CSCE 435 Group project

1. Group members:

1. Riley Szecsy
2. Vincent Lobello
3. Jonathan Kutsch
4. Nebiyou Ersabo
   Group is communicating over groupme.

2. Project topic

Comparing the performance of comparision based sorting algorithms. We will be scaling the GPU settings and change the number of threads and proccessors of the CPU based on problem size. We will be testing these on sorted, random, and reverse sorted inputs. For Radix sort we will be comparing it directly to sample sort as we can only test integers.

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

- Sample Sort (MPI + CUDA)
  - MPI on each core

- Pesudocode:

```
Where p = # processors and k = oversampling factor:
1. Sample p (* k) elements and sort them
2. Share these samples with every processor
(MPI_Allgather)
3. Each p select p-1 pivots aka splitters. These are
the same across p’s.
a. Each b-th pair of splitters denotes a “bucket” that
will be sent to the b-th processor.
4. Re-arrange local data into the buckets described
by the pivots.
5. Send the b-th bucket to the b-th processor
(MPI_Alltoall[v])
6. Combine buckets and sort local data.
```

- Mergesort (MPI + CUDA)
  - MPI on each core
  - Pesudocode:

```
function parallel_merge_sort(arr):
  if length(arr) <= 1:
      return arr

  middle = length(arr) / 2
  left_half = arr[0:middle]
  right_half = arr[middle:]

  left_sorted = parallel_merge_sort(left_half)
  right_sorted = parallel_merge_sort(right_half)

  result = parallel_merge(left_sorted, right_sorted)

  return result

function parallel_merge(left, right):
  result = []
  left_index = 0
  right_index = 0
  while left_index < length(left) and right_index < length(right):
      if left[left_index] < right[right_index]:
          result.append(left[left_index])
          left_index++
      else:
          result.append(right[right_index])
          right_index++
  while left_index < length(left):
      result.append(left[left_index])
      left_index++
  while right_index < length(right):
      result.append(right[right_index])
      right_index++
  return result
```

- Odd-Even Transposition Sort (MPI + CUDA)
  - MPI on each core
  - Pesudocode:

```
procedure ODD-EVEN PAR(n)
begin
  id := proccees's label
  for i := 1 to n do
  begin
    if i is odd then
      if id is odd then
       compare-exchange min (id+1);
      else
        compare-exchange max(id-1);
      if i is even then
        if id is even then
          compare-exchange min(id+1);
        else
          compare-exhange max(id-1);
    end for
 end ODD-EVEN PAR
```

- Radix Sort (MPI + CUDA)
  - MPI on each core
  - Pesudocode

```
procedure RadixSort(arr, n)
begin
    id := process's label
    for exp := 1 to maximum_digit_position do
    begin
        if exp is odd then
            if id is odd then
                CompareExchangeMin(id + 1)
            else
                CompareExchangeMax(id - 1)
        end if
        if exp is even then
            if id is even then
                CompareExchangeMin(id + 1)
            else
                CompareExchangeMax(id - 1)
        end if
        Call CountSort(arr, n, exp)
    end for
end RadixSort

procedure CountSort(arr, n, exp)
begin
    Create an output array of size n
    Create a count array of size 10 and initialize to 0
    for i := 0 to n - 1 do
        count[(arr[i] / exp) % 10]++
    for i := 1 to 9 do
        count[i] += count[i - 1]
    for i := n - 1 down to 0 do
        output[count[(arr[i] / exp) % 10] - 1] := arr[i]
        Decrement count[(arr[i] / exp) % 10]
    Copy the output array to arr
end CountSort
```

2b. Pseudocode for each parallel algorithm

- For MPI programs, include MPI calls you will use to coordinate between processes
- For CUDA programs, indicate which computation will be performed in a CUDA kernel, and where you will transfer data to/from GPU

