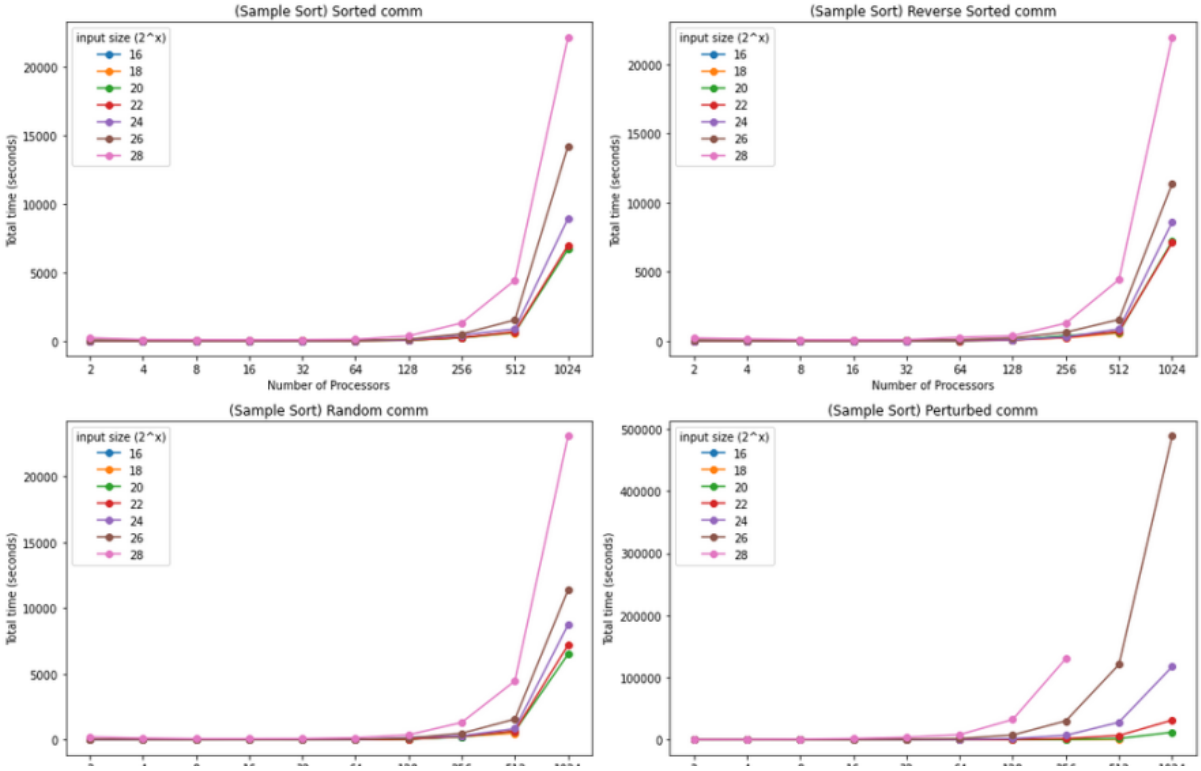
- Max time/Rank:




- Variance time/Rank:



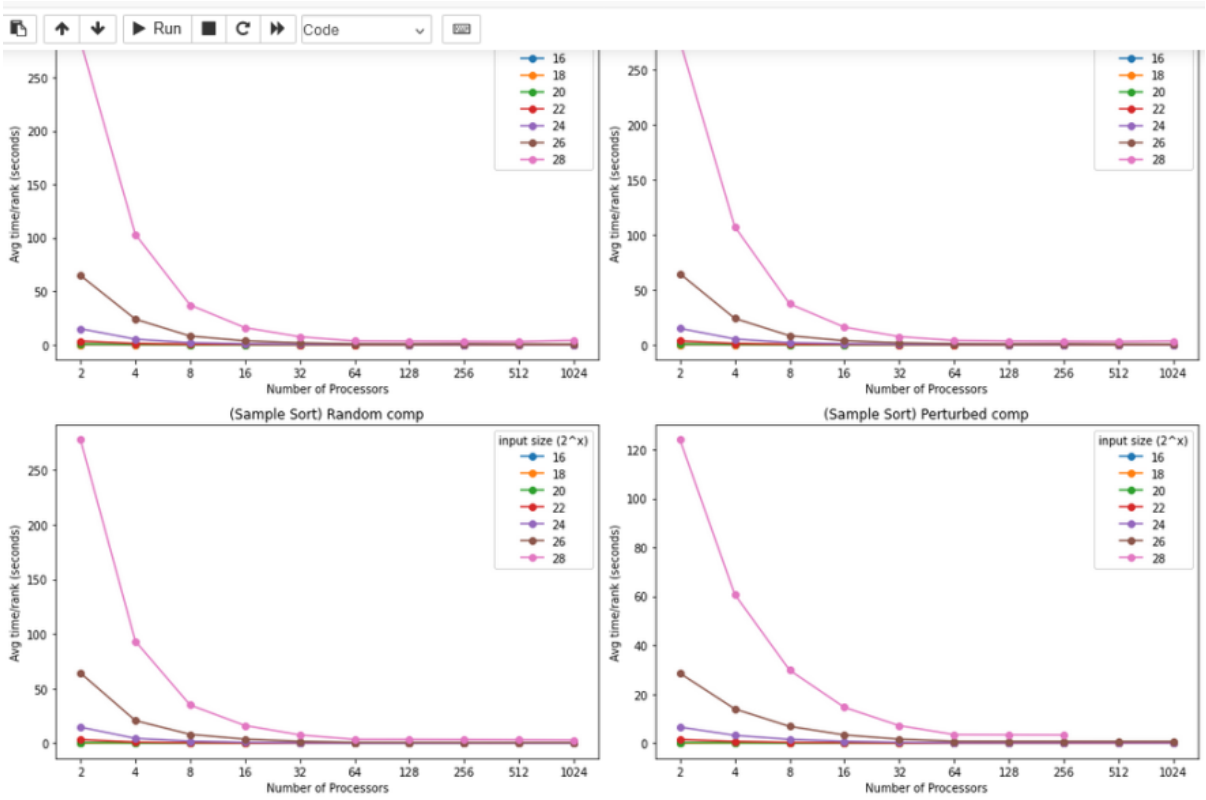
- Total time:



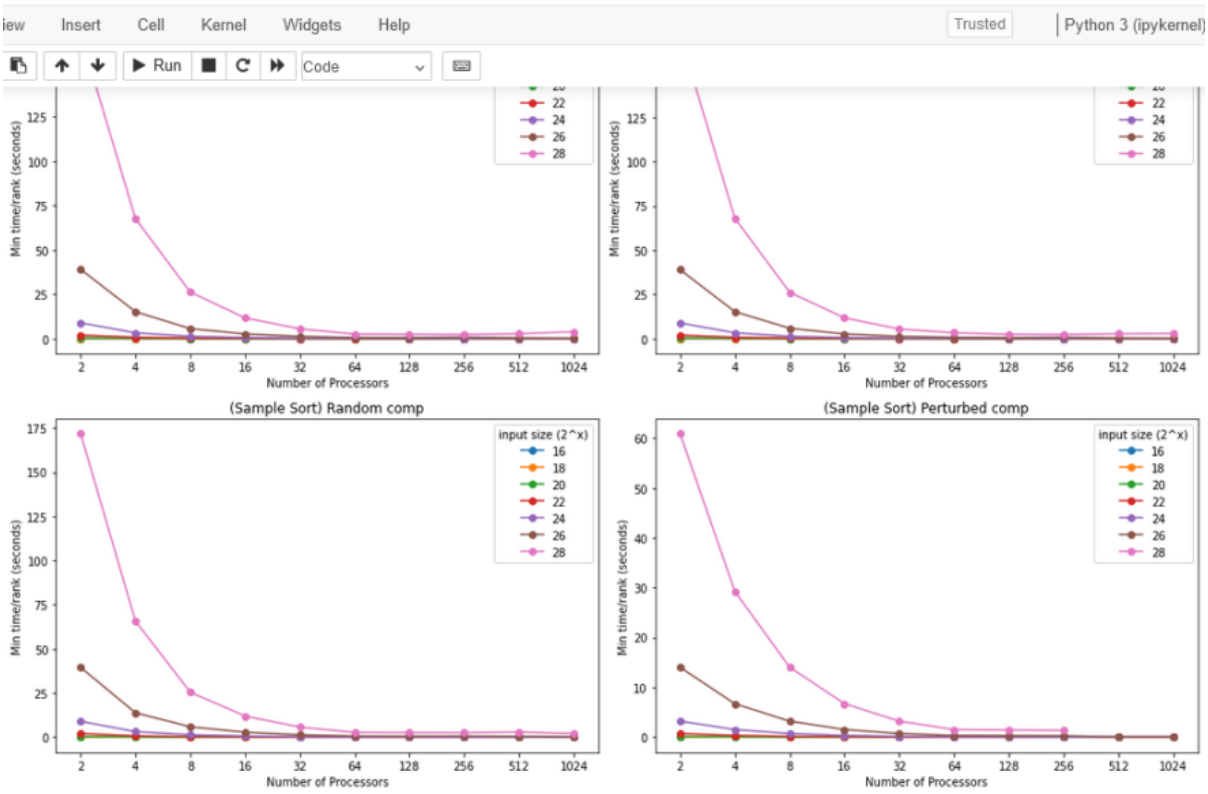

Similarly with main, comm time decreases as the number processors increases up until 64 processors and communication time increases after 64 processors on average per processor. Total time increaes exponentially after 64 processors, likely affecting total time a lot for main. This is likely due to message calls such as MPI_AlltoAll that send and gather from all other processors, in my case the data split by the partitions into their respective final buckets.

- **comp:**

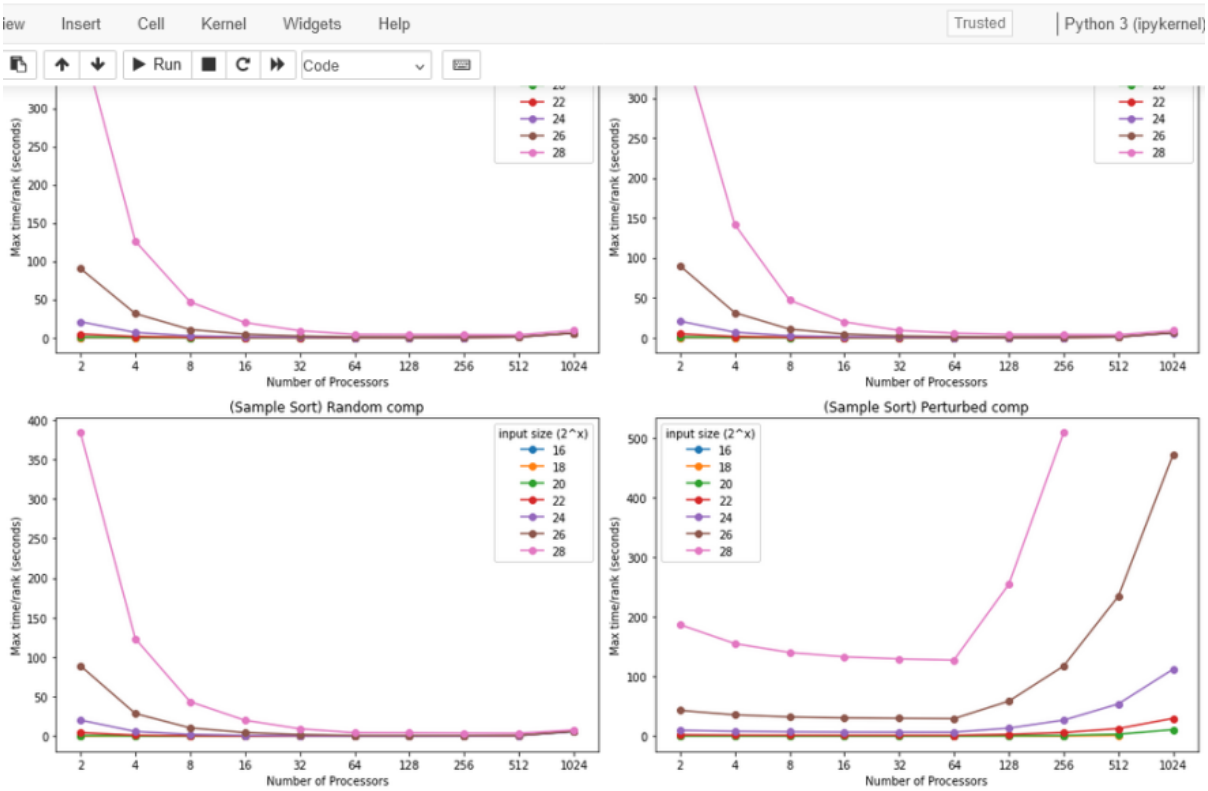
  - Avg time/Rank:



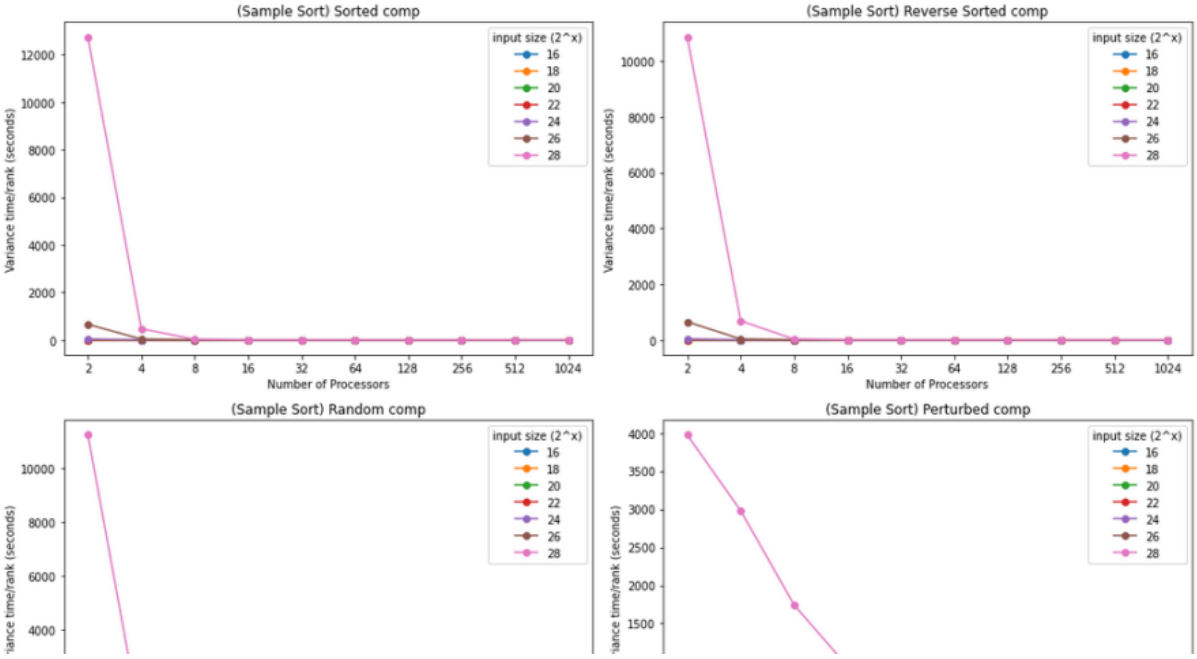
- Min time/Rank:



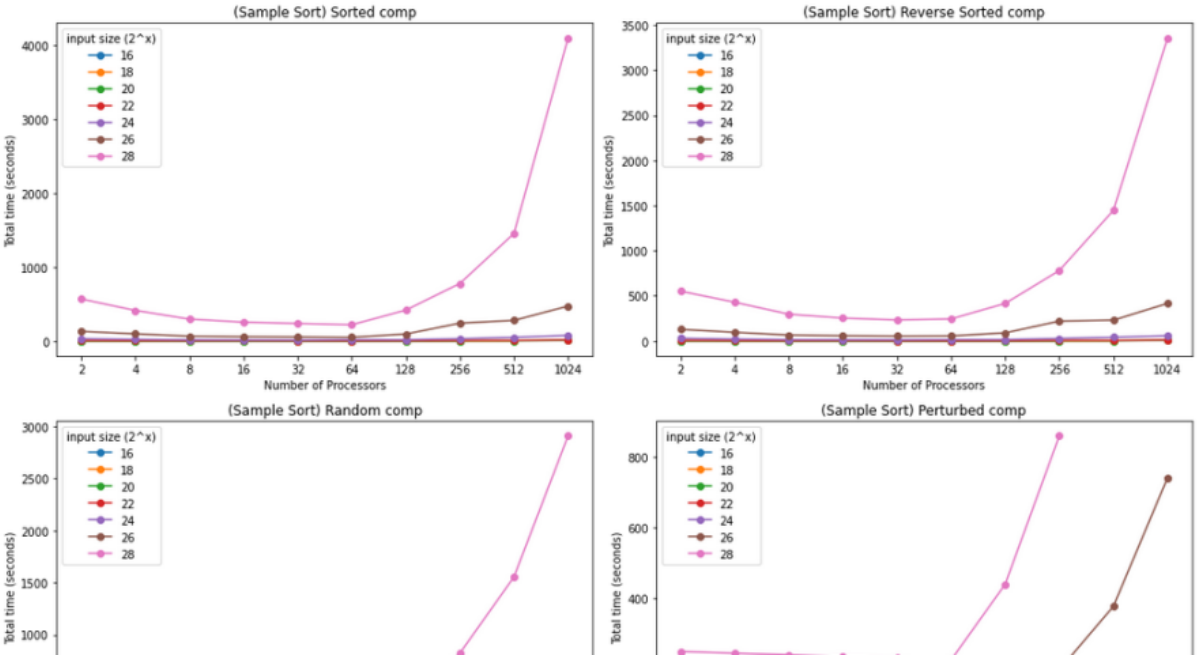

- Max time/Rank:



- Variance time/Rank:



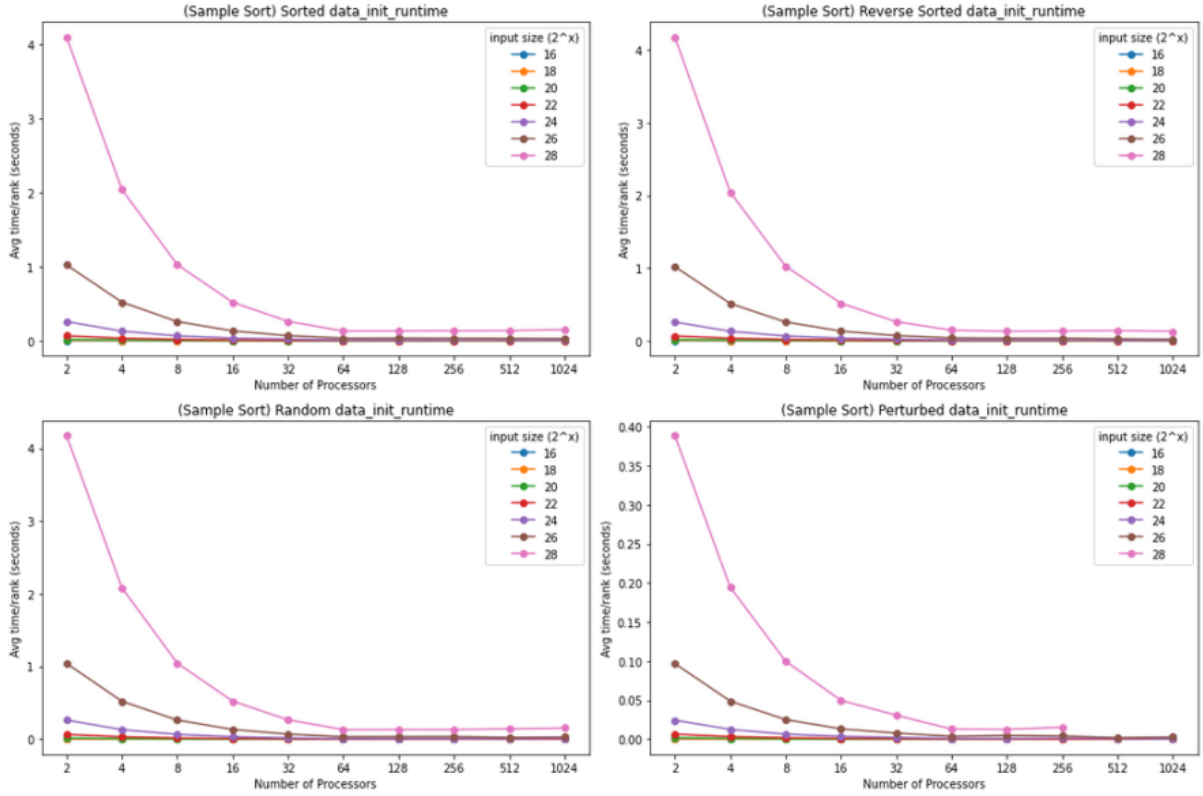

- Total time:



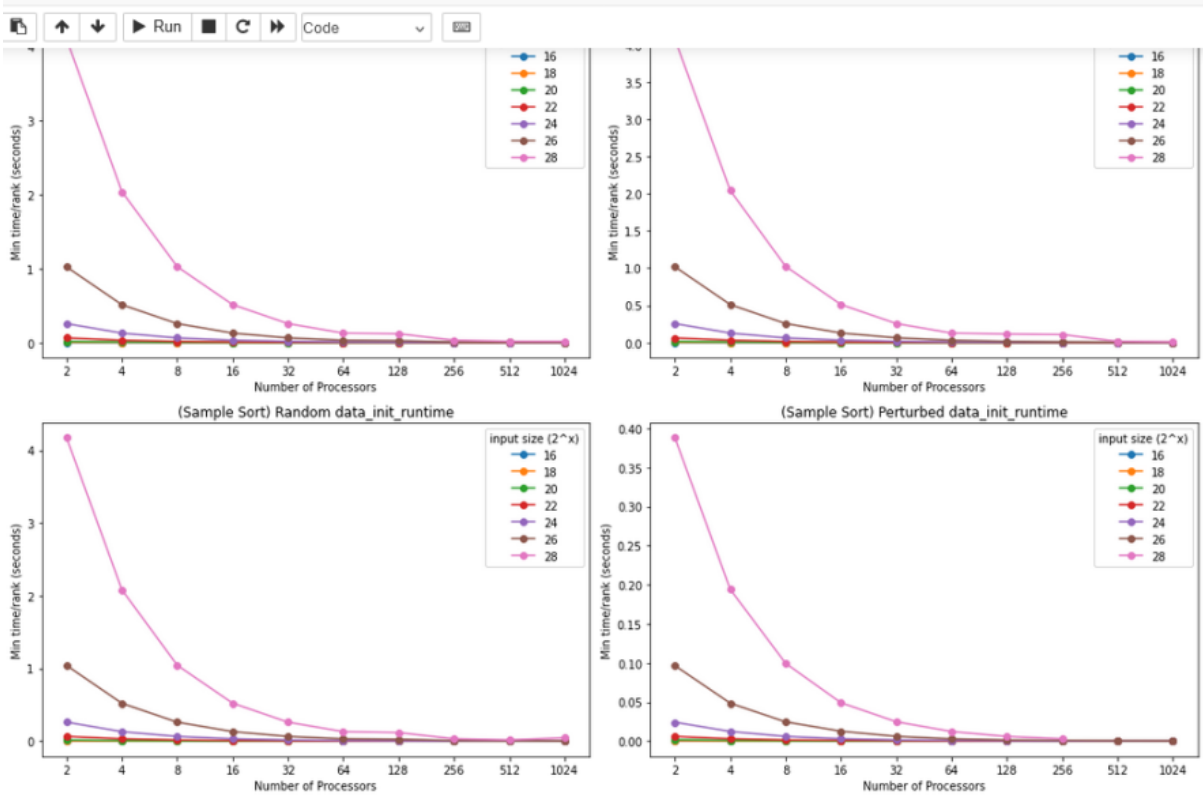

On average, computation time decays exponentailly on average as the number of processors increases for all input types. With the communication time taken out, one can see that operations like computing partition sizes can be done quicker in more parallel.

- **Data generation:**

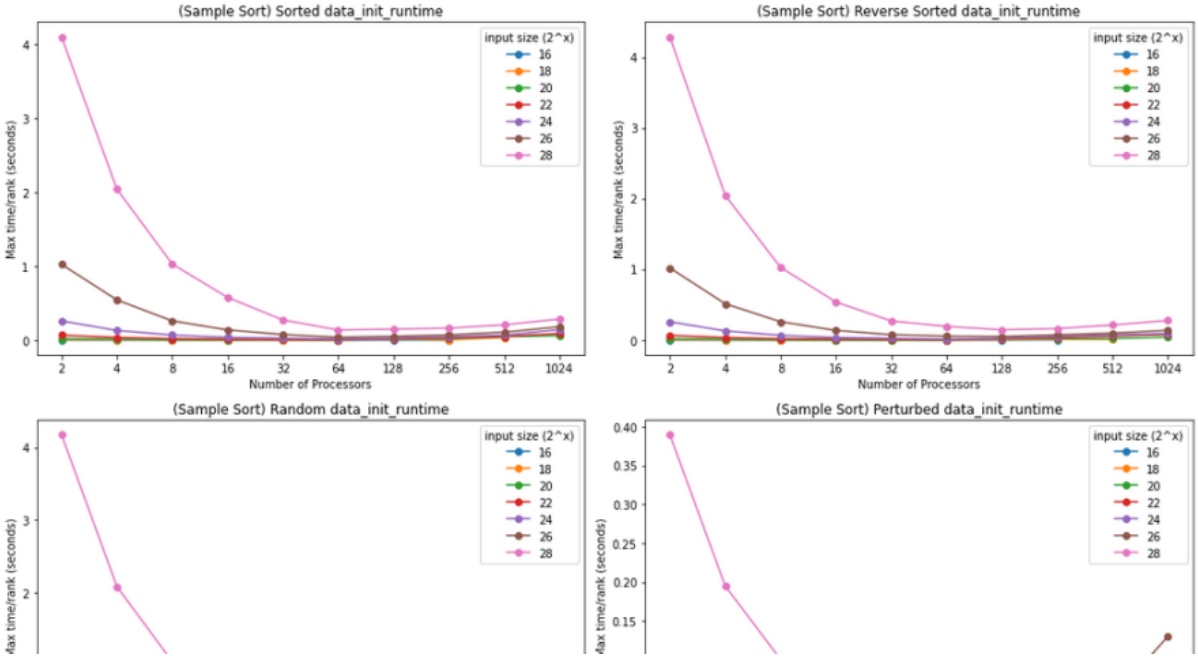
- Avg time/Rank:



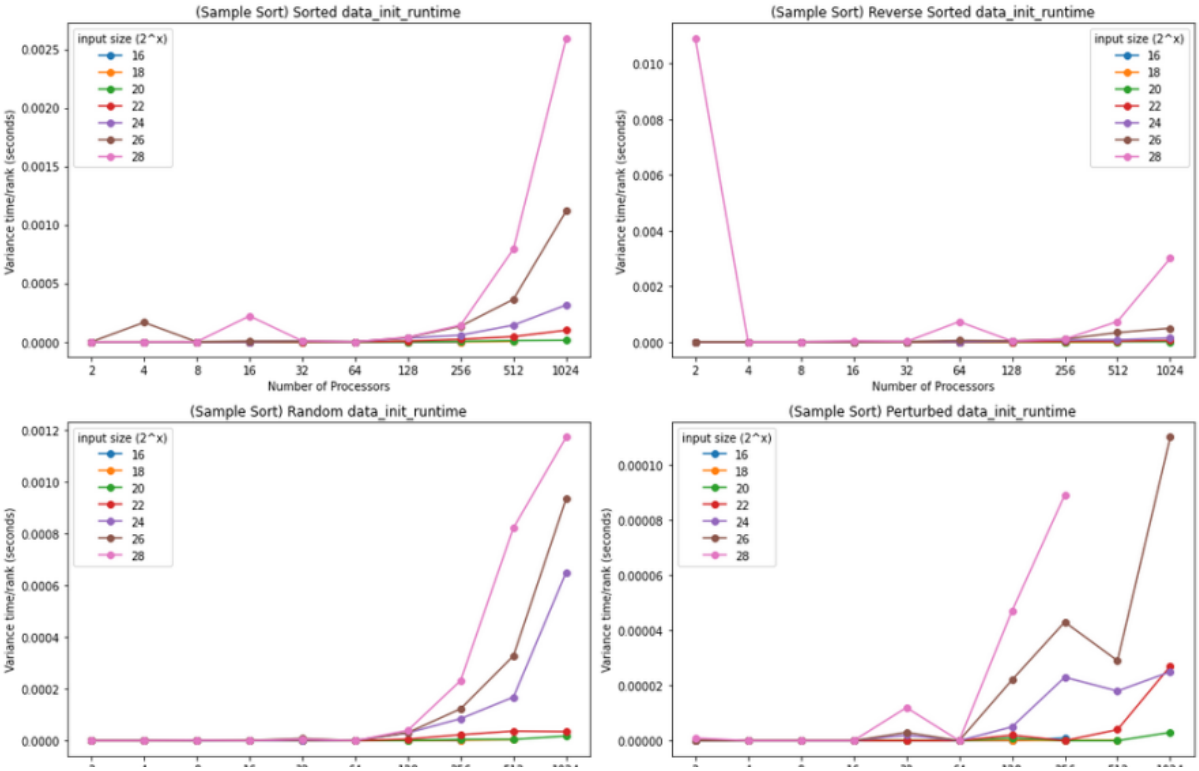

- Min time/Rank:



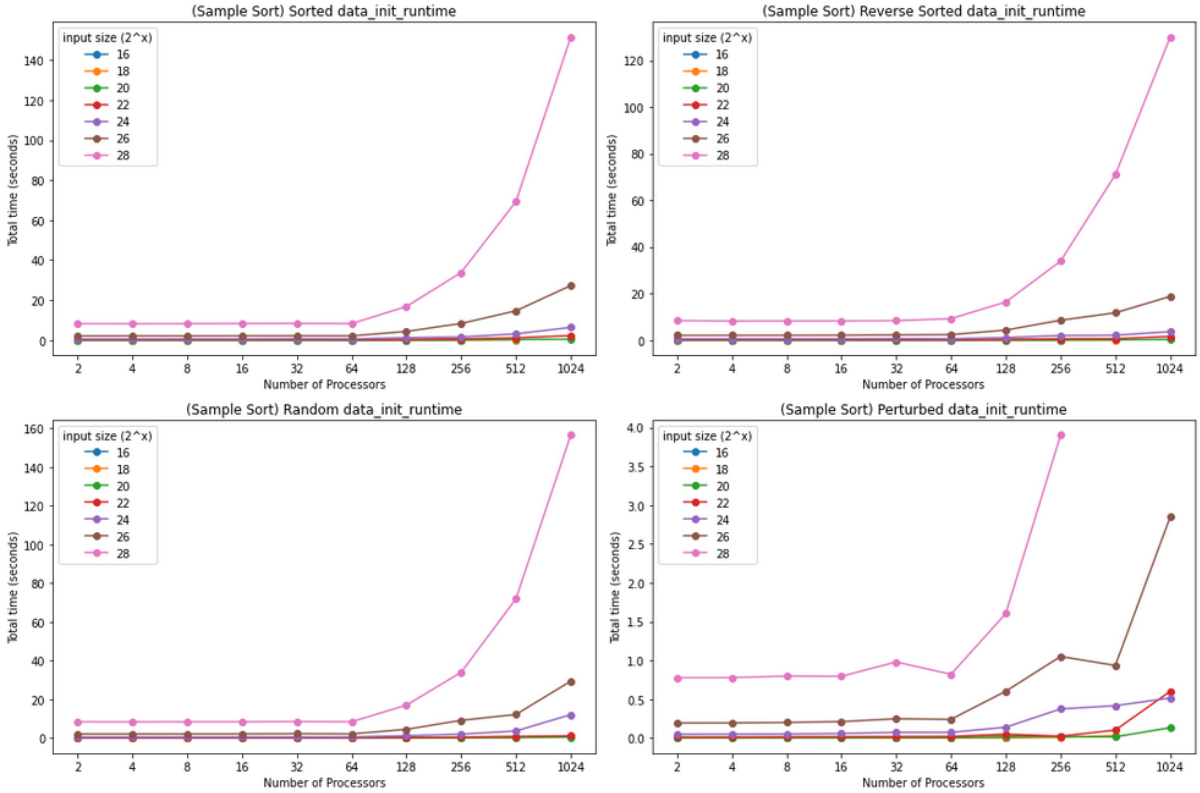
- Max time/Rank:




- Variance time/Rank:



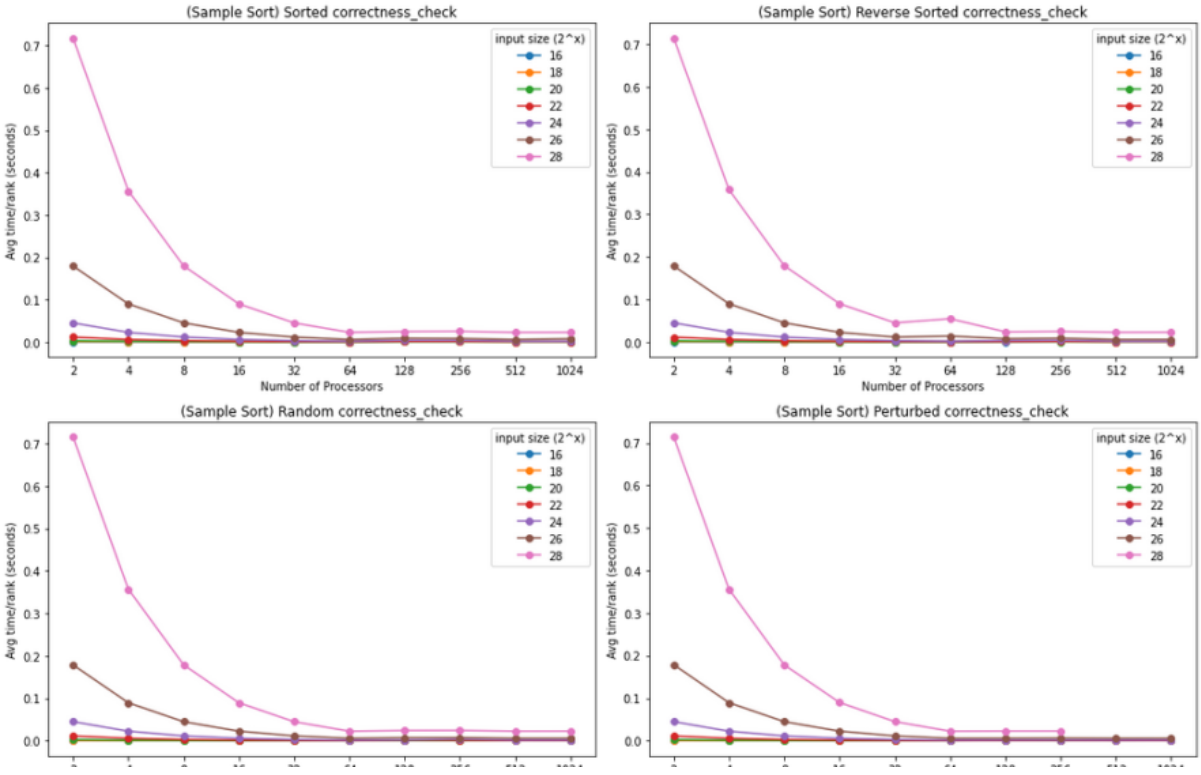
- Total time:



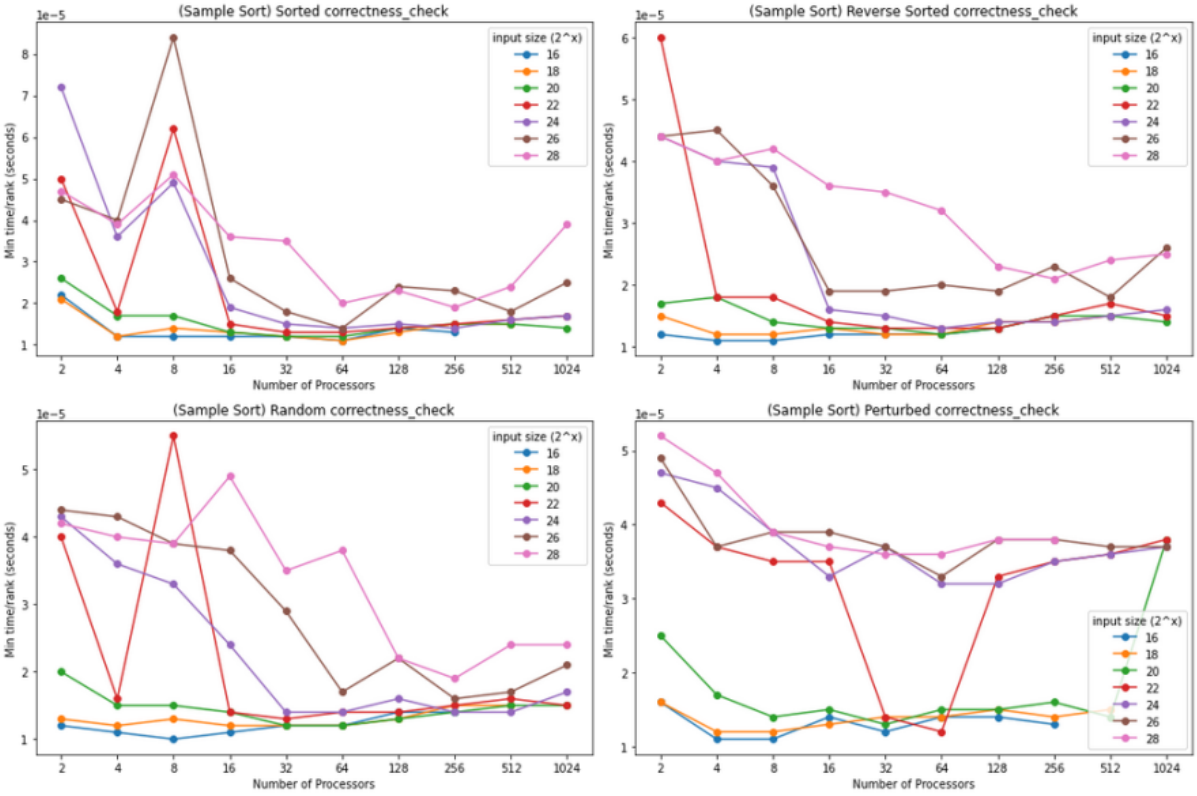

Data generation, much similar to computation, decays in time exponenitally as the number of processors increases on average.

- **Correctness check:**

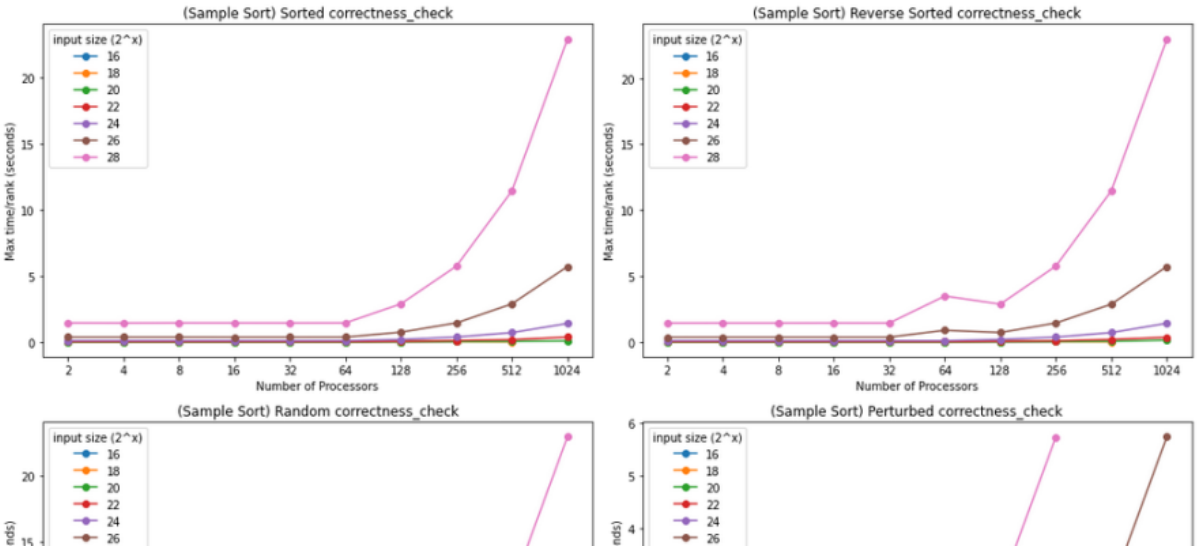
  - Avg time/Rank:



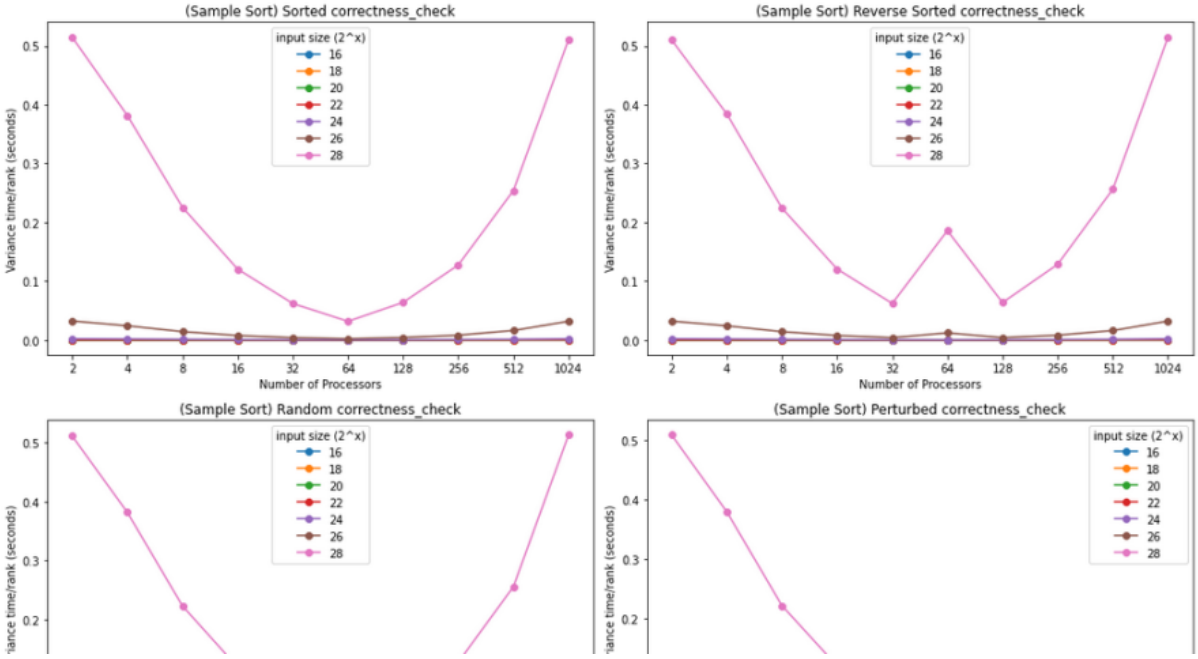
- Min time/Rank:



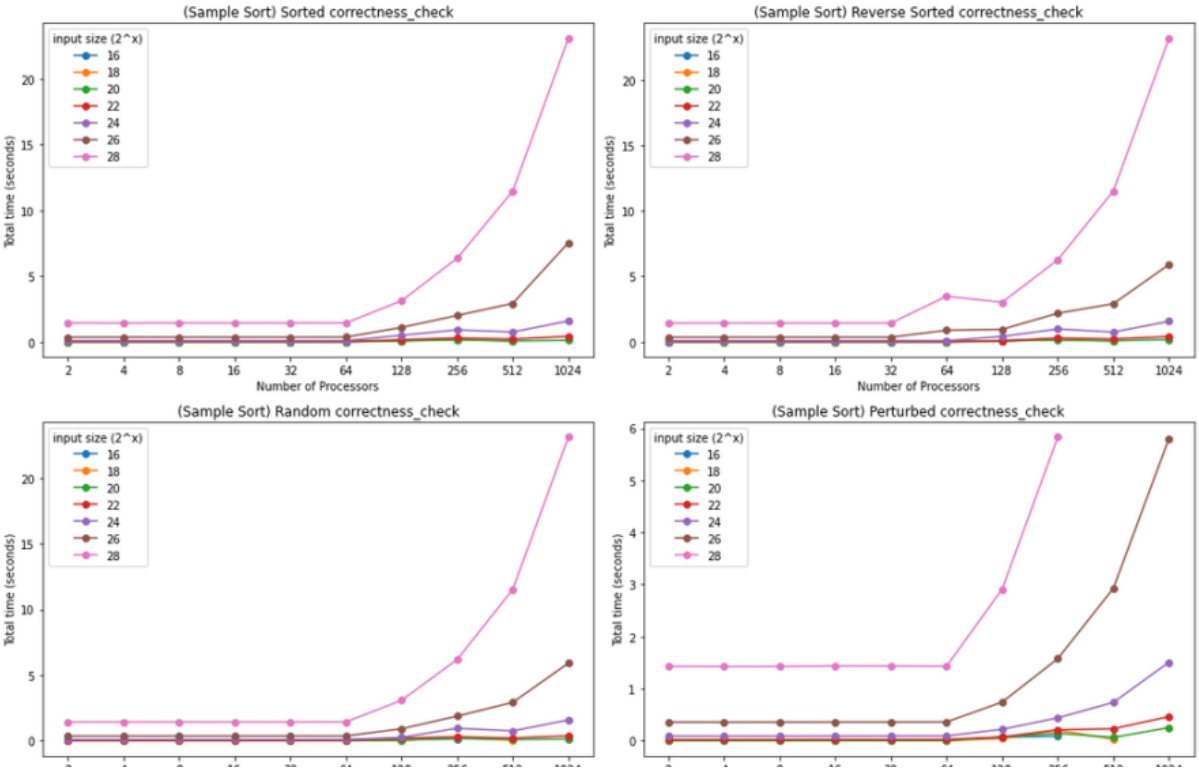

- Max time/Rank:



- Variance time/Rank:




- Total time:

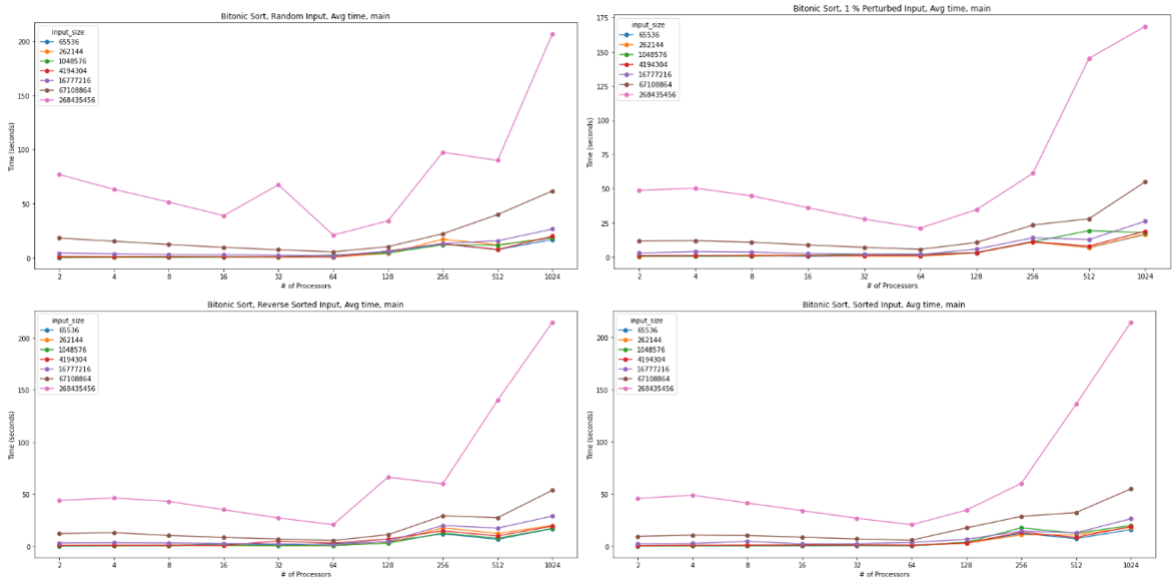



Similar to data generation and computation, correctness check decays in time exponenitally as the number of processors increases on average.