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

- Random inputs
- Strong scaling
- Weak scaling

3. Project implementation

- Sample Sort:
    - MPI - MPI_Init(), MPI_BCast(), MPI_Scatter(), MPI_Gather(), MPI_Finalize()

  - Cuda - I beleive I finished the implementation but Grace has been running slow and will not run my jobs
- Mergesort:
  - MPI - MPI_Init(), MPI_Scatter(), MPI_Gather(), MPI_Barrier(), MPI_Finalize()
  - Cuda - Completed CUDA implementation
- Odd-Even Transposition Sort:
  - MPI: MPI_Init(), MPI_Recv(), MPI_Bcast(), MPI_Scatter(), MPI_Sendrecv(), MPI_Finalize()
  - Cuda: Stuck on figuring out how to make processes in the GPU communicate with eachother as data needs to be transferred between processes inside the GPU, and Grace is down which makes it harder to experiment
- Radix Sort:
  - MPI - MPI_Init(), MPI_Bcast(), MPI_Send(), MPI_Recv(), MPI_finalize()
  - Cuda - Integration for MPI took longer than expected. Did not get a chance to test with Cuda before the Grace maintainance.

4. Performance Evaluation

- Sample Sort:
  - MPI -
    - By running the algorithm with different input sizes and threads, I was able to observe the timing and scaling of the algorithm.
    - Thicket Tree

      ```
      1.000 main
      ├─ 1.000 comm
      │  ├─ 1.000 comm_large
      │  └─ 1.000 comm_small
      ├─ 1.000 comp
      │  └─ 1.000 comp_large
      │     └─ 1.000 comp_small
      └─ 1.000 data_init
      ```

  - CUDA
    - Grace has paused my jobs and will not run them. I have over 5 queued jobs and all of them are waiting on Grace.
    - Unable to prodice thicket tree because Grace is not running my jobs. They have been queued for over an hour

```
1.000 main
├─ 1.000 comm
```

```
│  ├─ 1.000 comm_large
│  │  └─ 1.000 cudaMemcpy
│  └─ 1.000 comm_small
├─ 1.000 comp
│  ├─ 1.000 comp_large
│  └─ 1.000 comp_small
├─ 1.000 correctness_check
└─ 1.000 data_init
```

- Mergesort:
  - MPI -
    - Tried running the algorithm with larger array sizes & threds to observe how it scales. Read it into thicket to see the thicket tree and data frames, currently working on the plotting aspect of things.
    - Thicket Tree:

```
1.000 main_region
└─ 1.000 whole_computation
   ├─ 1.000 check_correctness
   ├─ 1.000 comm
   │  ├─ 1.000 comm_gather
   │  └─ 1.000 comm_scatter
   ├─ 1.000 comp
   │  ├─ 1.000 comp_large
   │  └─ 1.000 comp_small
   └─ 1.000 data_init
```

  - CUDA -
    - Implemented the cuda version of mergesort algorithm and ensured that it scales and works with larger array sizes and threads. Read it into thicket to see the thicket tree and data frames, currently working on the plotting aspect of things.
    - Thicket Tree:

```
1.000 main_region
├─ 1.000 comp_small
└─ 1.000 whole_computation
   ├─ 1.000 check_correctness
   ├─ 1.000 comm
   │  ├─ 1.000 comm_toDevice
   │  └─ 1.000 comm_toHost
   ├─ 1.000 comp
   │  └─ 1.000 comp_large
   └─ 1.000 data_init
```
```
```

- Odd-Even:
  - MPI -
    - Managed to get algorithm to scale, generated some CALI files and produced thicket tree. Working on plotting the CALI files. Looking at the data frame inside of Jupyter the algorithm seems to scale normally.
    - Thicket Tree:

```
1.000 main
└─ 1.000 whole_computation
   ├─ 1.000 check_correctness
   ├─ 1.000 comm
   │  └─ 1.000 comm_large
   │     ├─ 1.000 MPI_Gather
   │     ├─ 1.000 MPI_Recv
   │     ├─ 1.000 MPI_Scatter
   │     └─ 1.000 MPI_Send
   ├─ 1.000 comp
   │  └─ 1.000 comp_large
   └─ 1.000 data_init
```