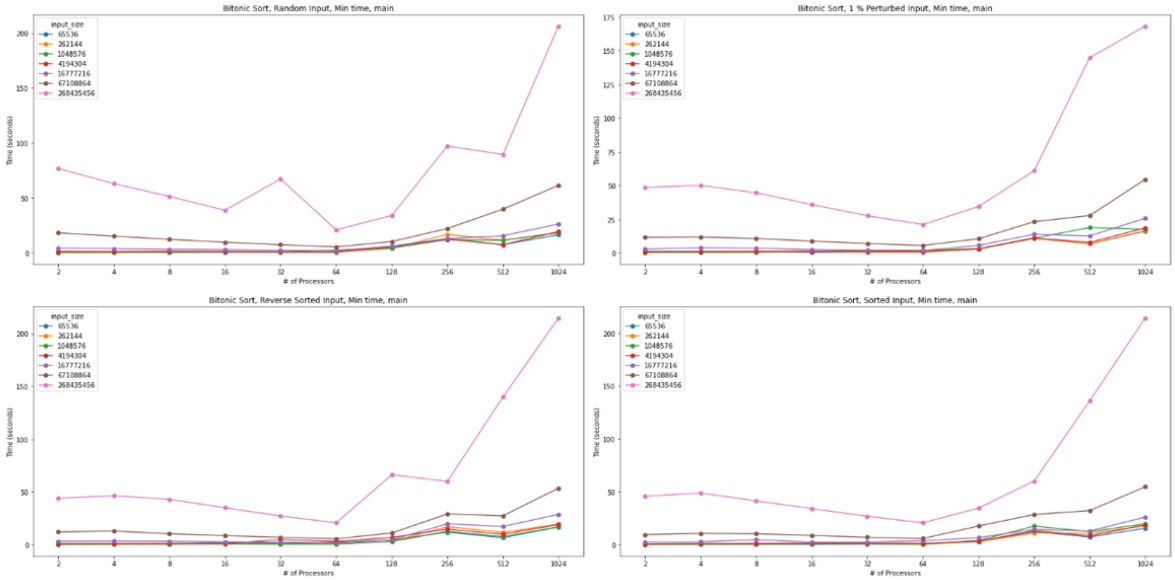
## III) Bitonic Sort

- **main:**

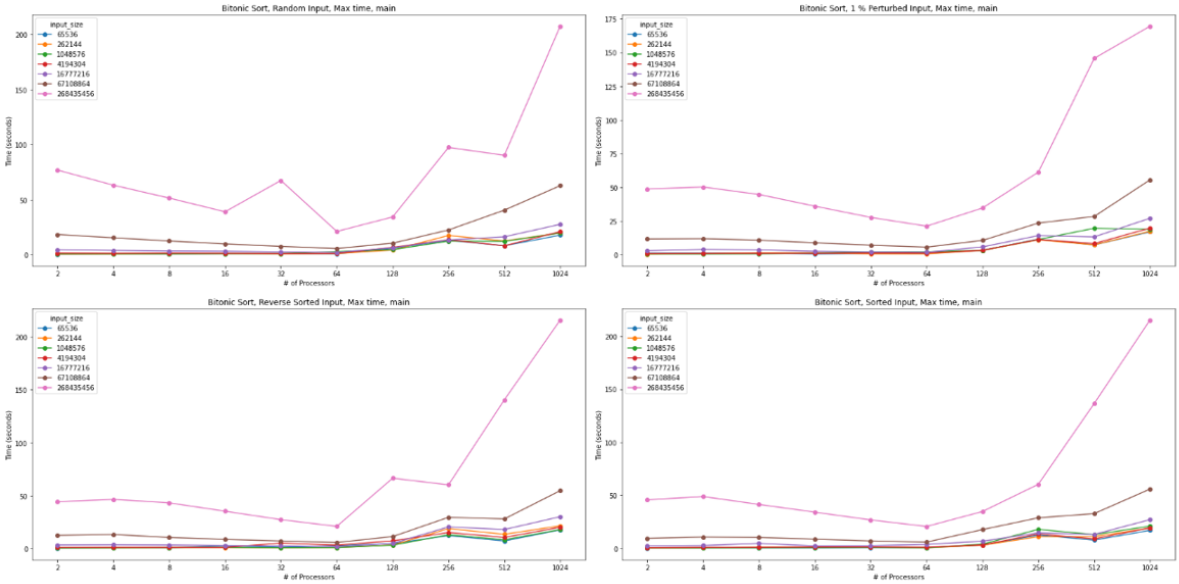
- Avg time/Rank:



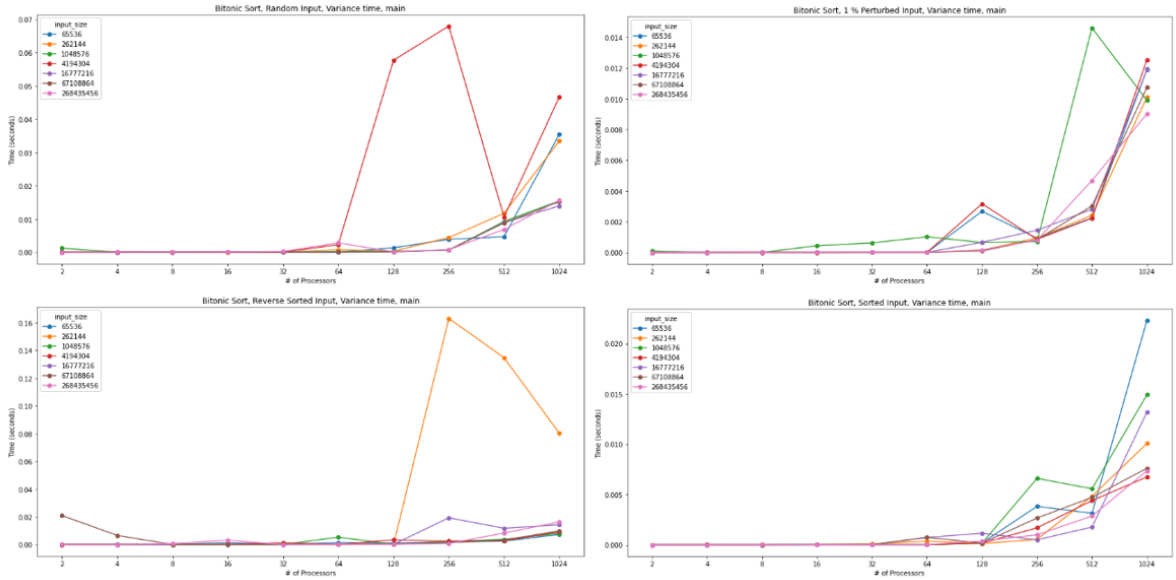

- Min time/Rank:



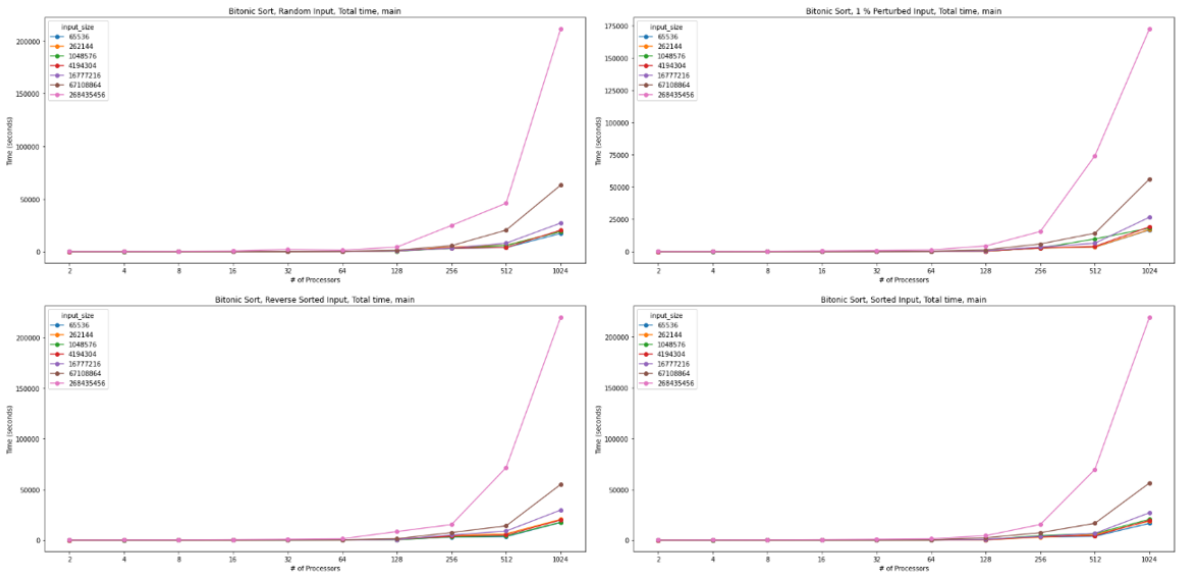

- Max time/Rank:



- Variance time/Rank:



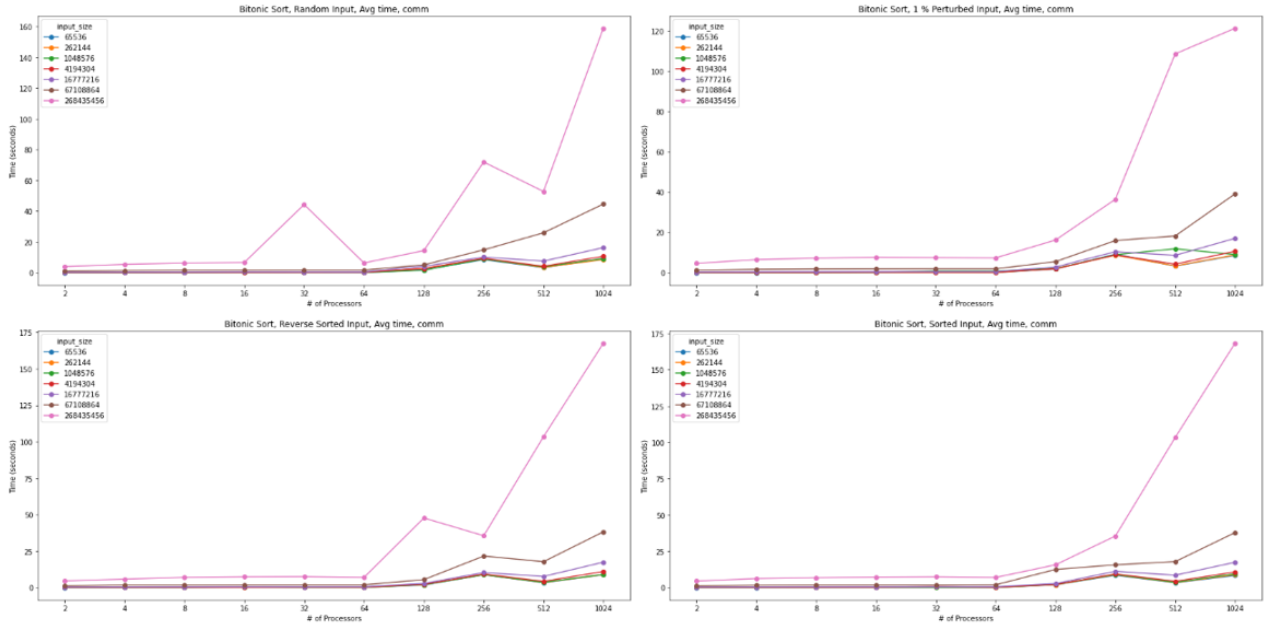

- Total time:



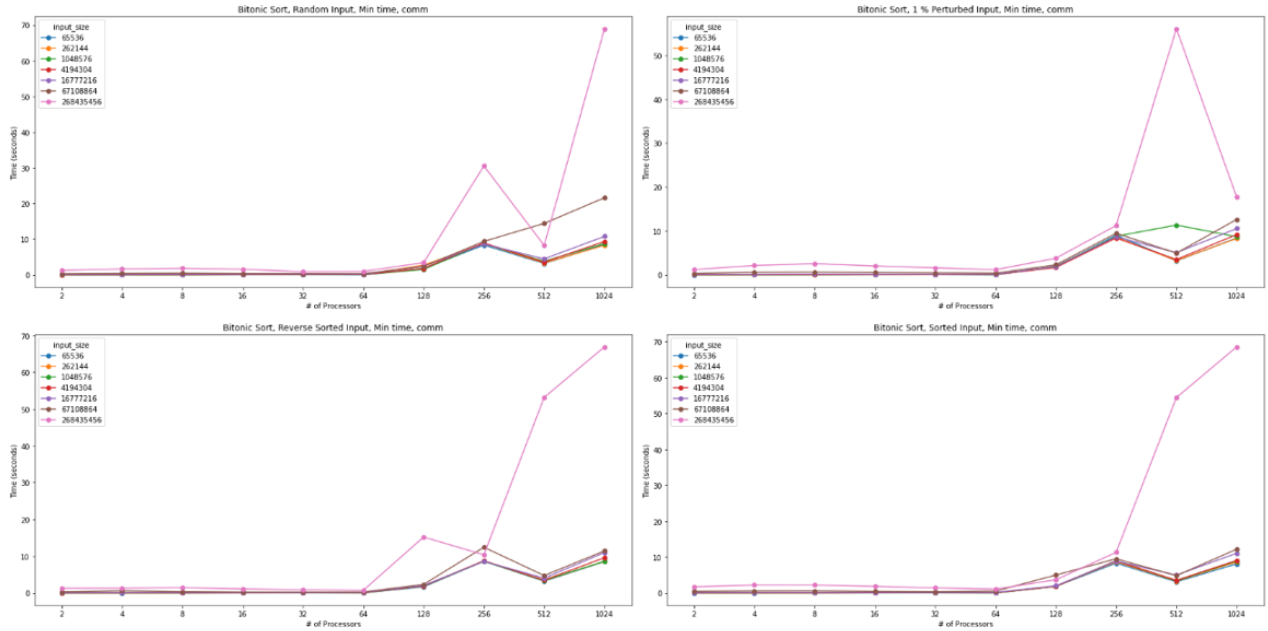

Across all input types, the runtime of main remains generally consistent. This is because bitonic sort operates on its data independent of order. After 64 processes, the total time increases dramatically, and this is because of the input generation/correctness check functions, which were done all on the master process, as well as increasing communication overhead. The input generation and correctness check functions are the main bottleneck on performance, and should be parallelized.

- **comm:**

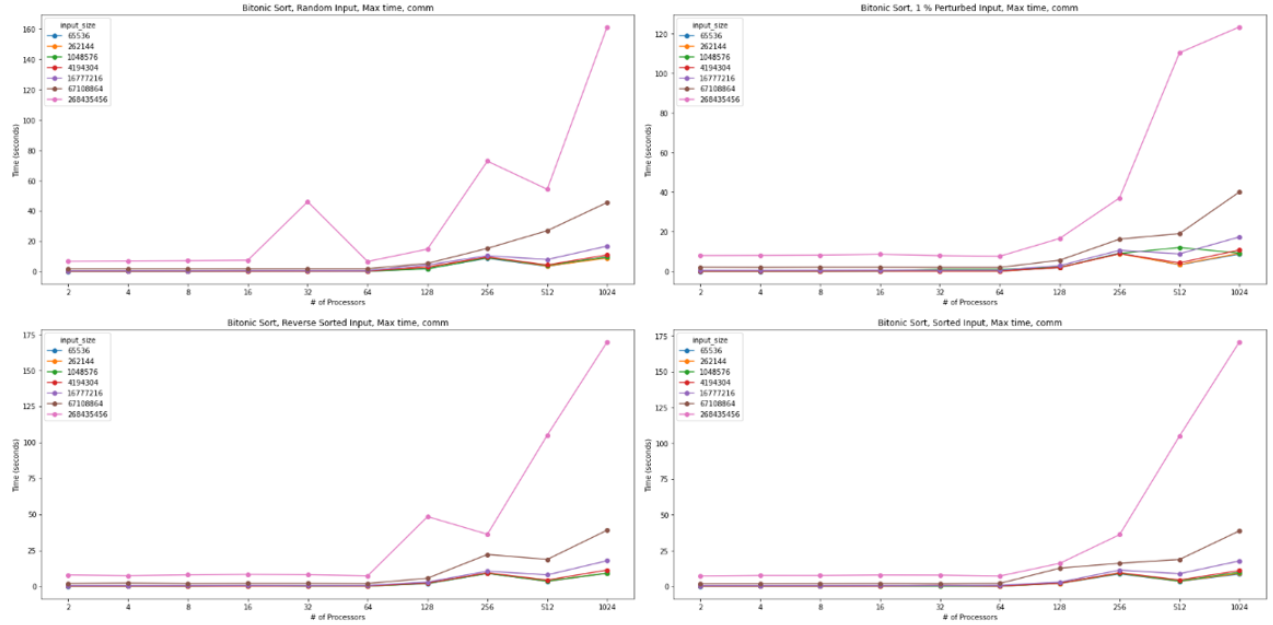
- Avg time/Rank:



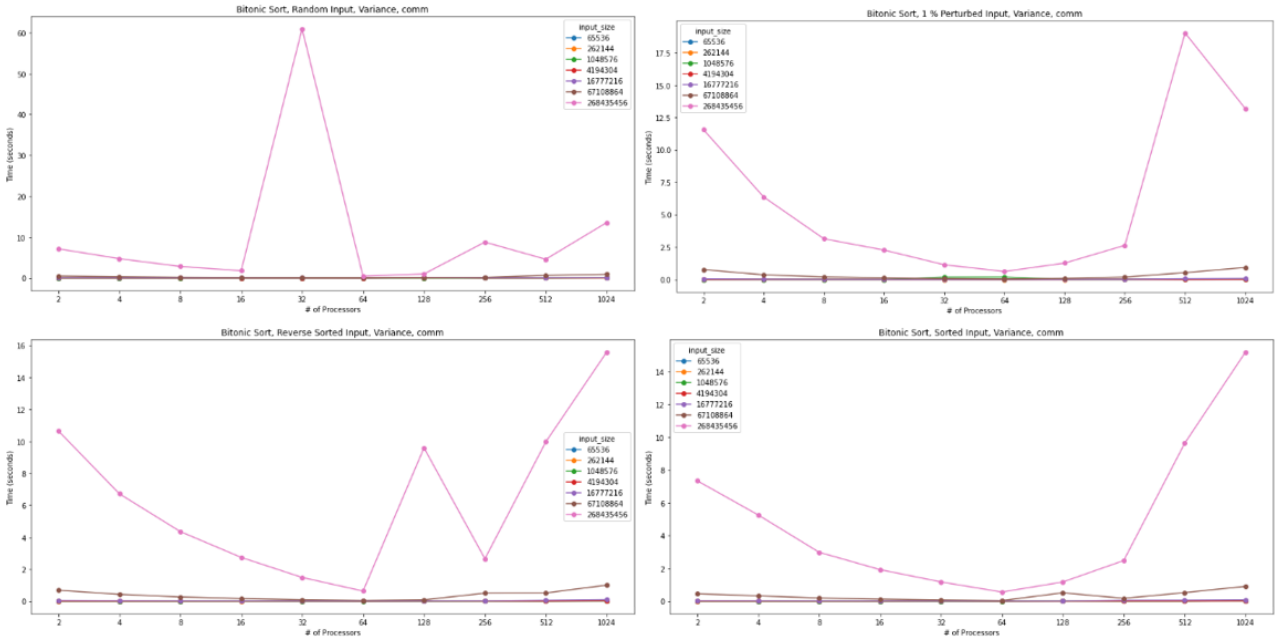

- Min time/Rank:



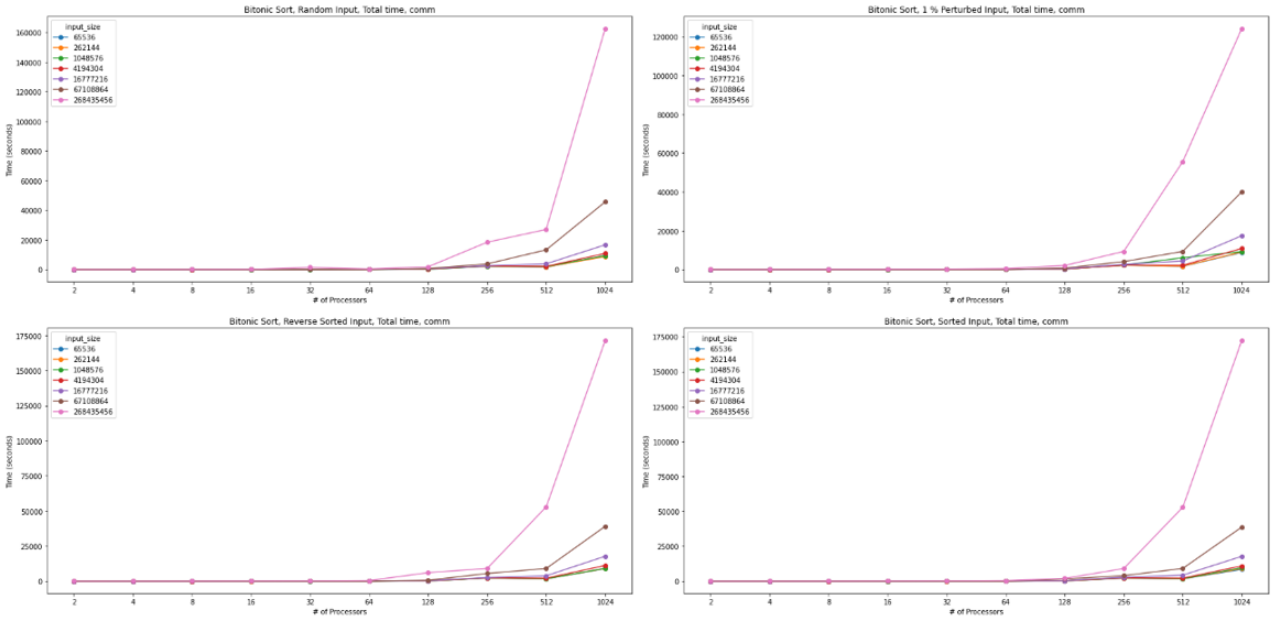
- Max time/Rank:




- Variance time/Rank:

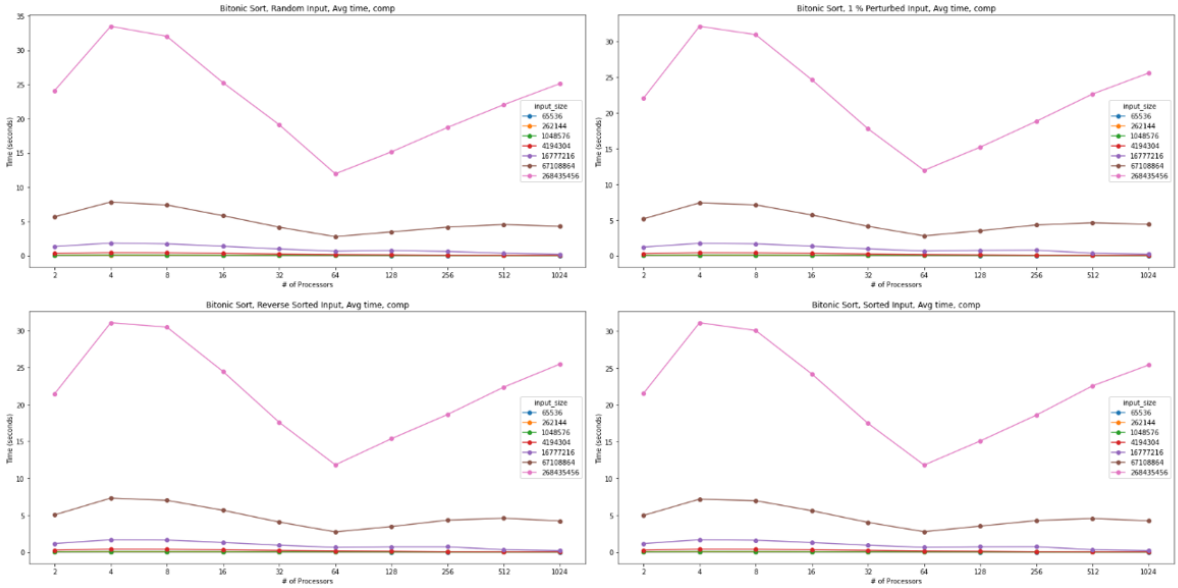


- Total time:

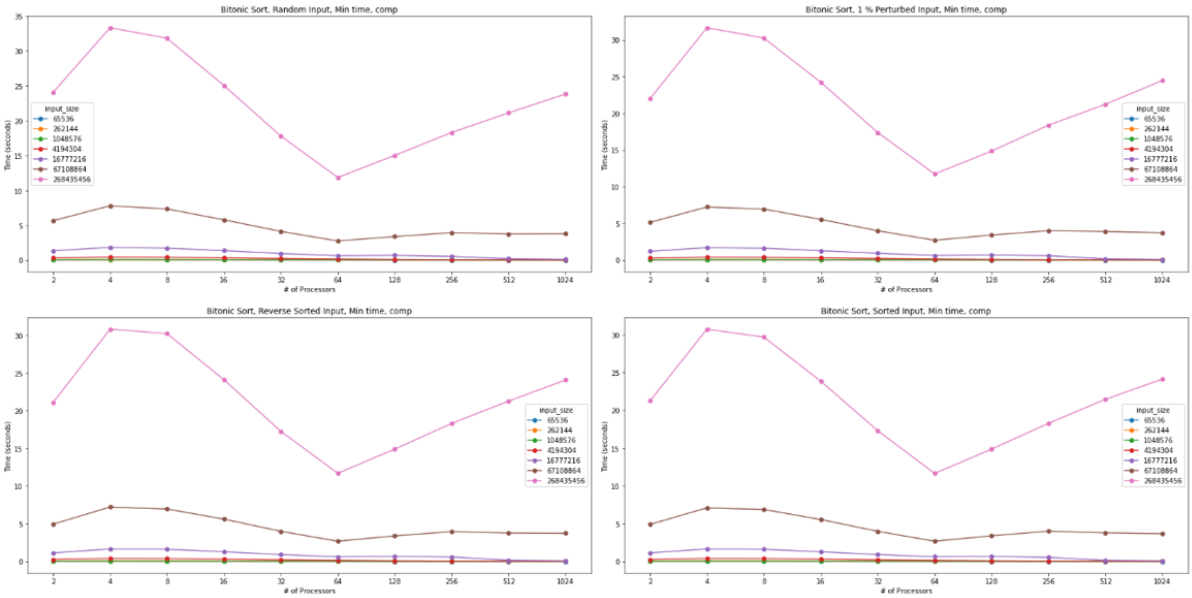



Once again, after 64 processors, each processor begins spending significantly more time communicating, which indicates that 64 processors is the point where resources are utilized most effectively. Additionally, as pointed out before, there is not much of a significant difference in performance between input types. Once past the 128 processor mark, communication operations become very expensive for MPI_Scatter and MPI_Gather, due to network congestion from the large number of processors. These are only used in input generation and the correctness check at the end, and are the largest contributors to communication time. Within the sorting function, MPI_sendrecv is the only communication between processes, and this operation also adds a lot to the overall communication time with a large number of processors. This is due to the fact that there are multiple rounds where each process is communicating with its partner, and each process communicates with all other processes at least once.

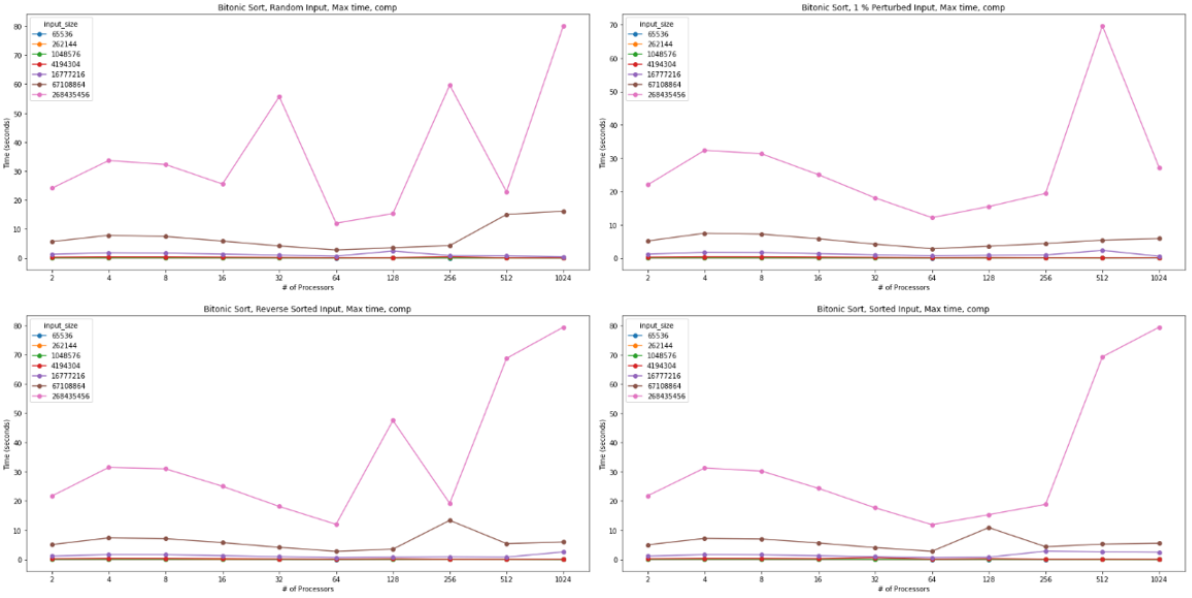
- **comp:**
  - Avg time/Rank:



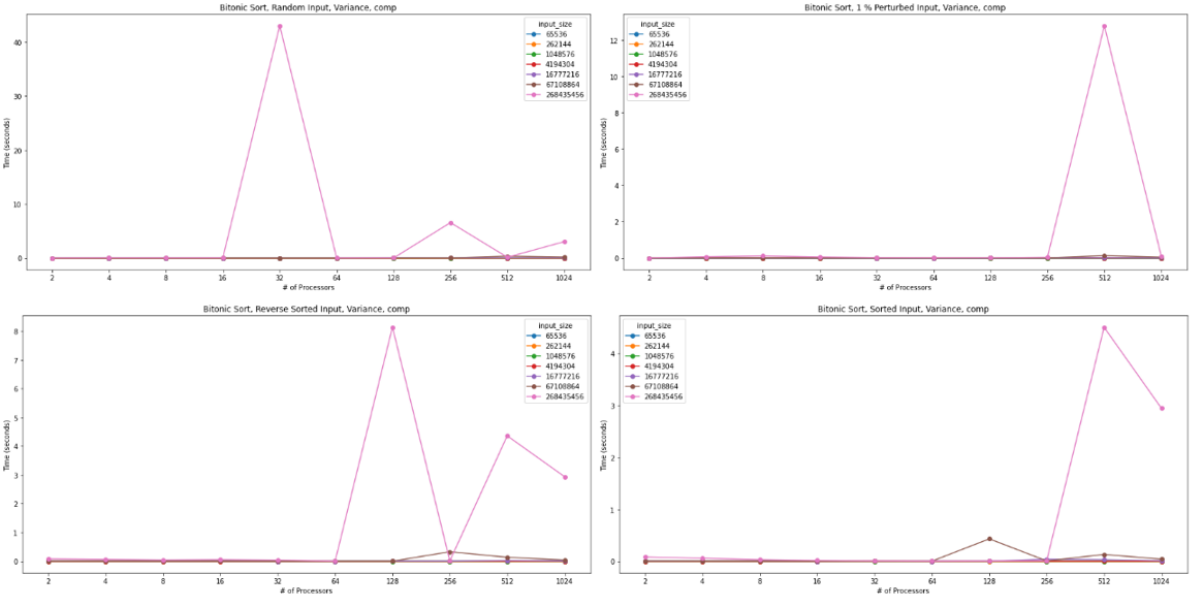
- Min time/Rank:



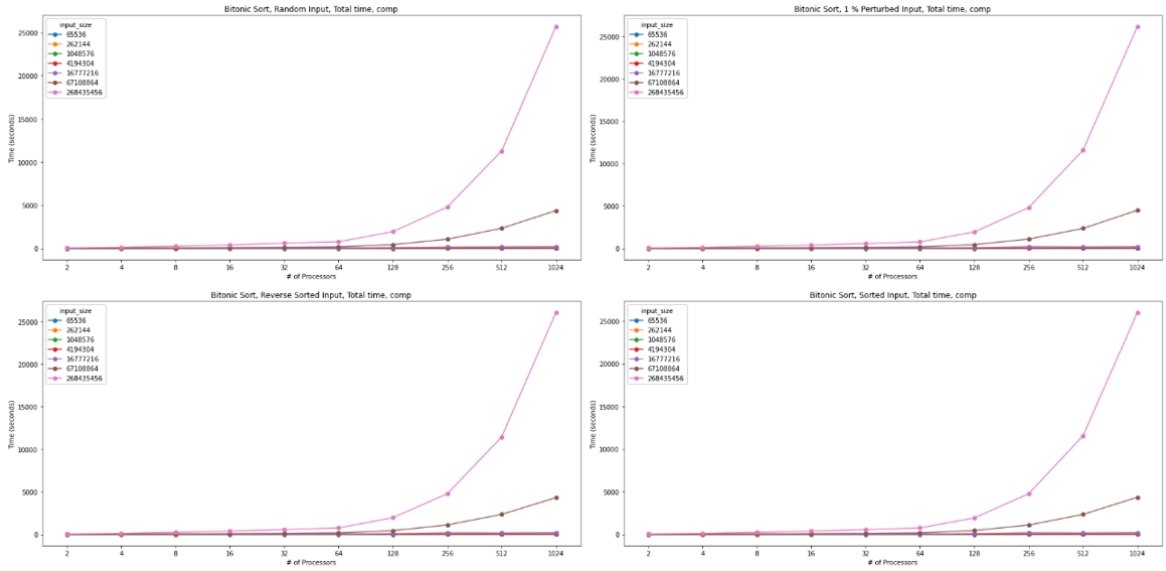

- Max time/Rank:




- Variance time/Rank:



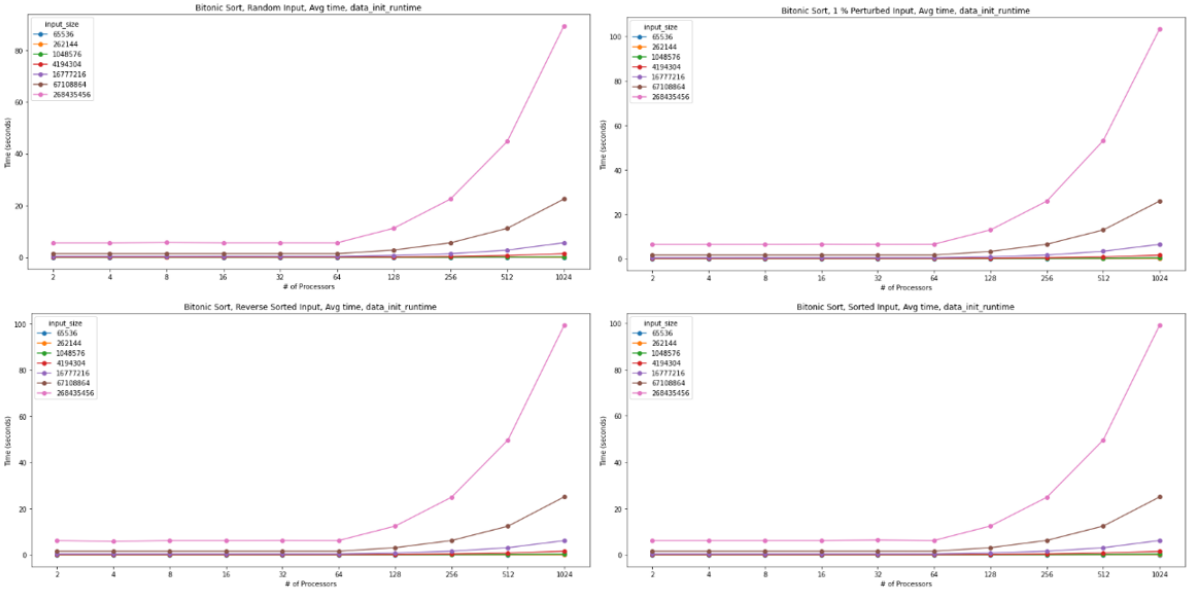
- Total time:



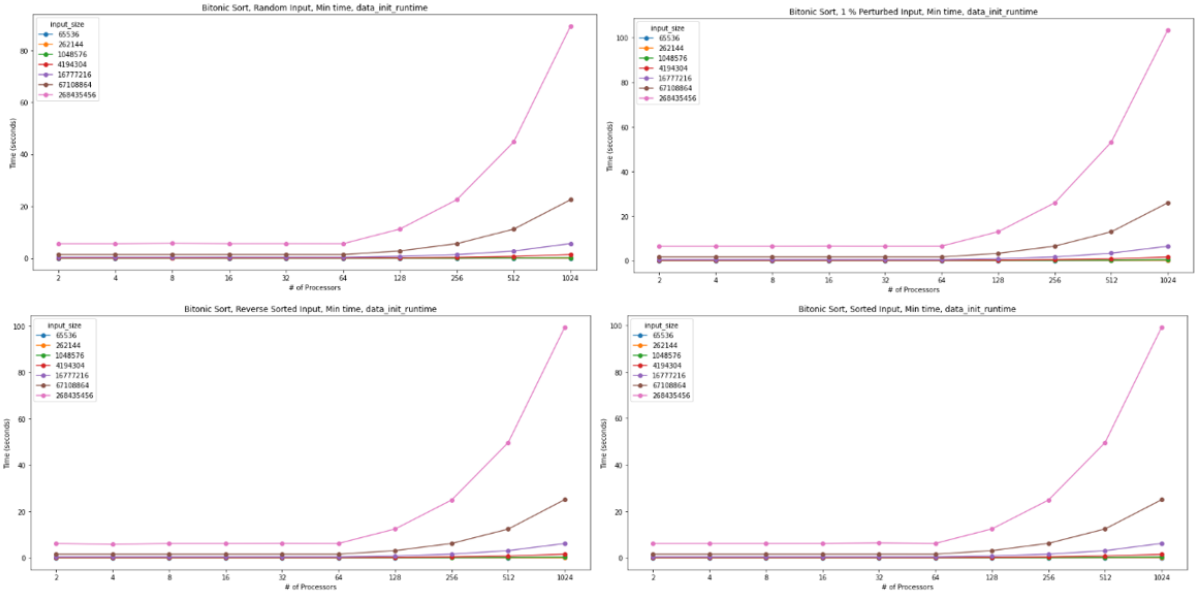

Before the 64 processor mark, the average amount of time spent on computation per processor is decreasing, and after the 64 processor mark, the time spent on computation per processor starts increasing. In theory, these times should be continually going down due to the size of the local array decreasing on each processor. The way that I implemented this may be to blame, due to large memory overhead caused by copying over the partner process's array, then concatenating both, sorting, and keeping the higher or lower half. The portion of the code contributing to this pattern the most is comp_large, which consists of the local sorting function, qsort. Past 64 processors, the length of the array in qsort (quick sort) may be too short for it to benefit from being parallelized.

- **Data generation:**

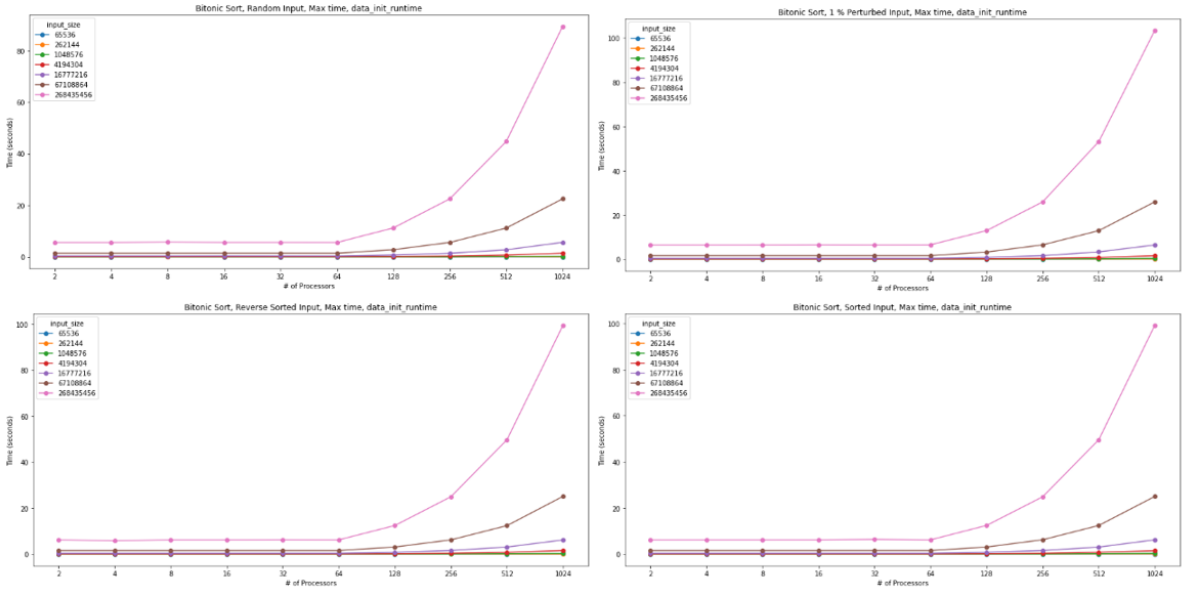
  - Avg time/Rank:



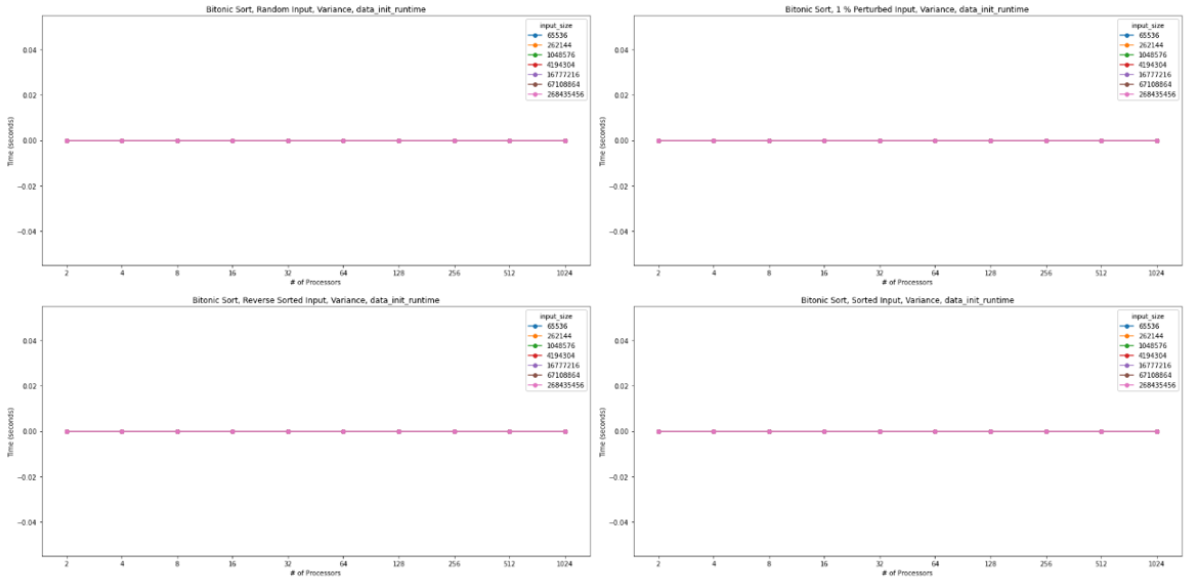
- Min time/Rank:



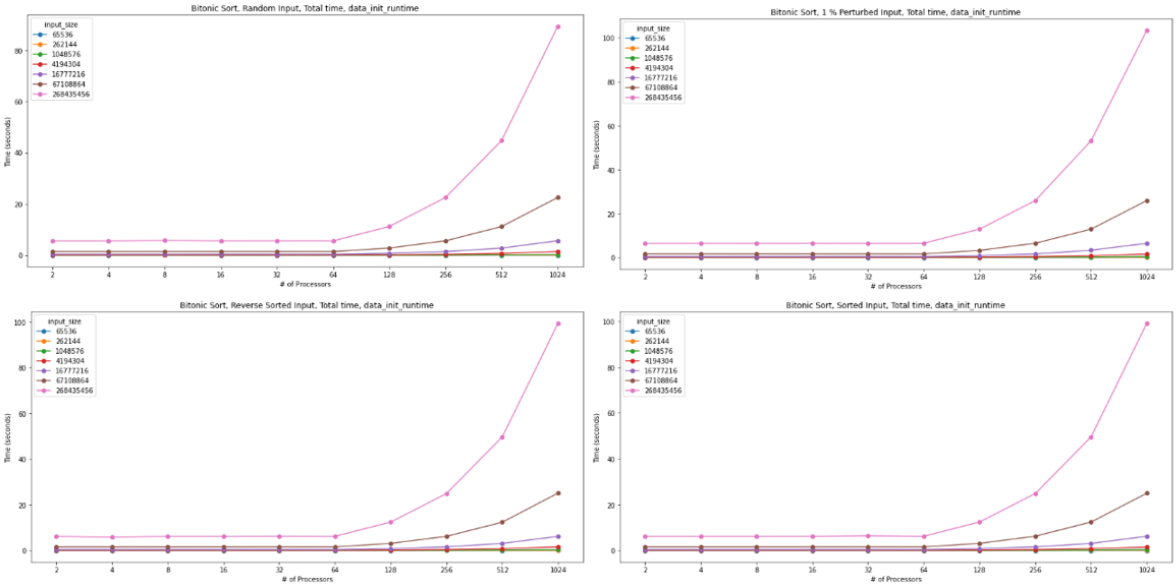

- Max time/Rank:




- Variance time/Rank:

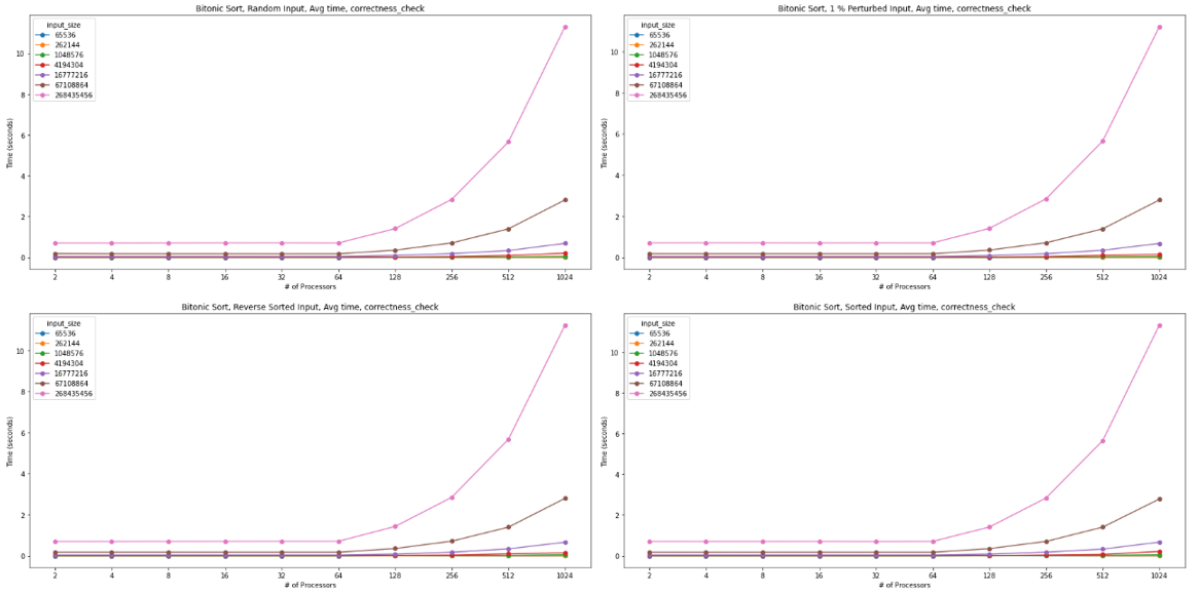


- Total time:

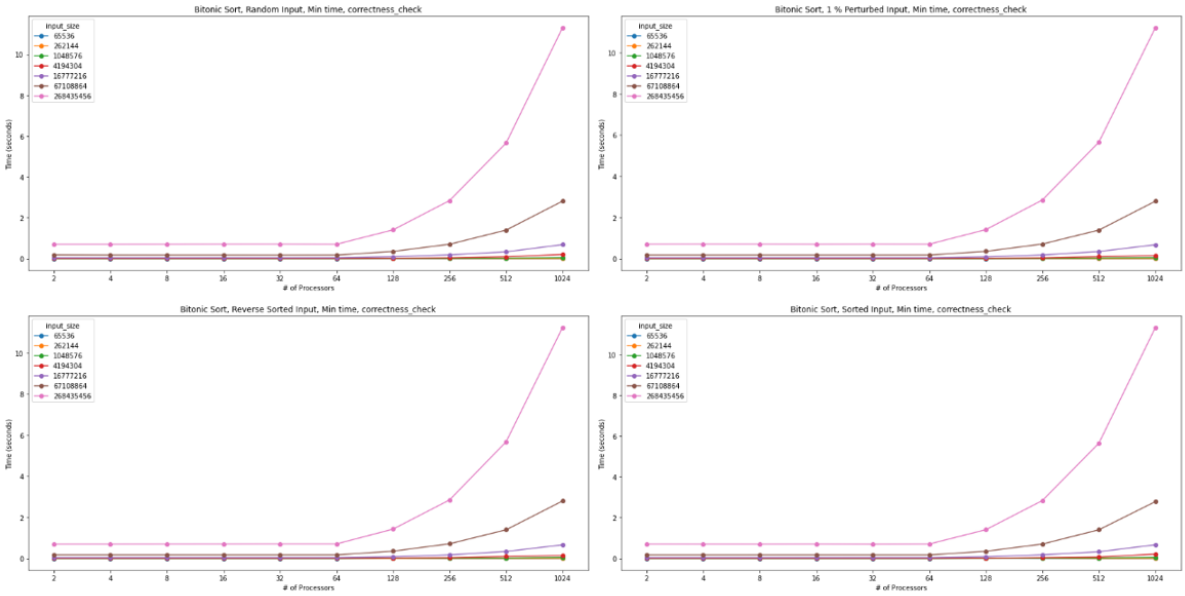



- **Correctness check:**

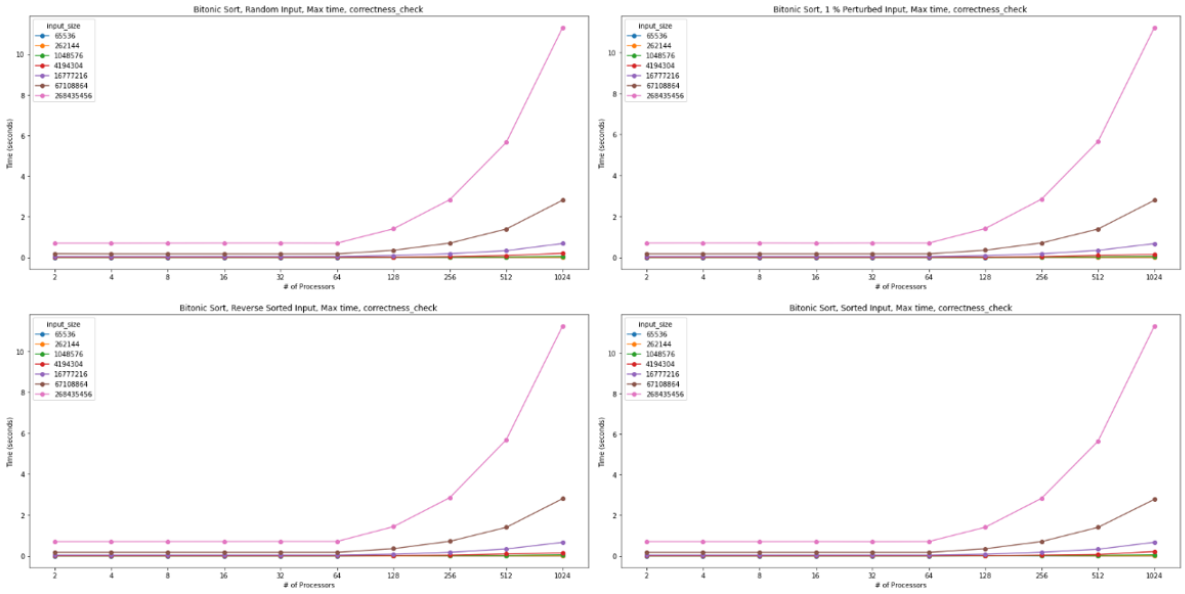
  - Avg time/Rank:



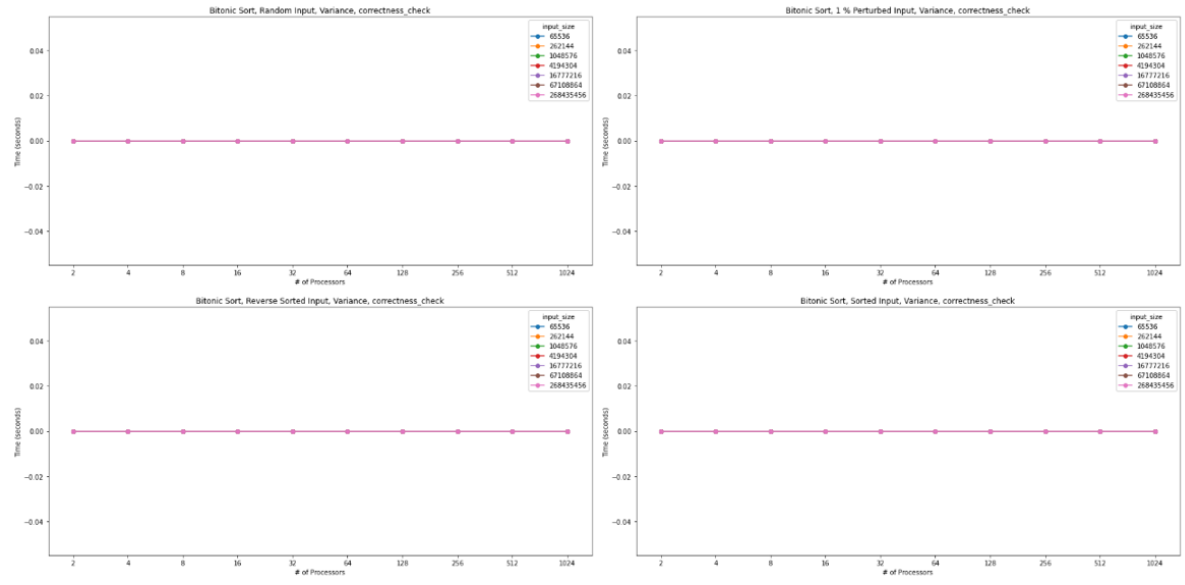
- Min time/Rank:



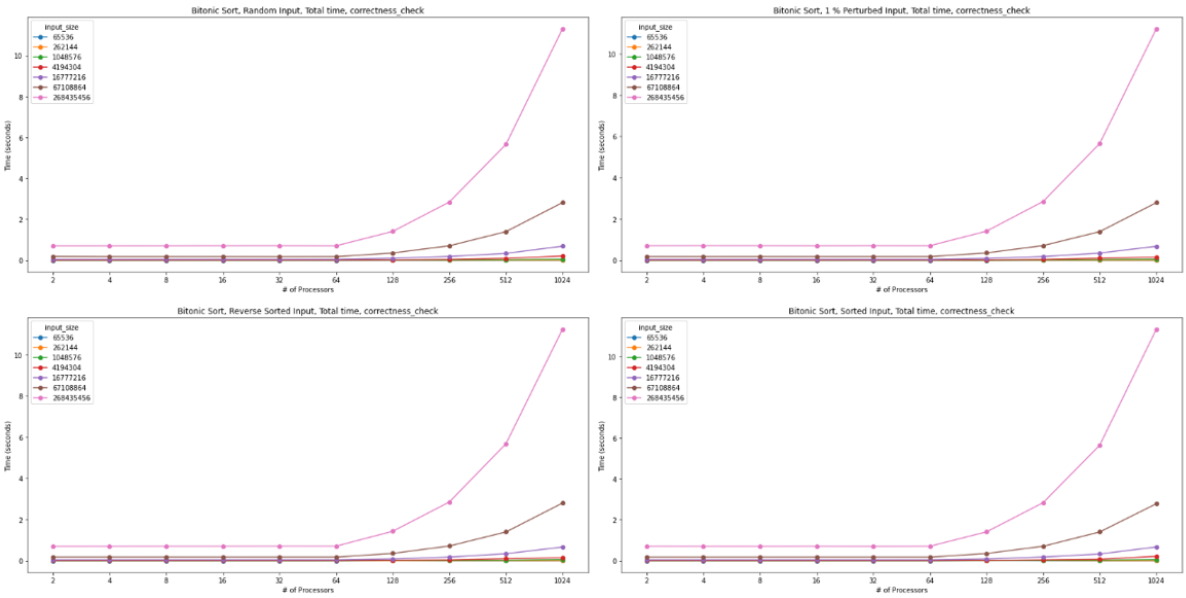

- Max time/Rank:




- Variance time/Rank:



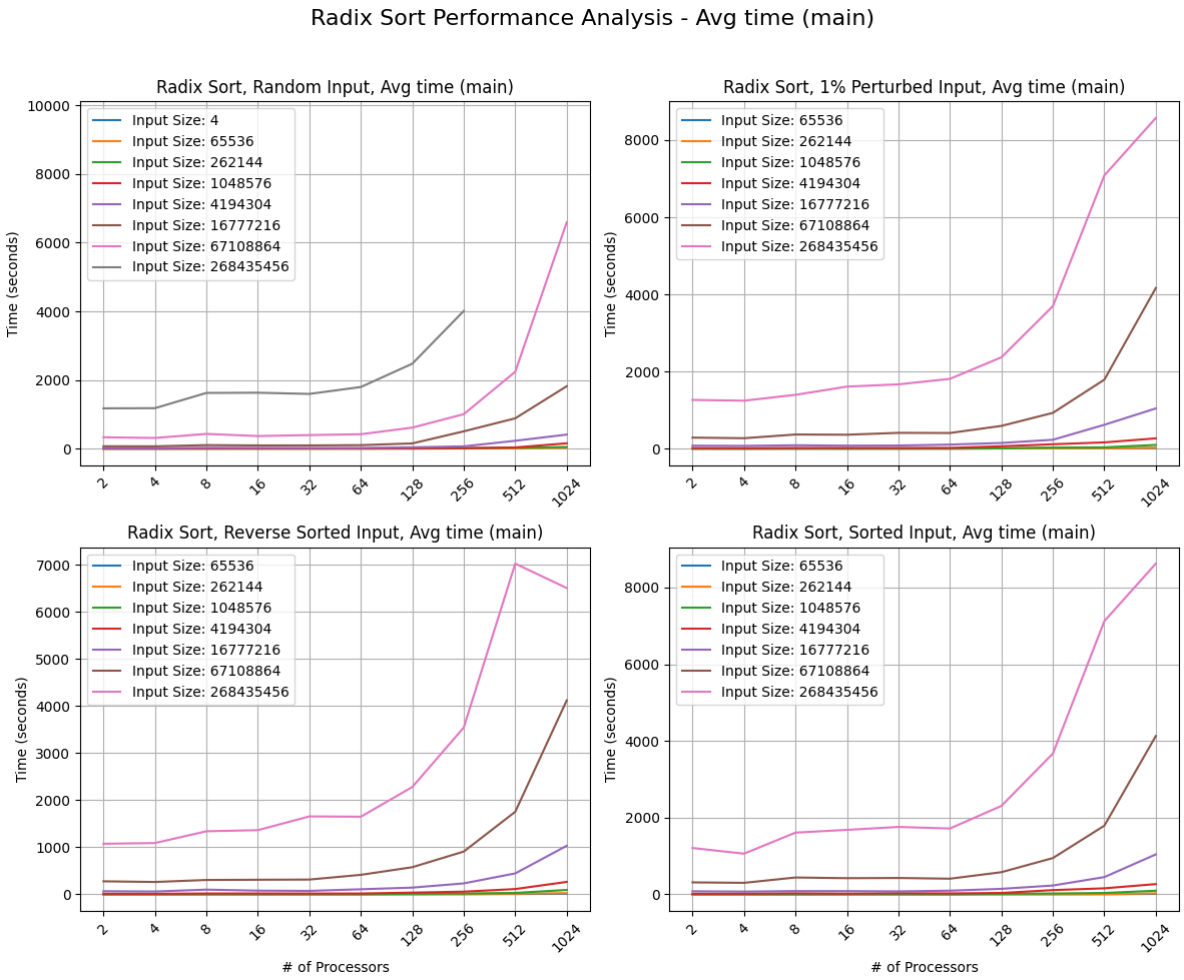
- Total time:



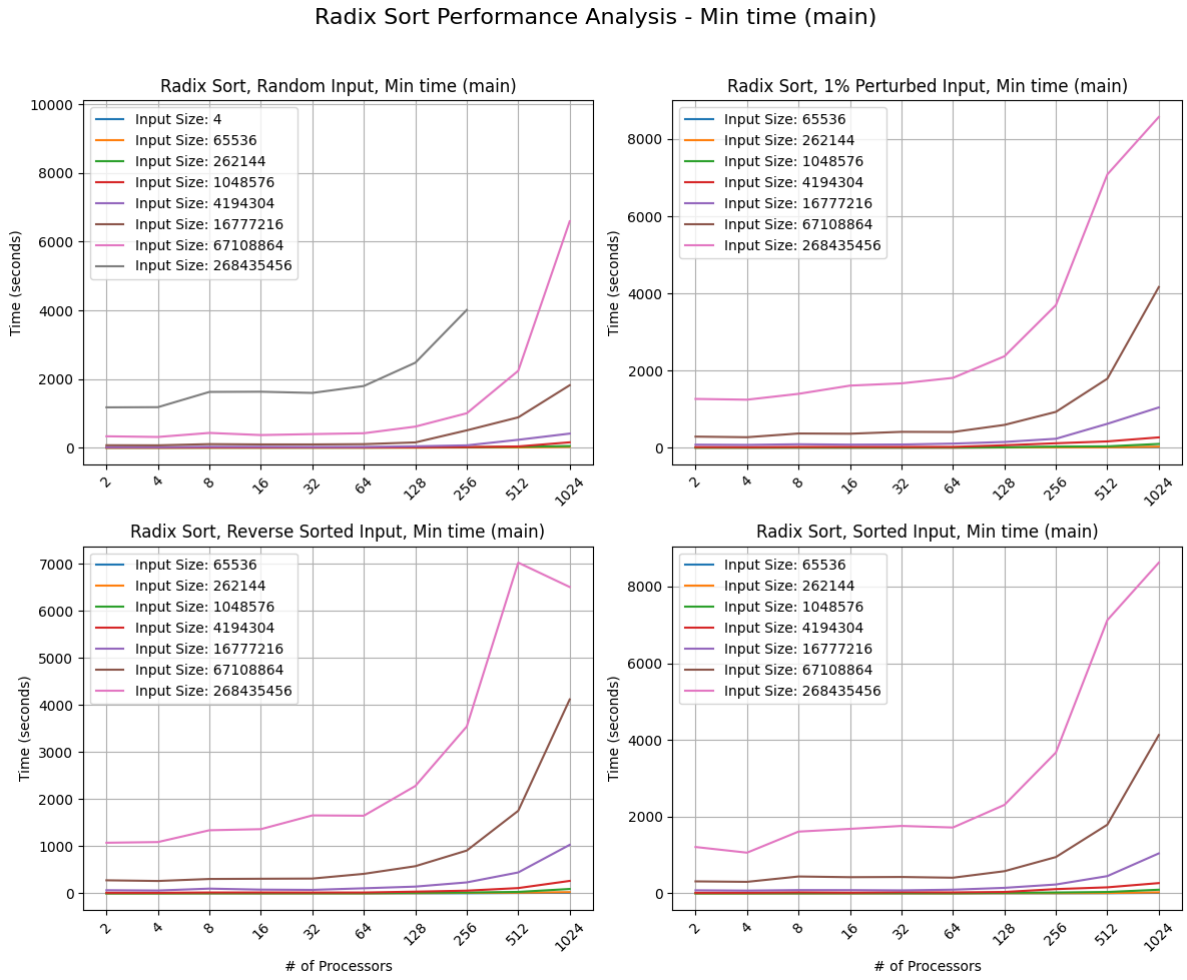

## IV) Radix Sort

- **main:**

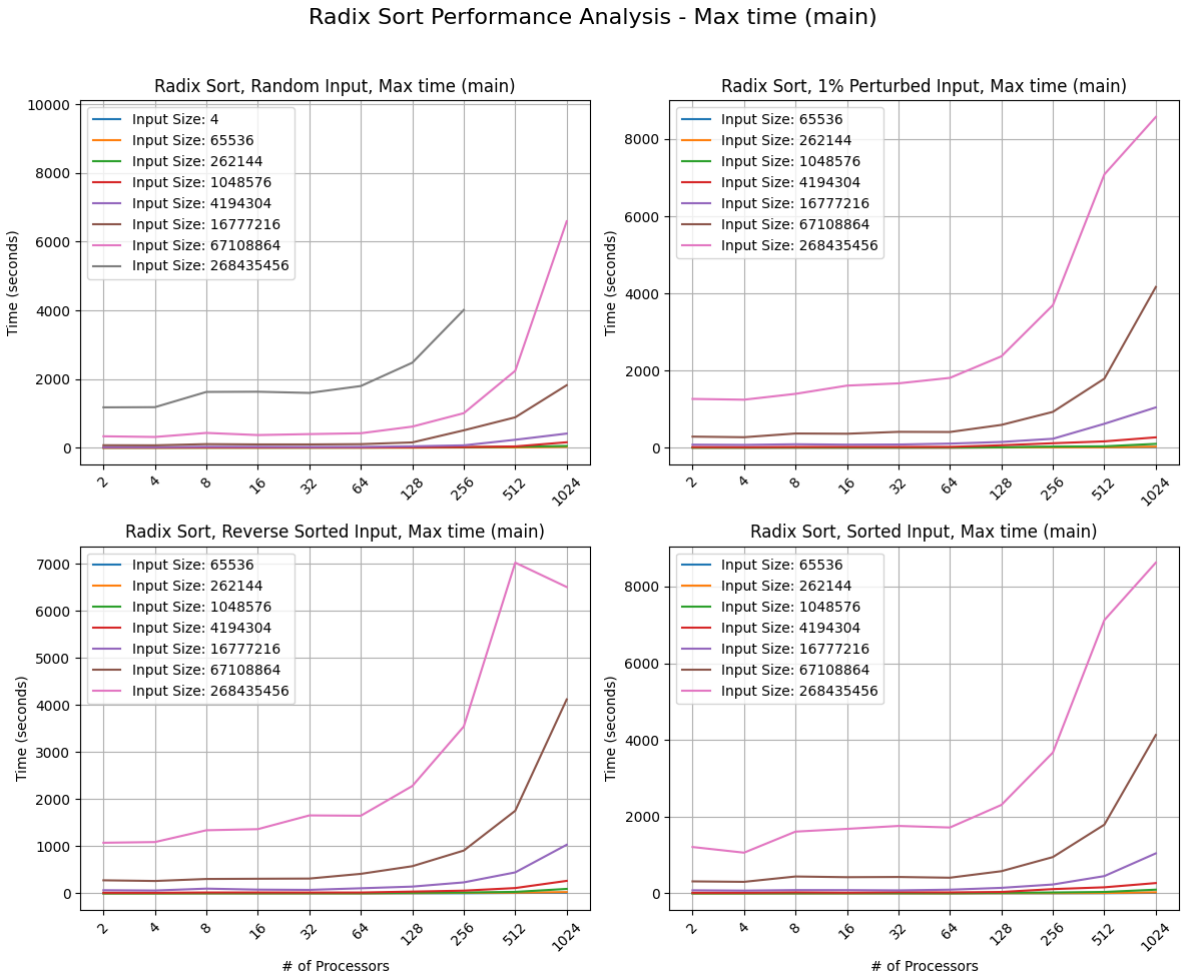
  - Avg time/Rank:



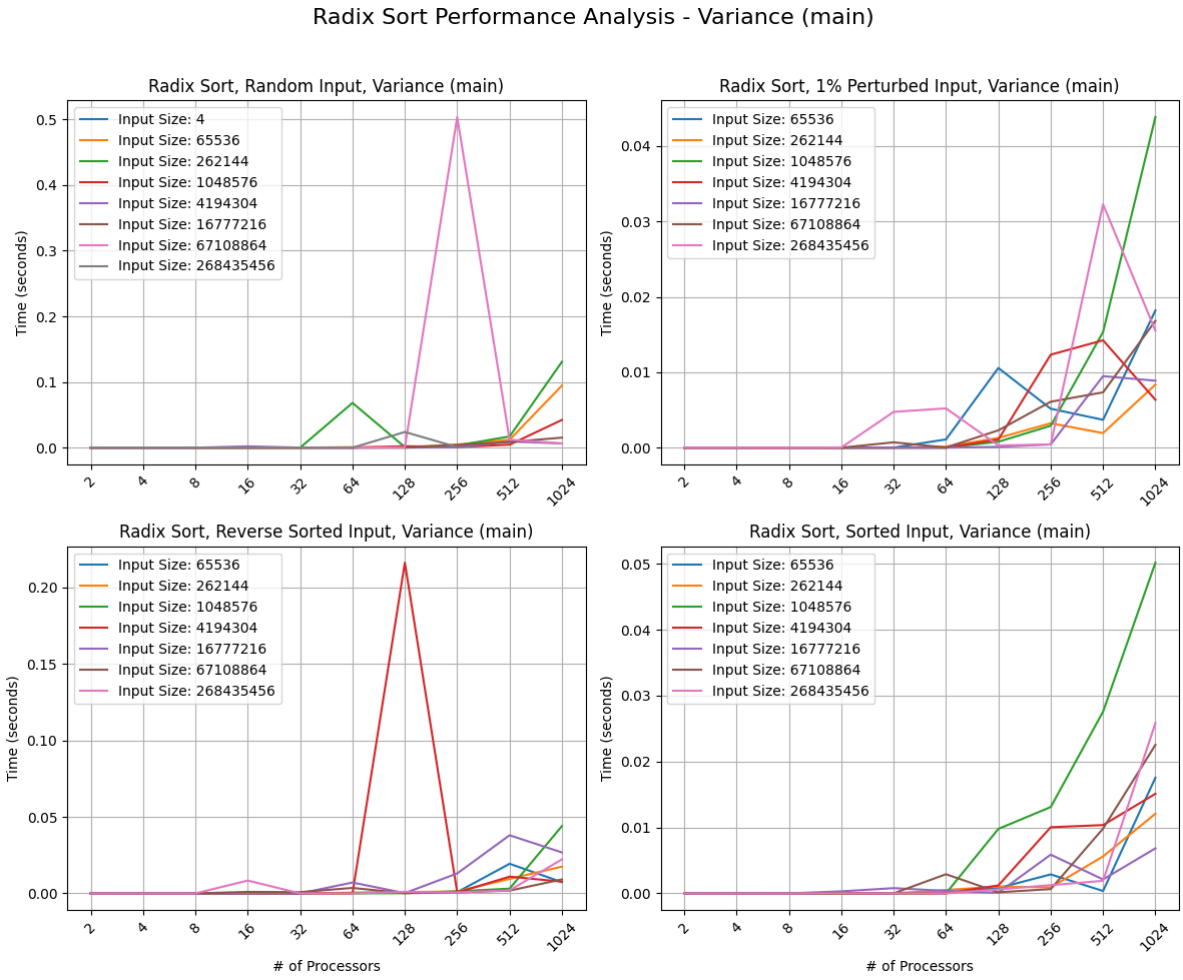
- Min time/Rank:



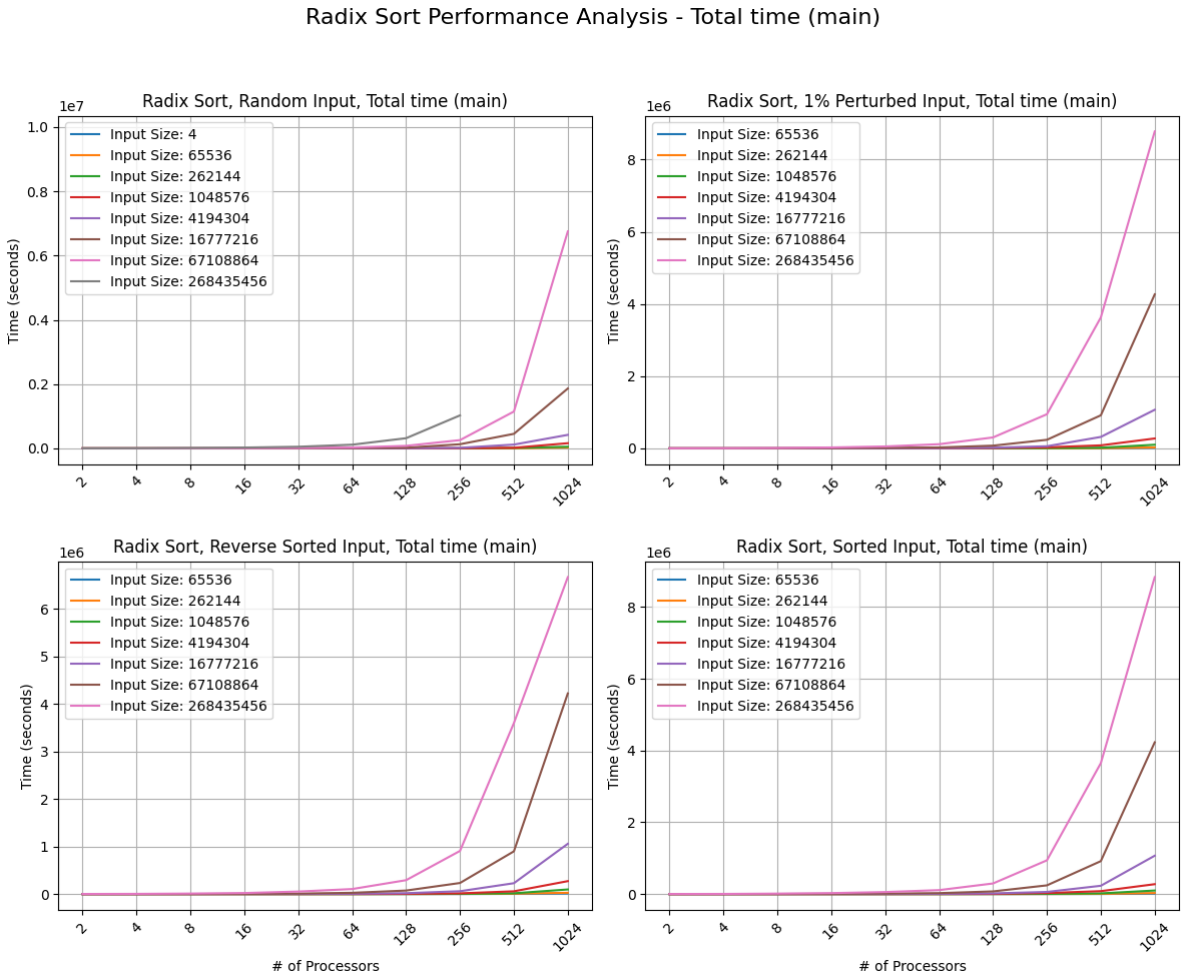
- Max time/Rank:



- Variance time/Rank:



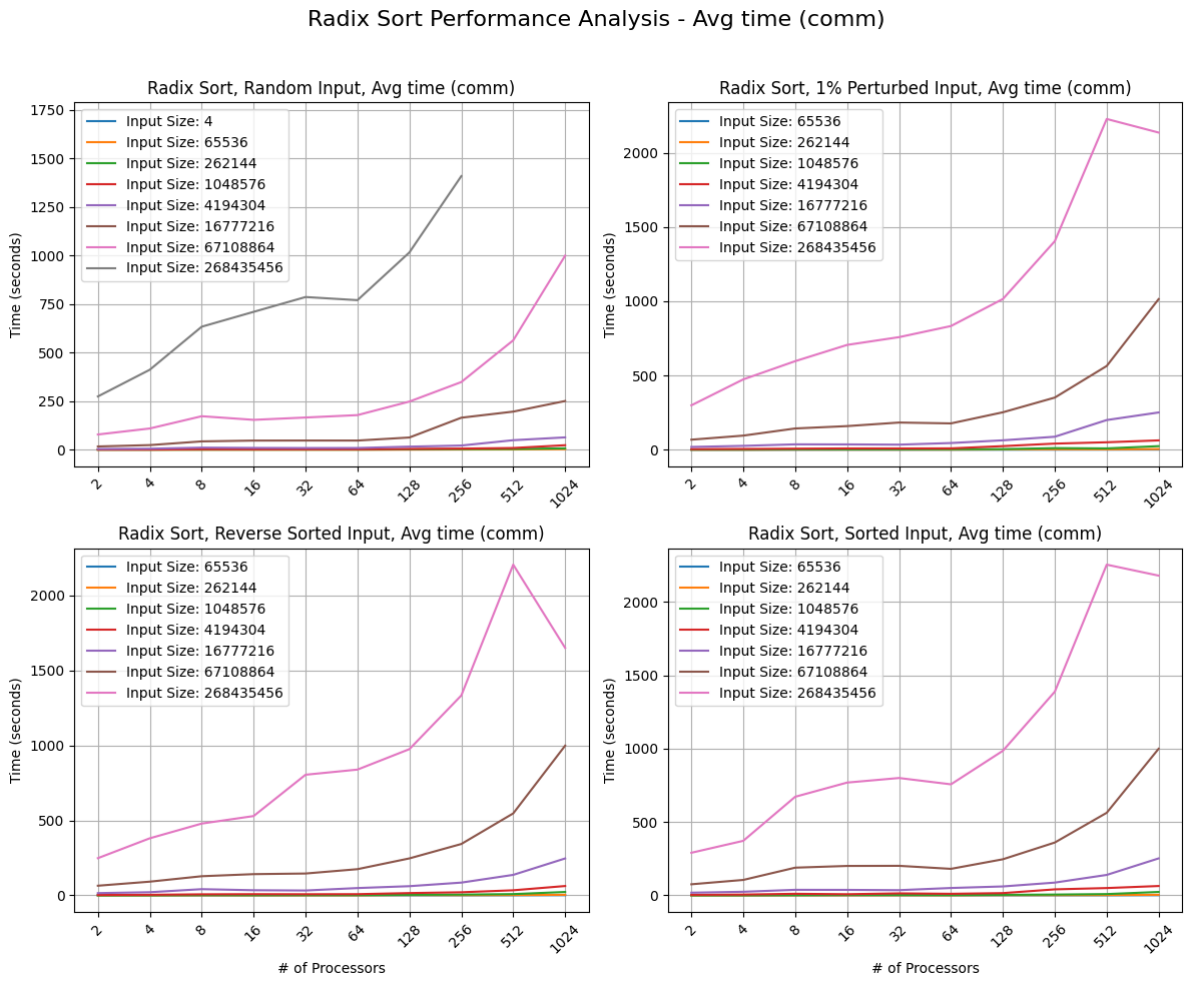
- Total time/Rank:



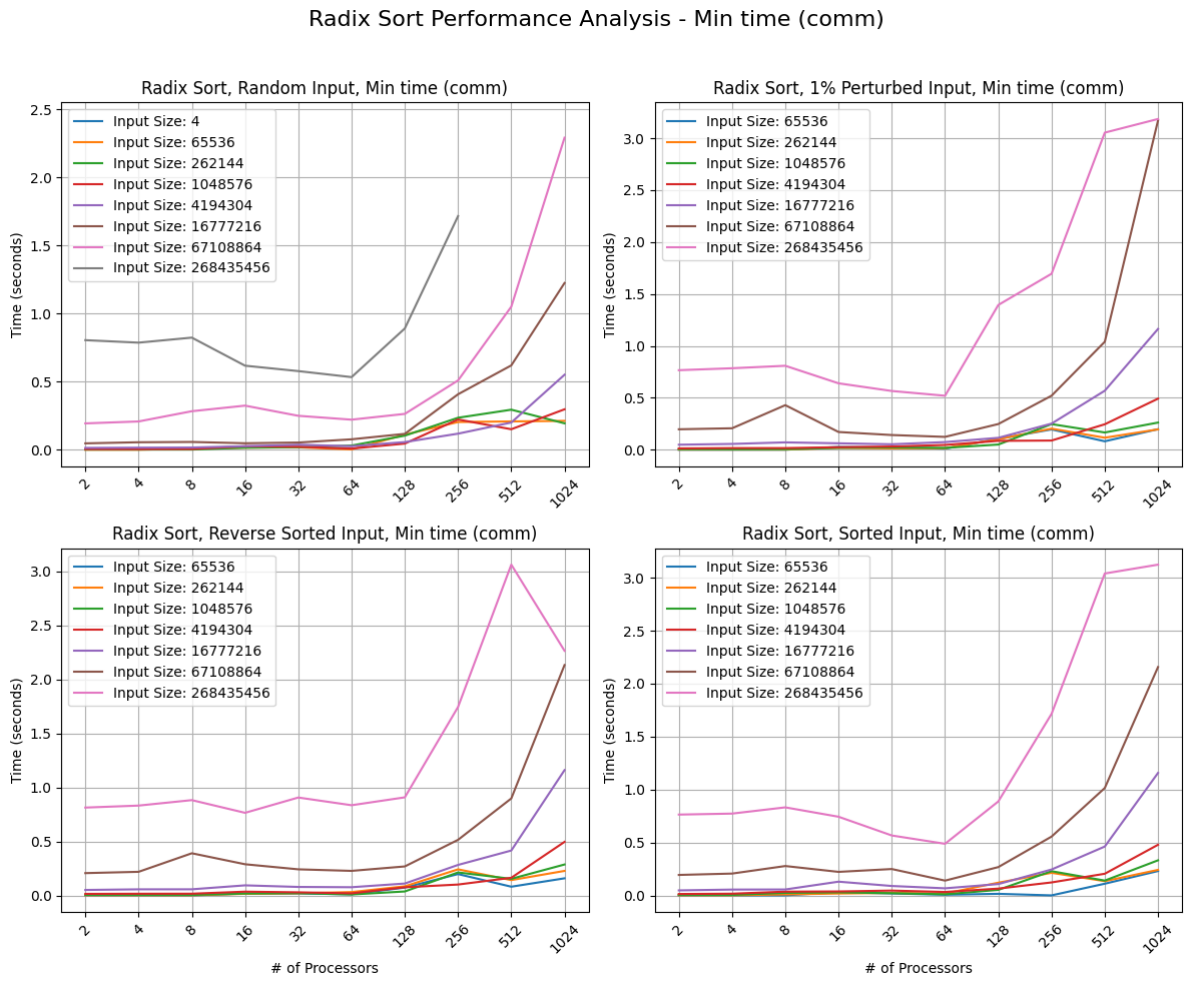

The total times increase as both the input size increases and the proceses increase. This is due to several reasons. The first being that due to more data being present, more time must be spent sending and receiving the data to and from procesess meaning a higher communication overhead. Further, more data means a higher total computation time. In addition, more processes means that data needs to be sent to and from more locations resulting in longer runtimes. These observations are evident in the figures above.

- **comm:**

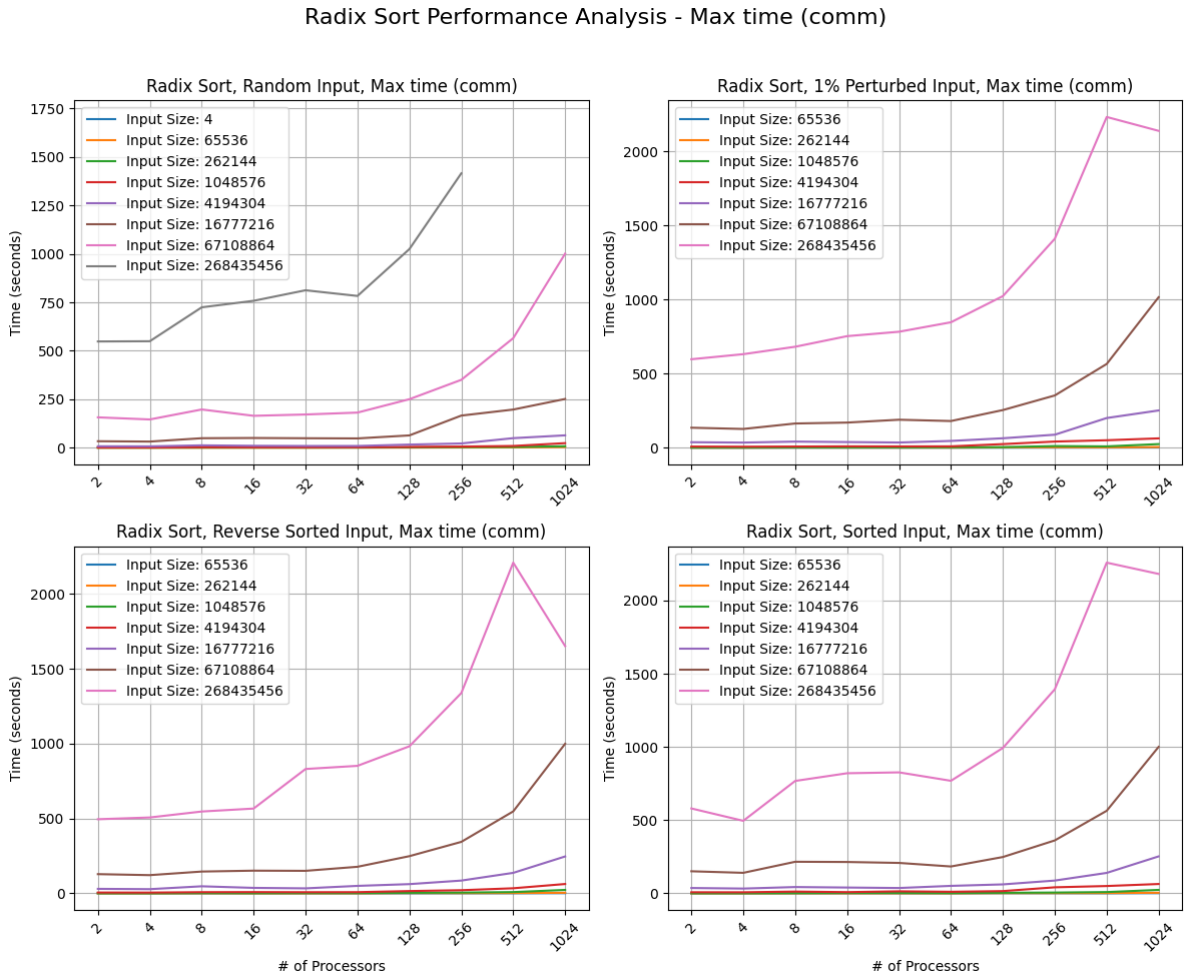
- Avg time/Rank:



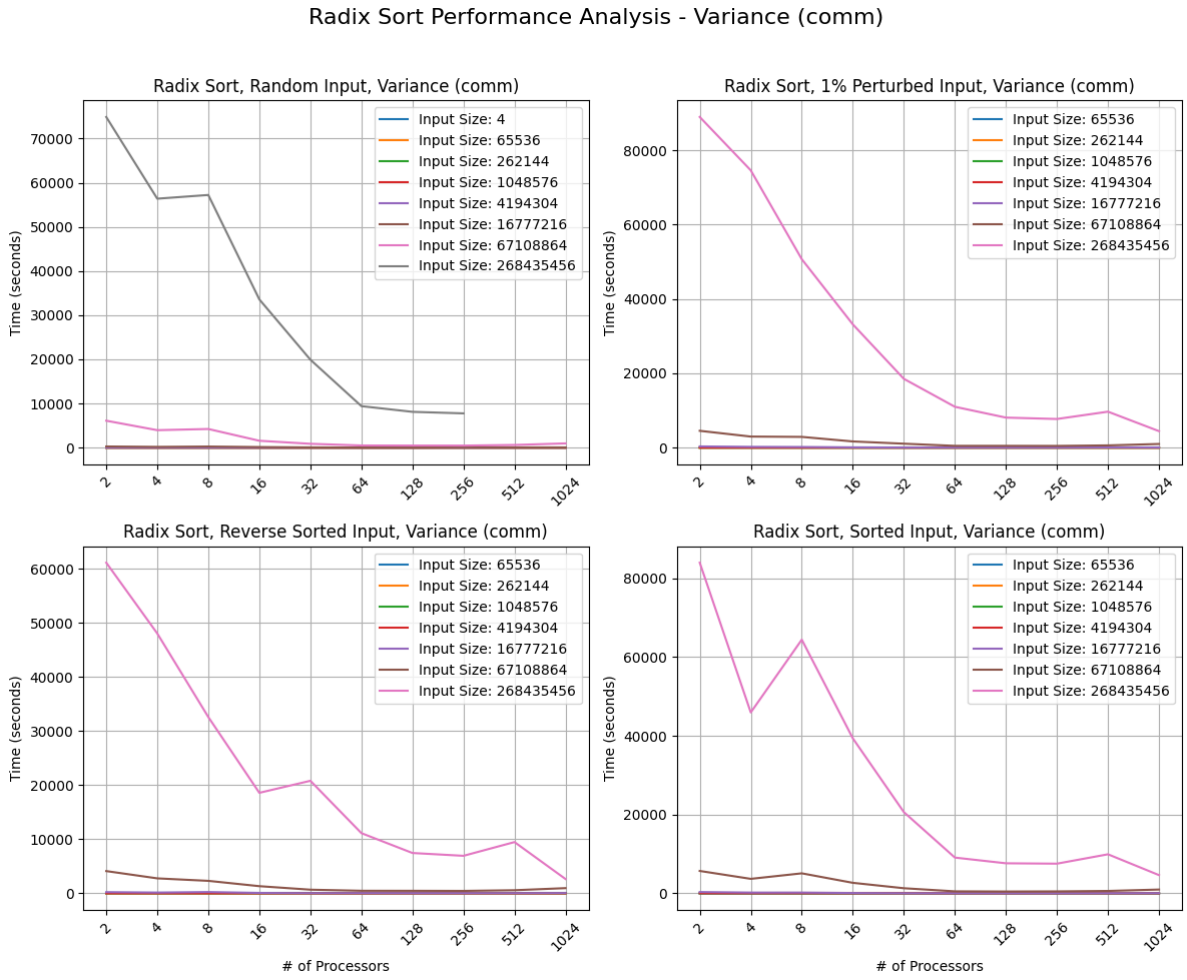
- Min time/Rank:



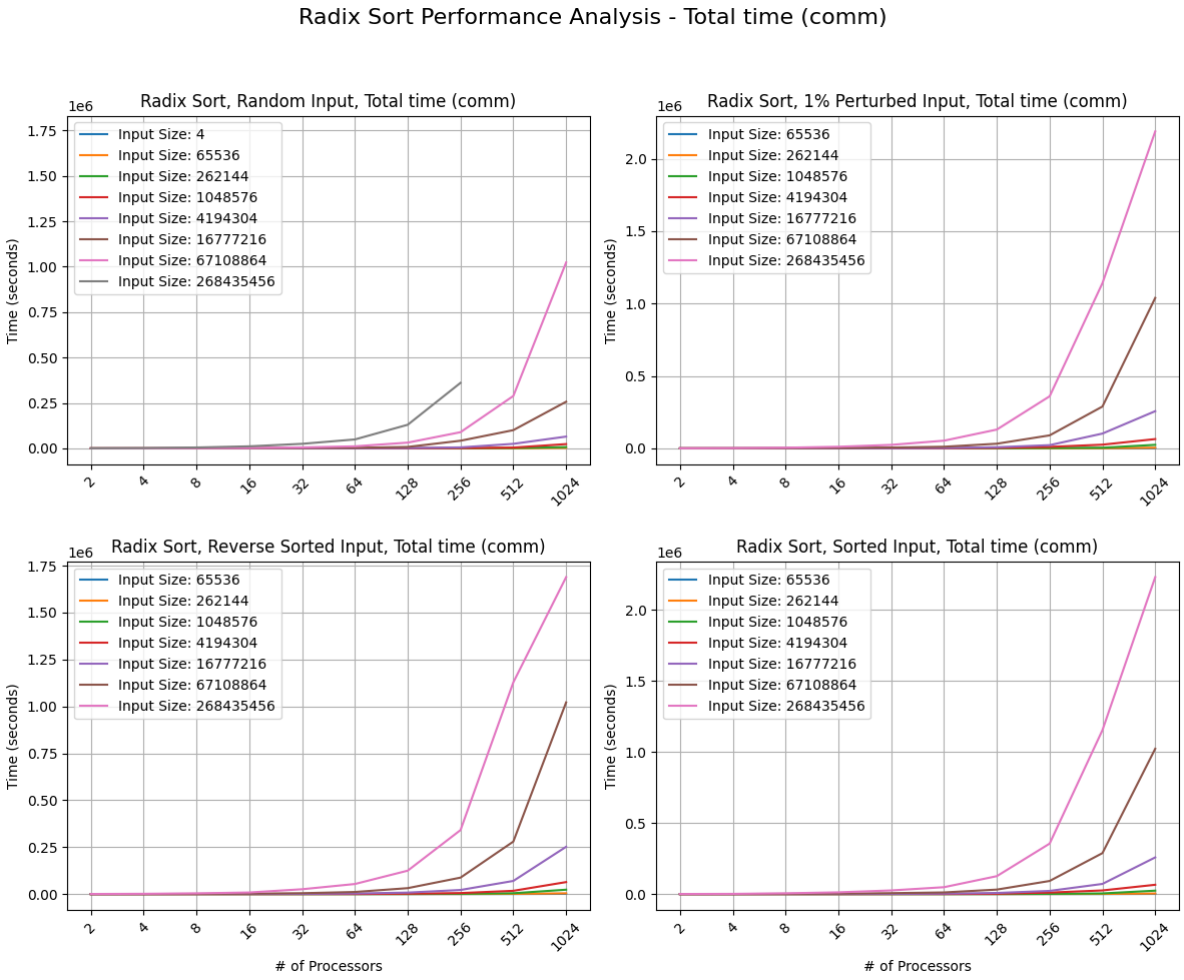
- Max time/Rank:



- Variance time/Rank:



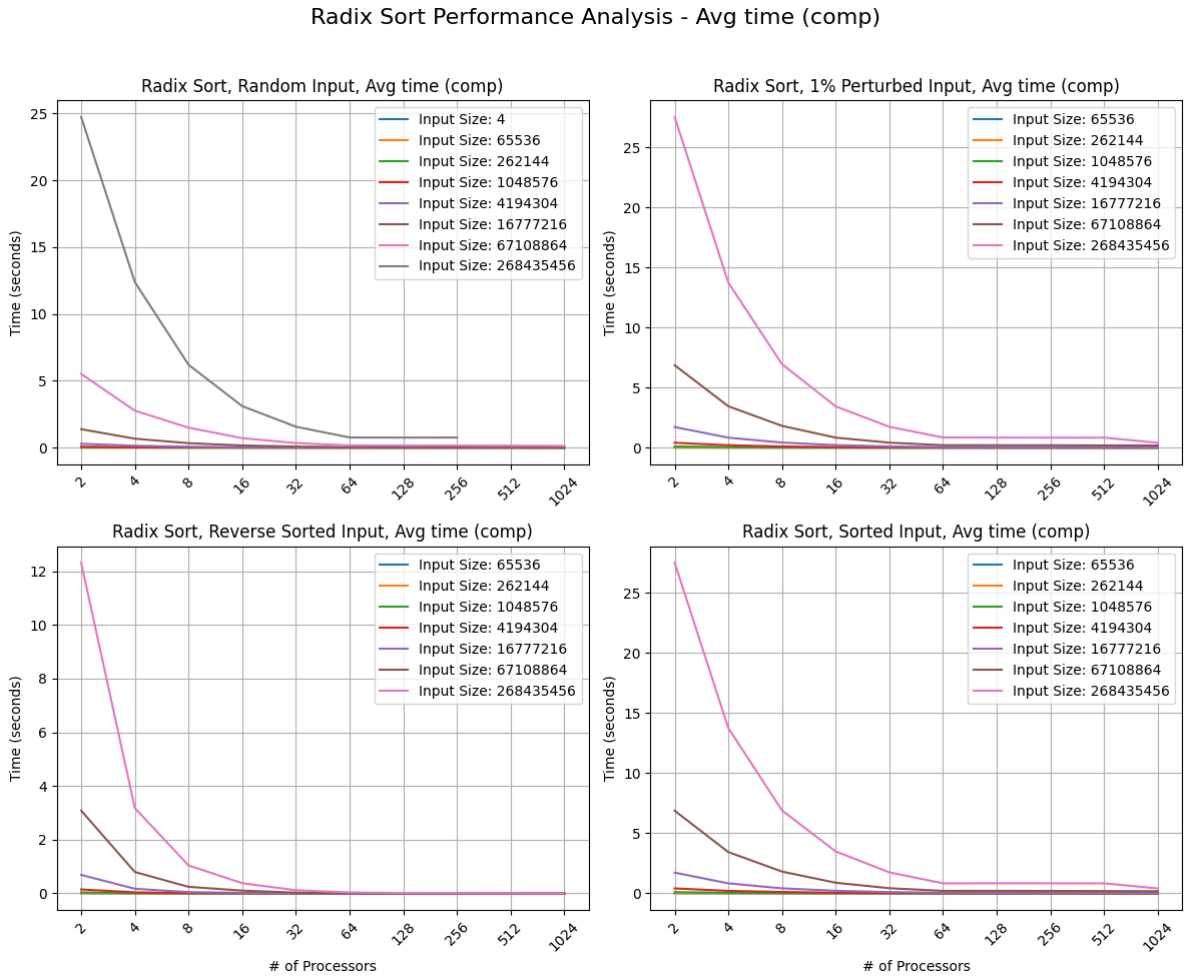
- Total time/Rank:



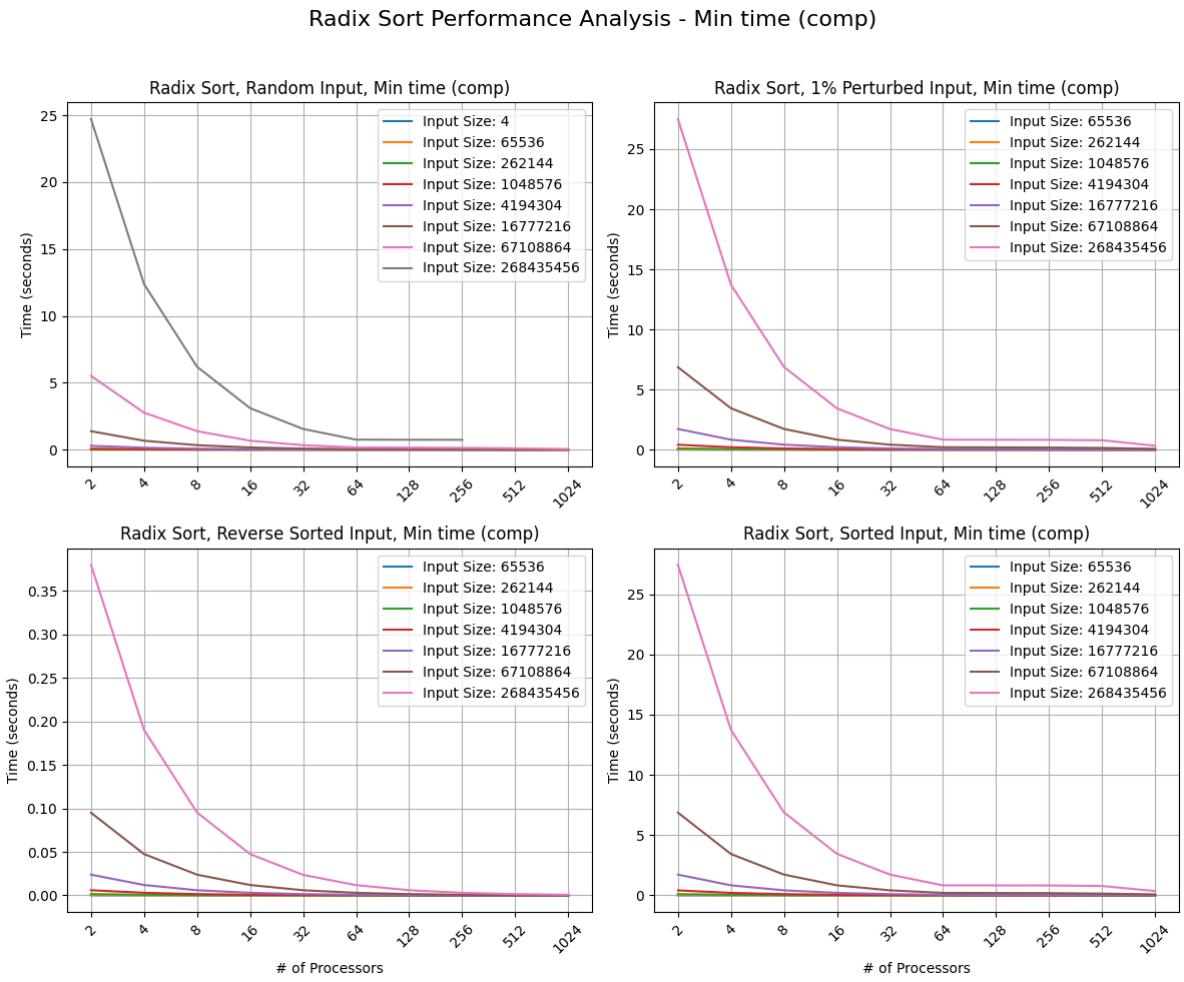

The communication times increase as the input size increases. Further, more processes also cause an increase in time. This is due to several reasons. The first being that due to more data being present, more time must be spent sending and receiving the data to and from procesess. In addition, more processes means that data needs to be sent to and from more locations resulting in longer runtimes as evident in the figures above.