  - CUDA -
    - Managed to get algorithm to scale, generated some CALI files and produced thicket tree. Working on plotting the CALI files. Scaling for this CUDA as of now is somewhat unique as the blocks are always going to be array_size/2 as the block id determines which phase (either odd or even) gets completed.
    - Thicket Tree:

```
1.000 comm_small
1.000 comp_small
1.000 whole_computation
├─ 1.000 comm
│  └─ 1.000 comm_large
├─ 1.000 comp
│  └─ 1.000 comp_large
└─ 1.000 data_init
```

- Radix Sort:
  - MPI -
    - I got the algorithm to scale, generated over 240 cali files and produced a thicket tree; last few files were not able to run due to the largest array sizes. Jupyter files ran smoothly and produce weak, strong, and speedup sclaing appropriately.
    - Thicket Tree:

```
1.000 main
└─ 1.000 whole_computation
   ├─ 1.000 check_correctness
   ├─ 1.000 comm
   │  └─ 1.000 comp_large
   │     └─ 1.000 comm_large
   ├─ 1.000 comp
   │  └─ 1.000 comp_large
   │     └─ 1.000 comm_large
   └─ 1.000 data_init
```

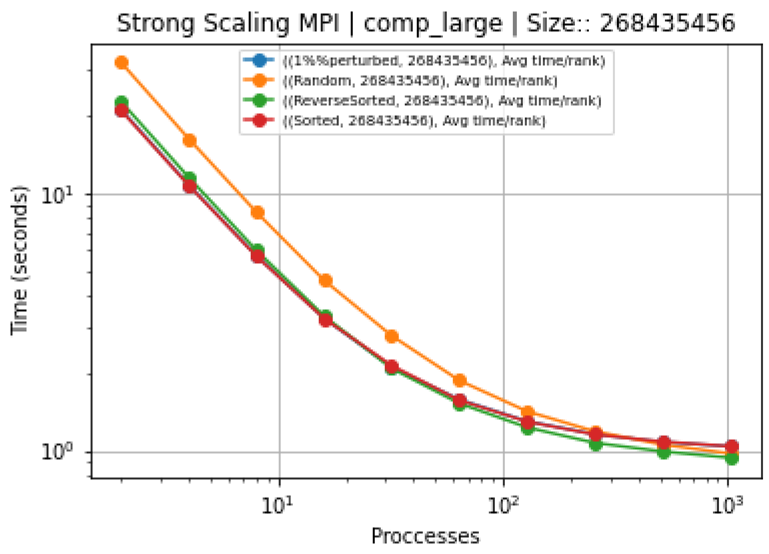
- CUDA -
  - I was able to produce the CUDA implementation to its fullest, got a ton of CALI files for testing purposes, produced a thicket tree as seen below and larger array sizes are continuously being tested for data analysis. Looking at the data frame inside of Jupyter the algorithm seems to scale normally.
  - Thicket Tree:

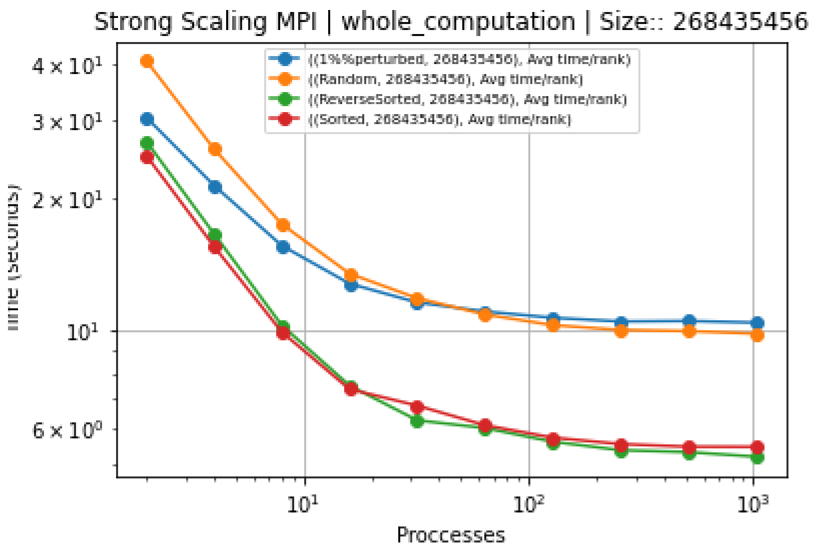
```
1.000 comm_small
1.000 comp_small
1.000 whole_computation
├─ 1.000 comm
│  └─ 1.000 comm_large
├─ 1.000 comp
│  └─ 1.000 comp_large
└─ 1.000 data_init
```

Final Report (Plots images are within each implementations folders)

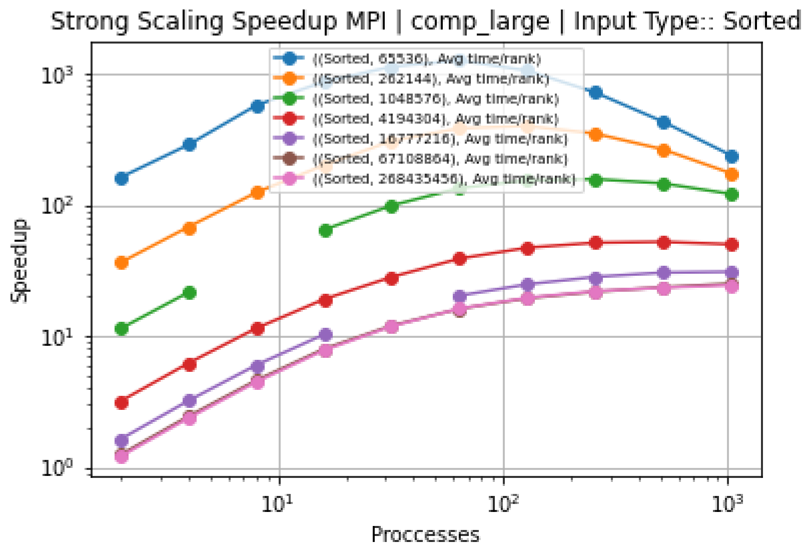
- Odd-Even Sort: - Riley Szecsy
  - MPI:

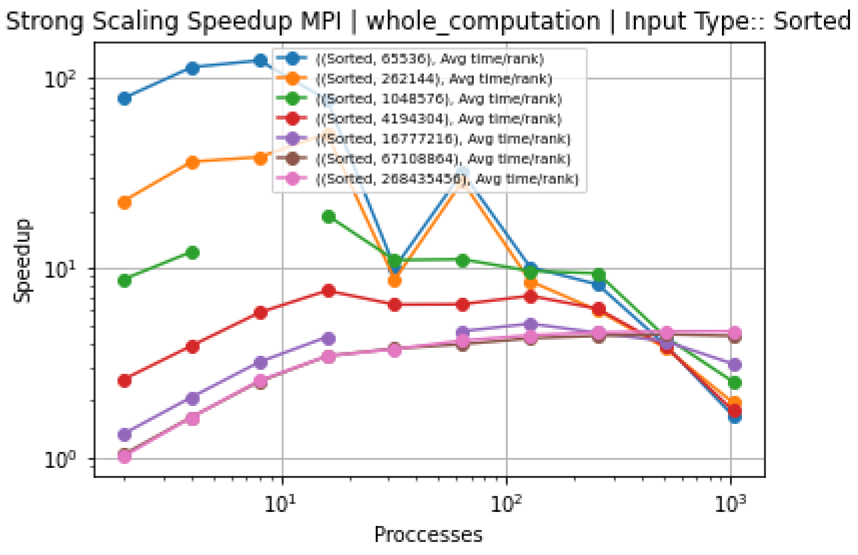
- Strong Scaling: When keeping the array size constant as processors increase it should be expected that time is decreasing. The first plot is for the comp_large region of the sort; it shows that this is the case for all input types of odd-even sort. More specifically this plot is for an array size of 2^28 where the plot is mostly a downward linear trend and flattens out around 128 processes. With smaller array sizes the flattening happens earlier as less work needs to be done which can mean processes end up waiting. When looking at the whole_computation portion of sort the same trends are occurring.