- **comp**

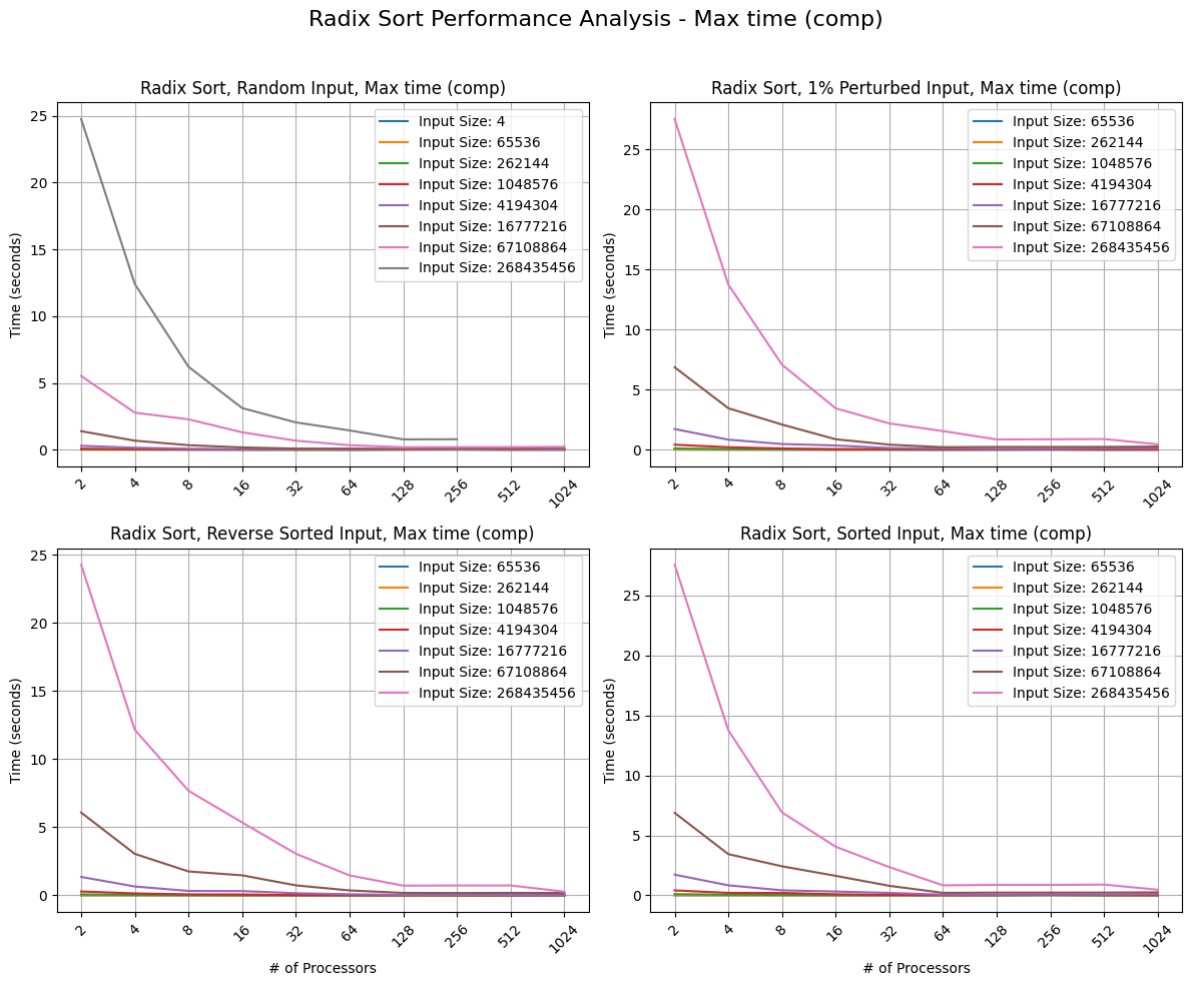
- Avg time/Rank:



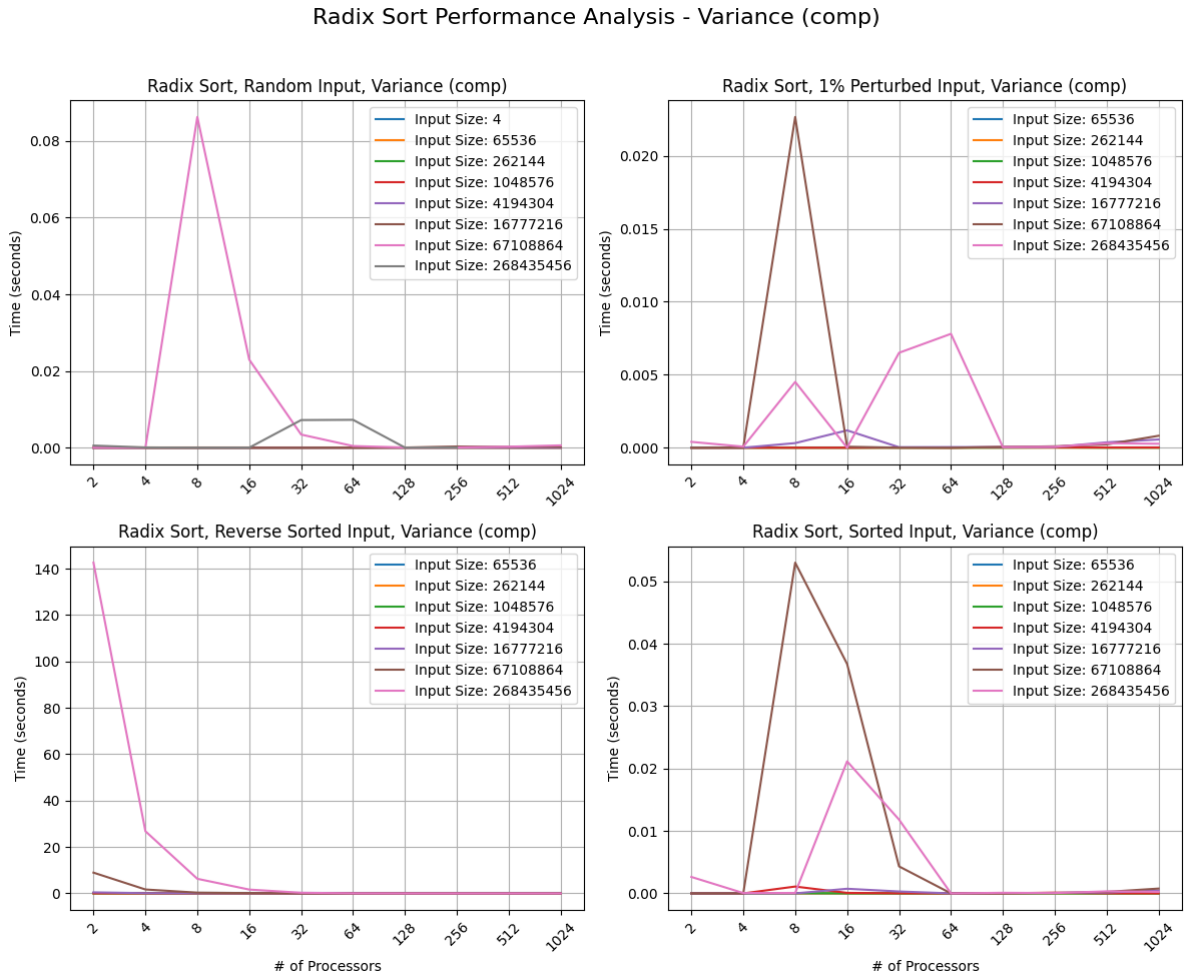
- Min time/Rank:



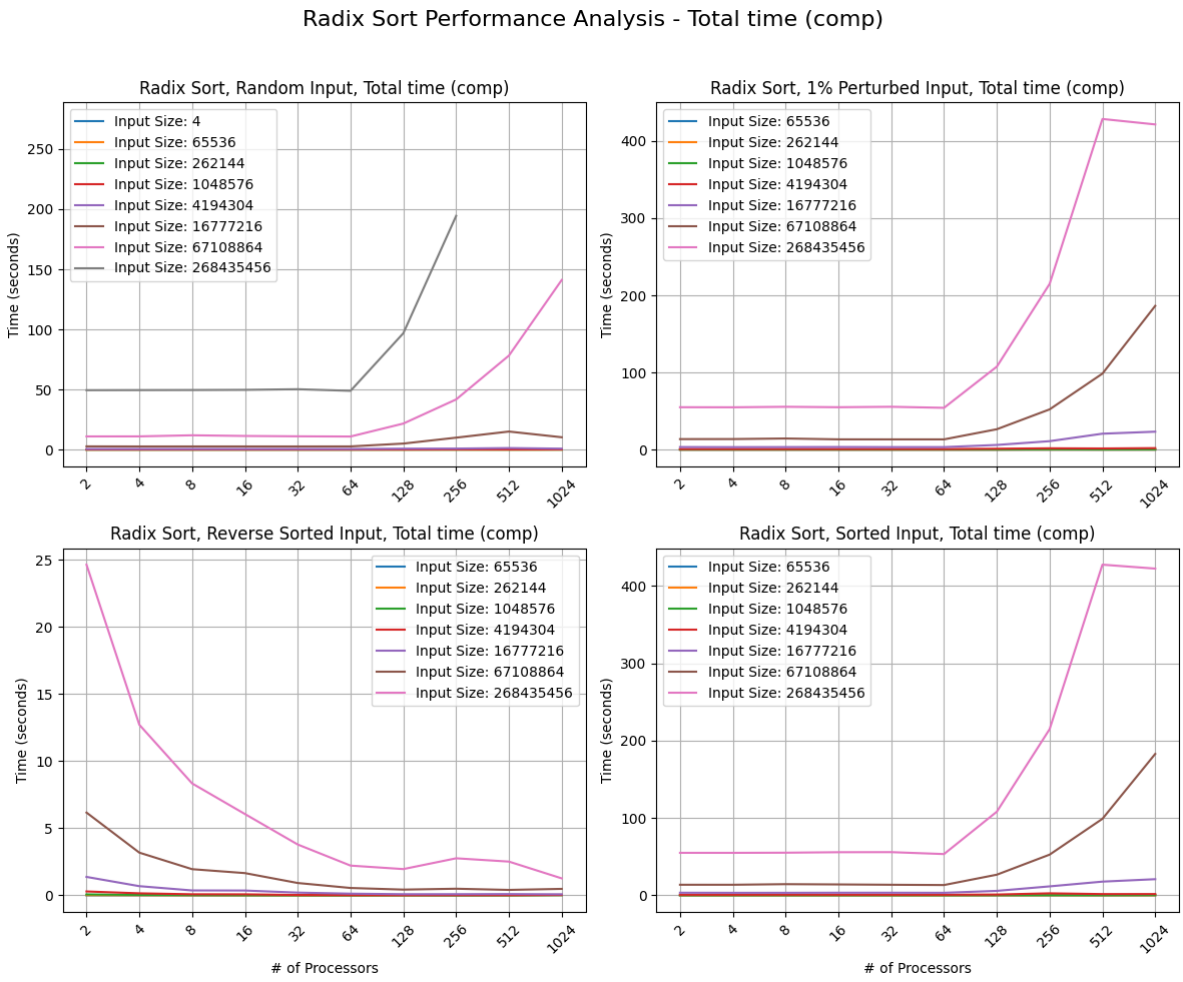
- Max time/Rank:



- Variance time/Rank:

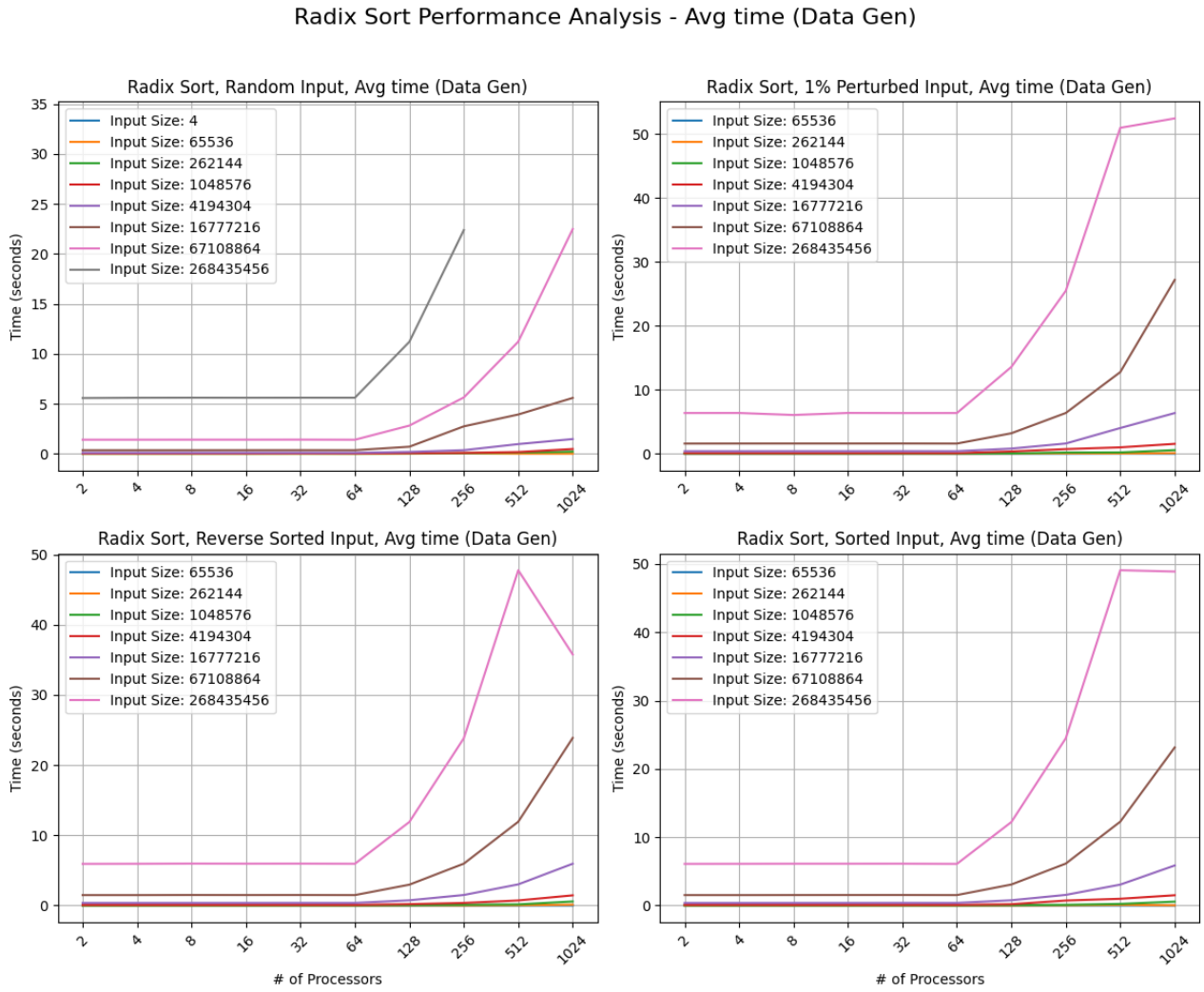


- Total time/Rank:

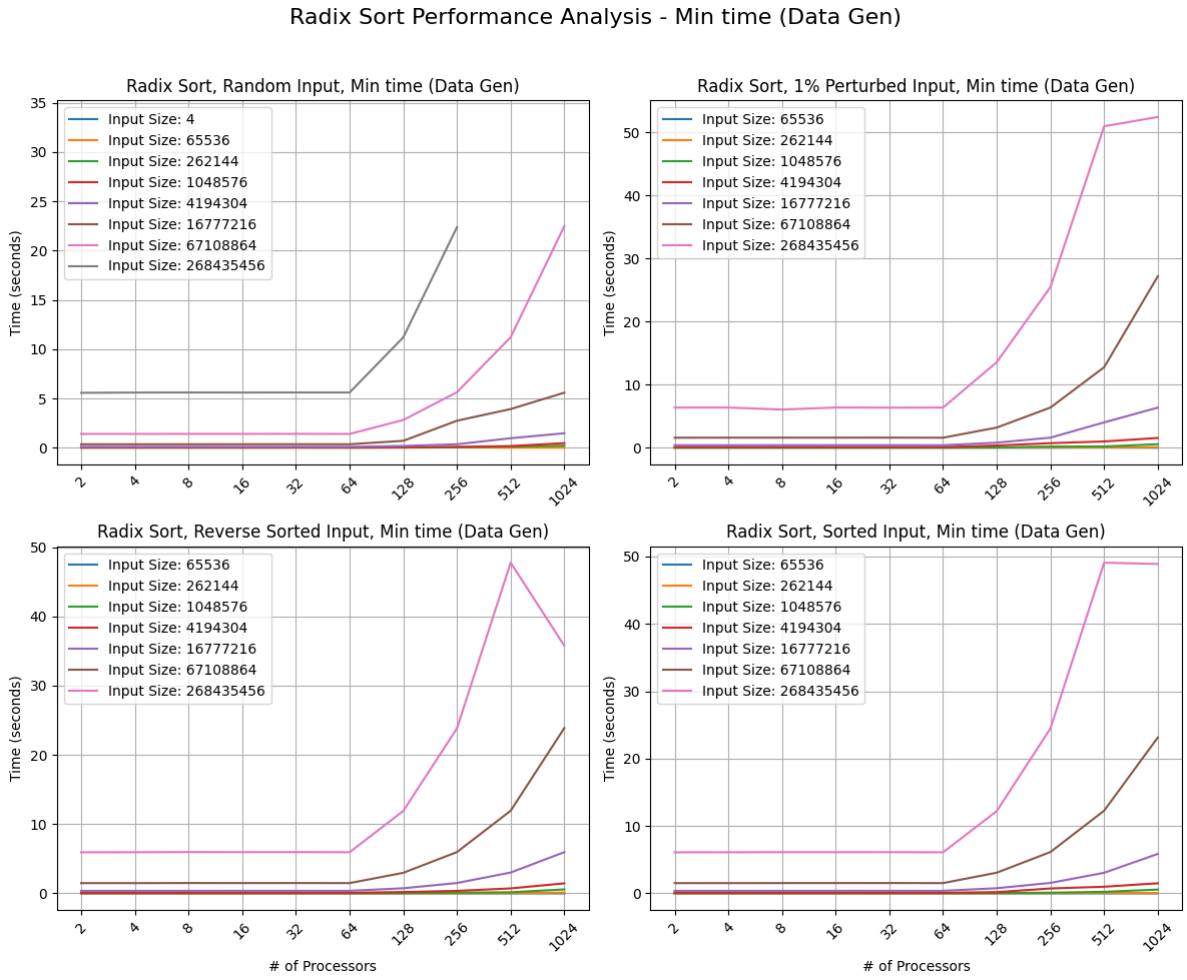



On average the computation time should be decreasing however upon analyzing the total times for the computation this is not necessarily the case. There could be several reasons. One is the fact that our data generation function when provided with a large number also generates numbers that have a large number of digits. Therefore, since the runtime of the radix sort is dependent on the number of digits could end up increasing the computation time. The implementation might also be flawed. I realized that the algorithm has one small portion that is not parallelized, which needs to be changed before our final presentation. The high variance can be explained by the fact that different generation can lead to much different sets of data leading to differing performance rates.

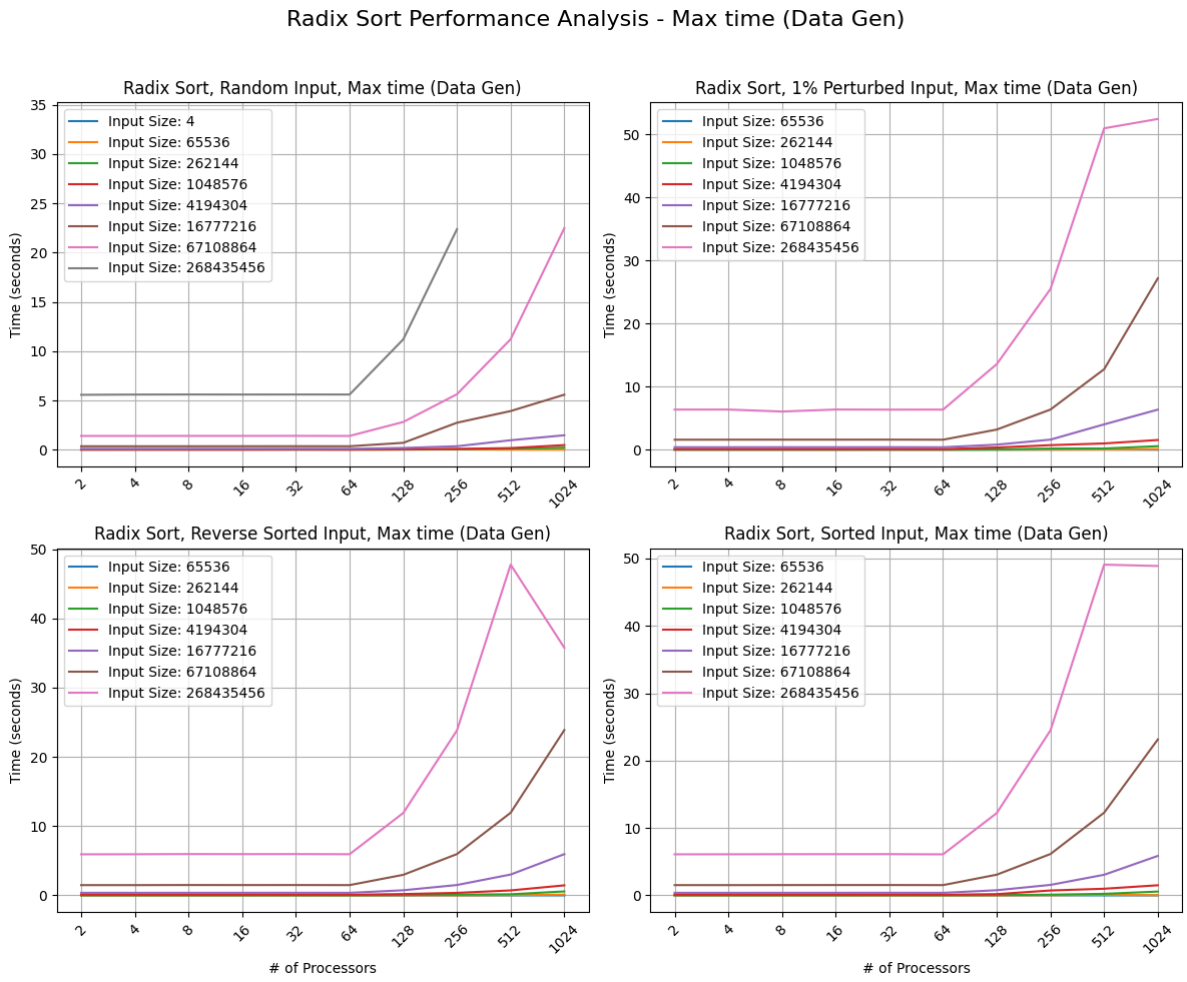
- **Data Generation**


Radix Sort Performance Analysis - Avg time (Data Gen)
Radix Sort, Random Input, Avg time (Data Gen)
Radix Sort, 1% Perturbed Input, Avg time (Data Gen)
Radix Sort, Reverse Sorted Input, Avg time (Data Gen)
Radix Sort, Sorted Input, Avg time (Data Gen)
Time (seconds)
# of Processors
Input Size: 4
Input Size: 65536
Input Size: 262144
Input Size: 1048576
Input Size: 4194304
Input Size: 16777216
Input Size: 67108864
Input Size: 268435456

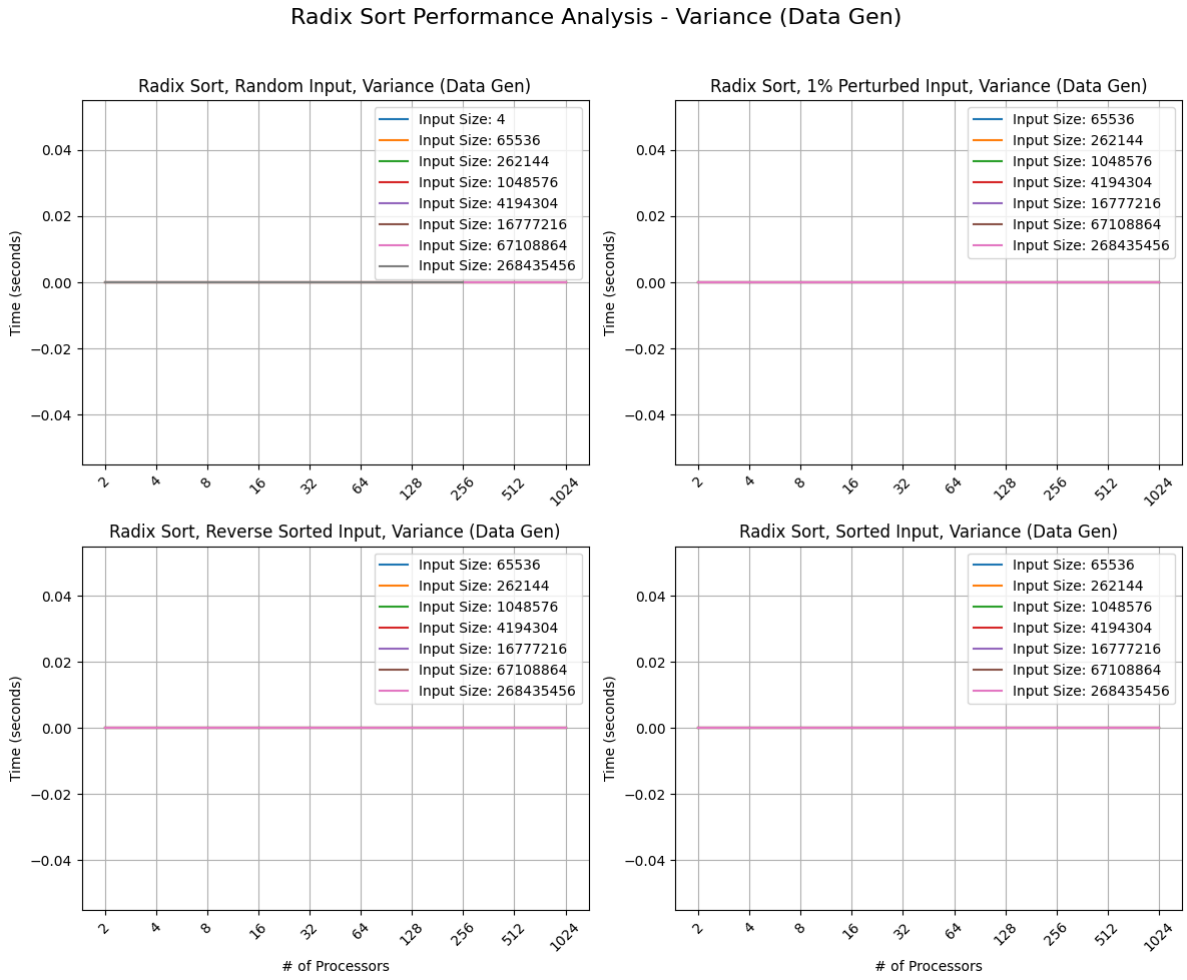
- Min time/Rank:



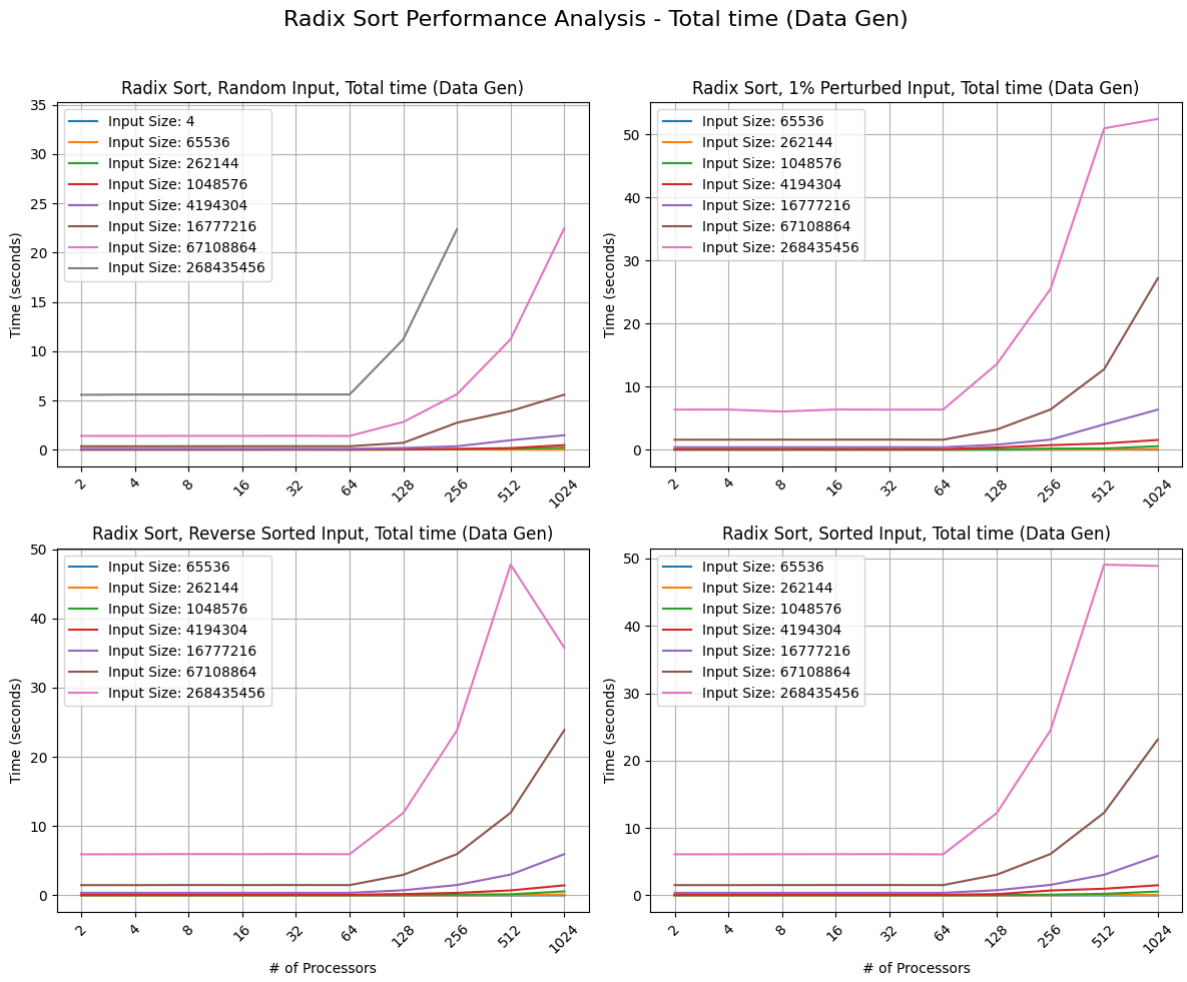
- Max time/Rank:



- Variance time/Rank:

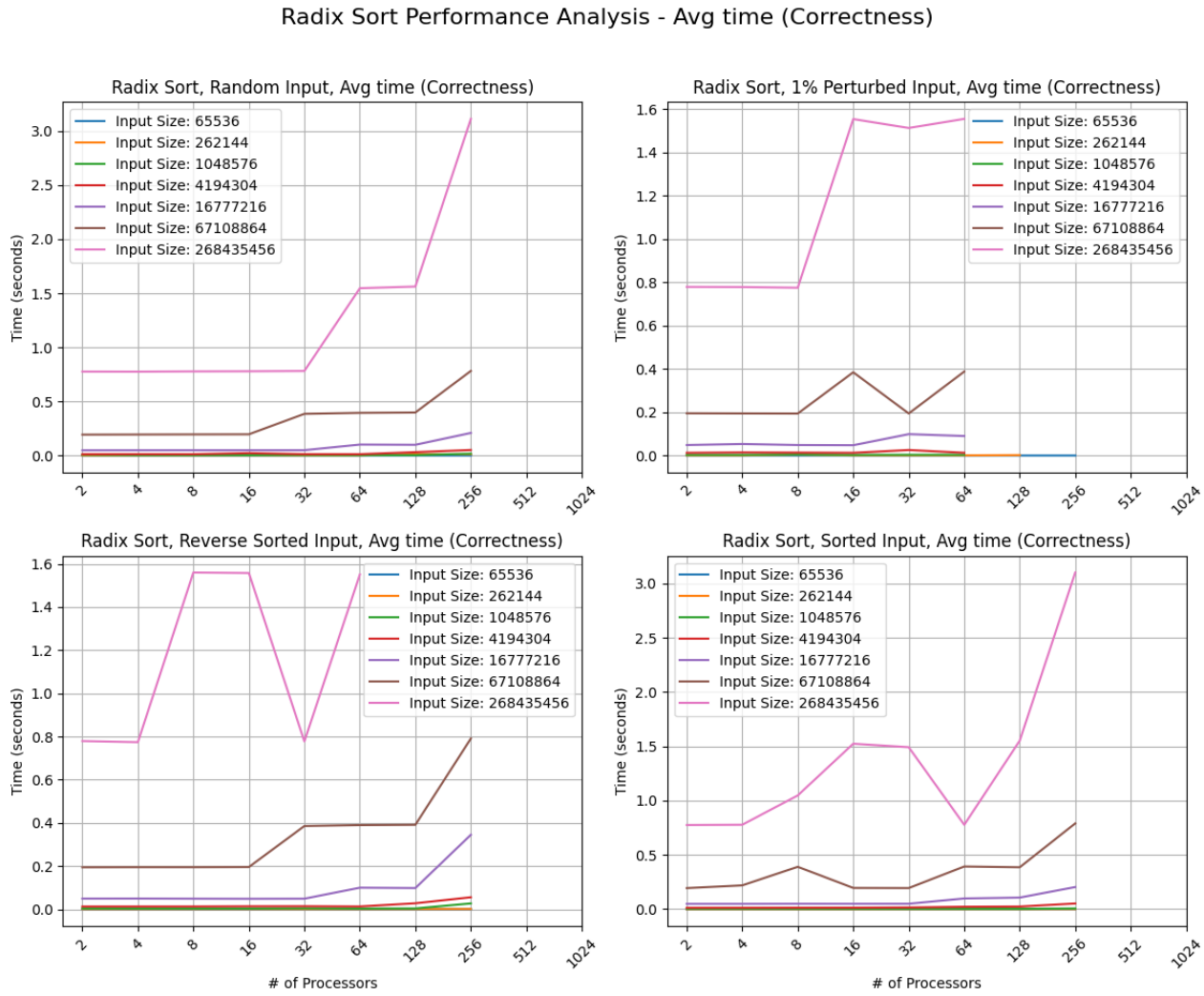


- Total time/Rank:

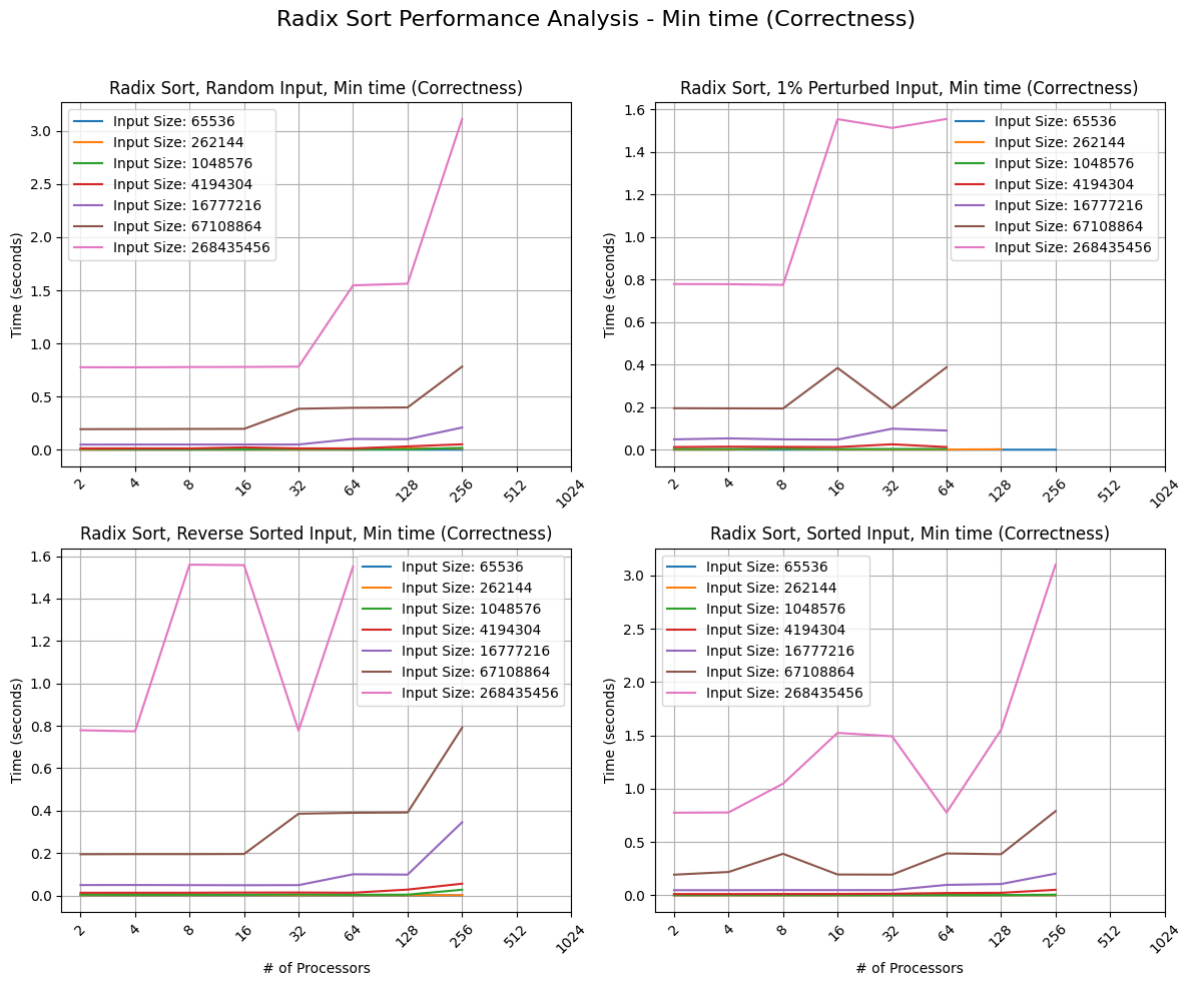



As the total amount of data increases it will take longer to generate the data needed for our sort this increased time can be seen above. There is little variance because the same function to generate our data is used on every iteration.

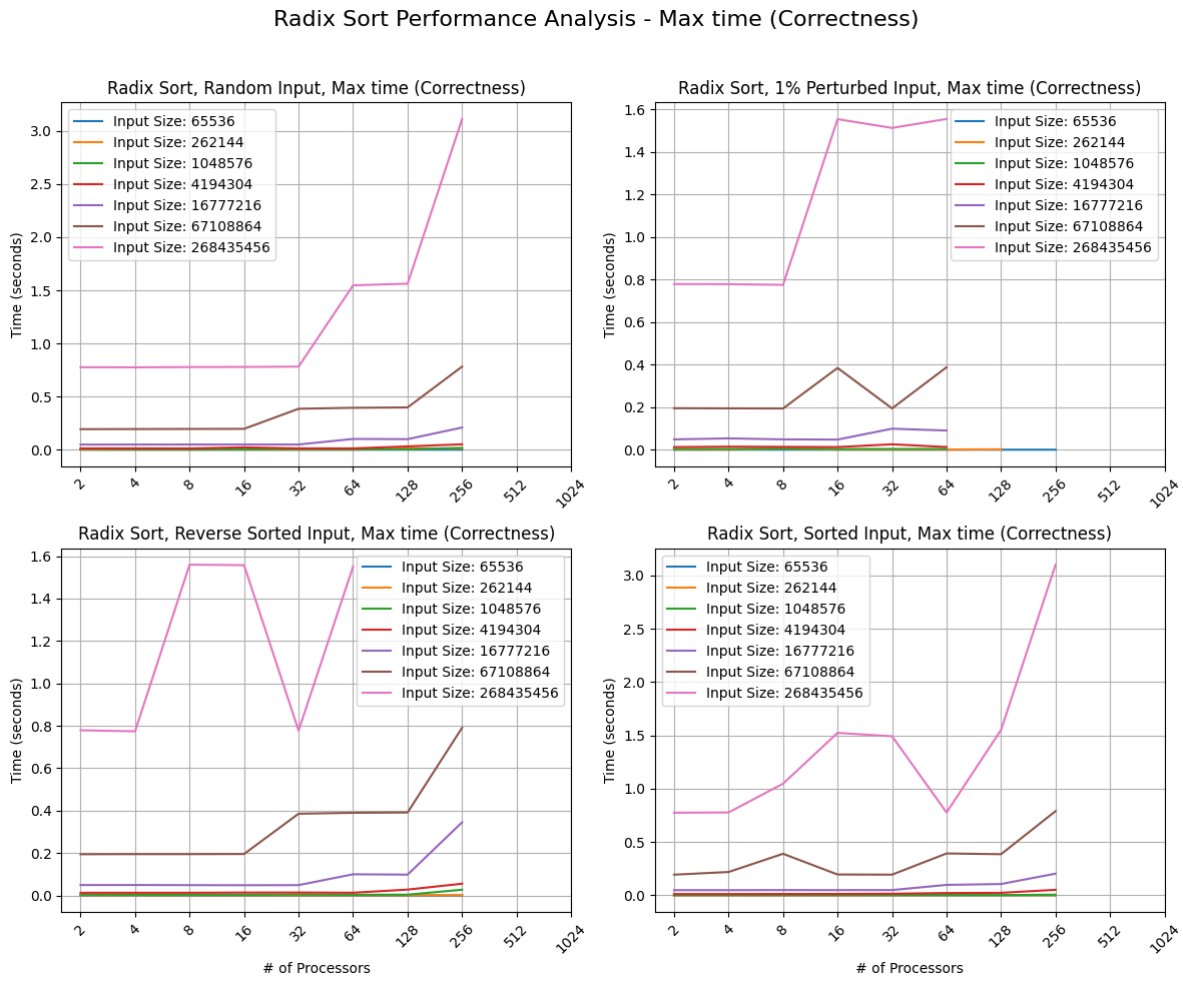
- **Correctness Check:**


Radix Sort Performance Analysis - Avg time (Correctness)
Radix Sort, Random Input, Avg time (Correctness)
Radix Sort, 1% Perturbed Input, Avg time (Correctness)
Radix Sort, Reverse Sorted Input, Avg time (Correctness)
Radix Sort, Sorted Input, Avg time (Correctness)
Time (seconds)
# of Processors
Input Size: 65536
Input Size: 262144
Input Size: 1048576
Input Size: 4194304
Input Size: 16777216
Input Size: 67108864
Input Size: 268435456

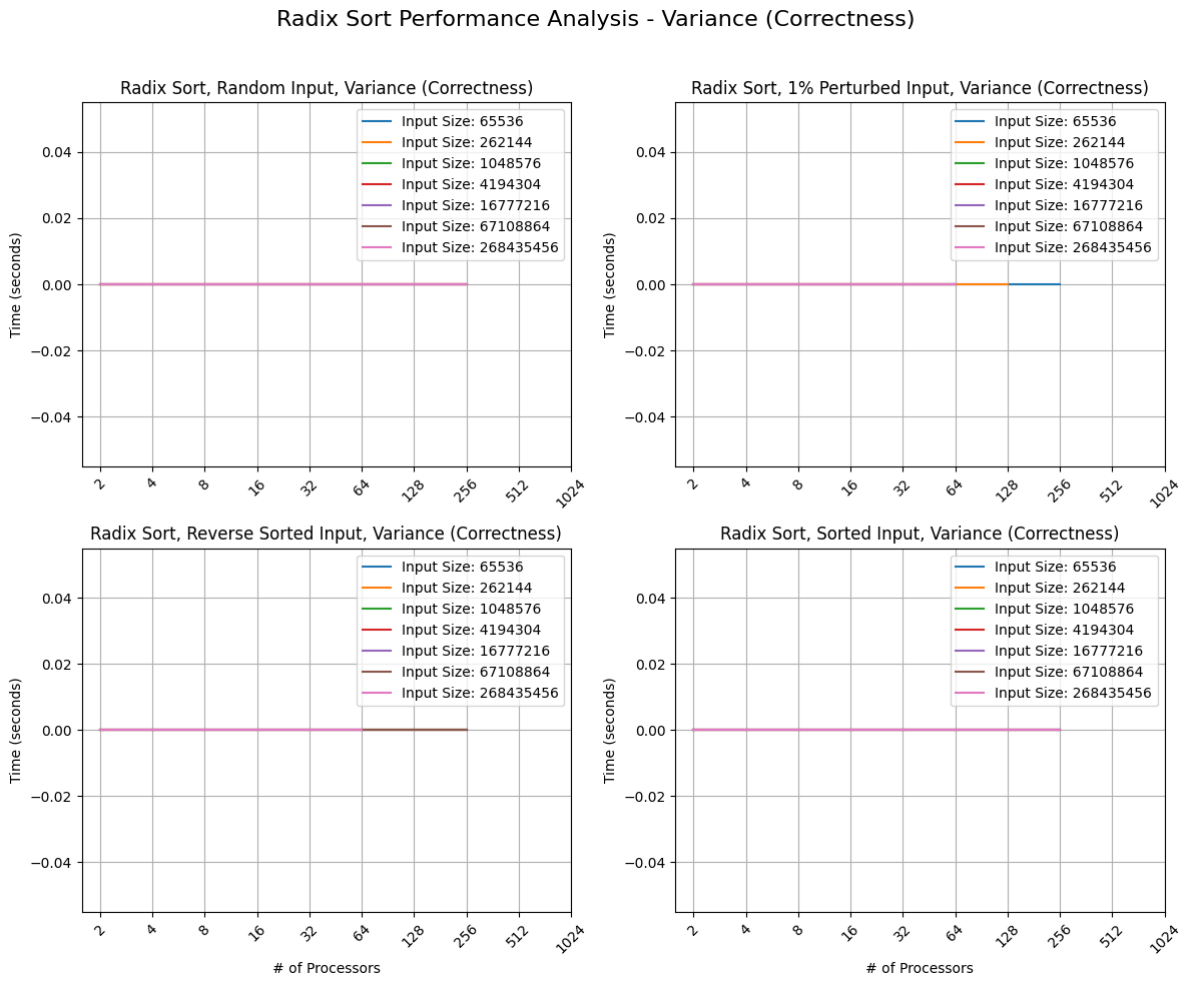
- Min time/Rank:



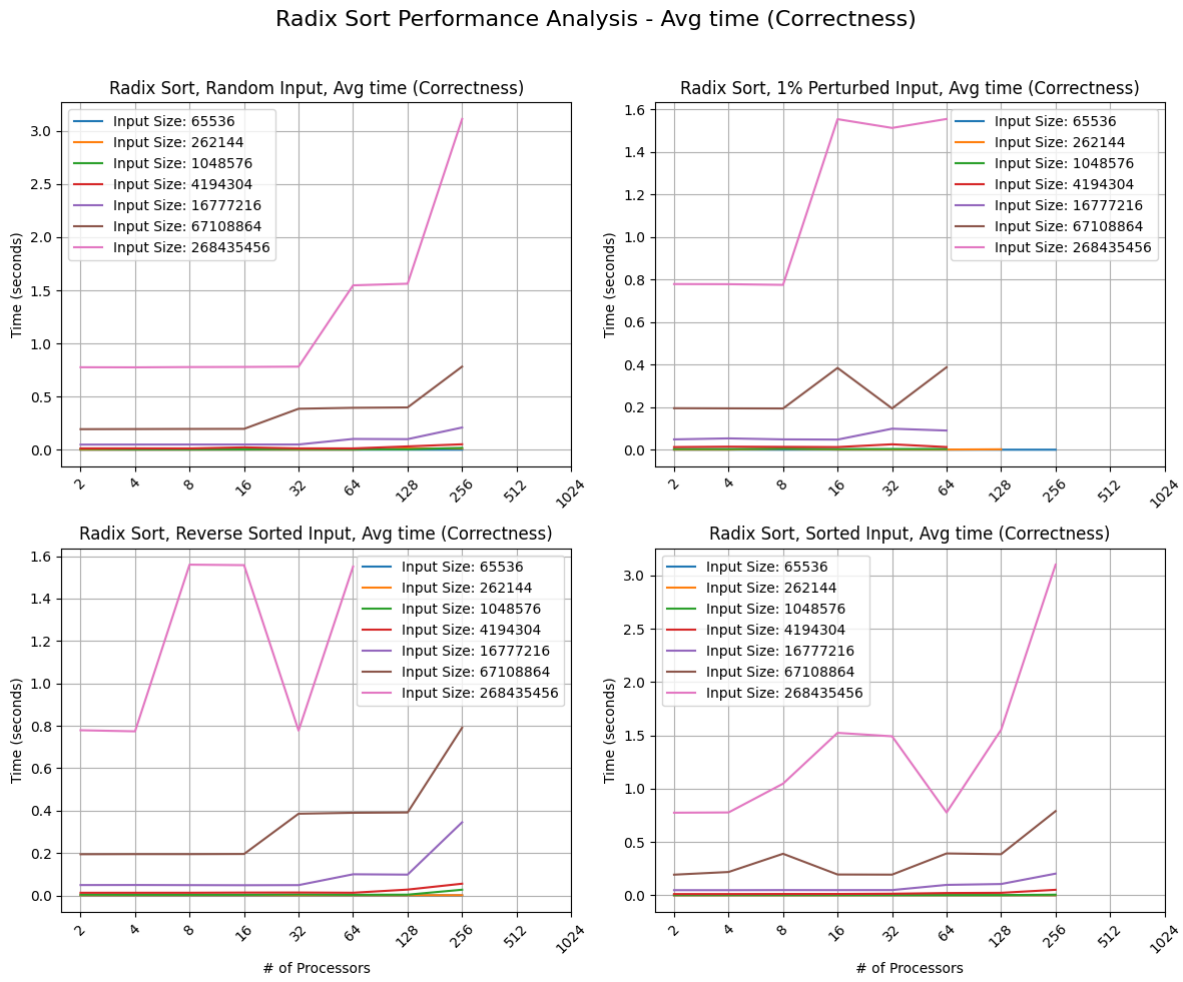
- Max time/Rank:



- Variance time/Rank:



- Total time/Rank:




As the total amount of data increases it will take longer to check if our output is correctly sorted as supported by the graphs above. There is little variance because the same function to check that our data sorted is used on every iteration. I was not able to obtain every sample size and process count for the data check as Grace's scheudler was over crowded leading to queues so long that no jobs could be executed. I was assured that this would not negatively affect us.

# 5. 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.

# 6. 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 seperated by algorithm/implementation.

- Jupyter notebook: The Jupyter notebook(s) used to generate the plots for the report.
- Report.md