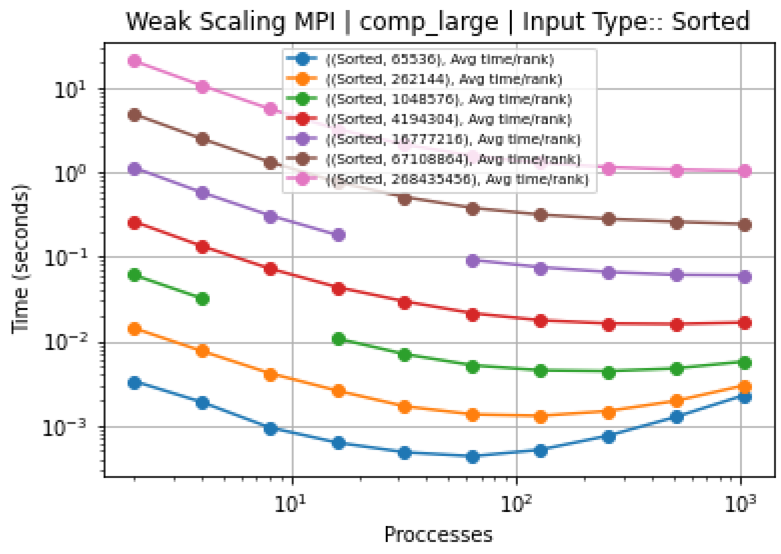


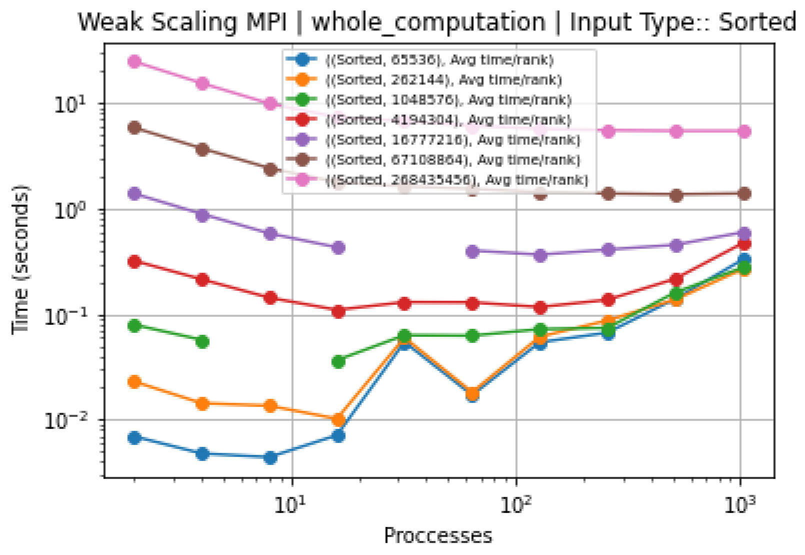
- Speedup: As processes get higher and time decreases there should be a scaled speedup. When looking at a speedup graph for lower array sizes (2^16, 2^18, 2^20) the optimal number of processes is 64. This makes sense as these sizes of arrays may not need as many processes which could result in processes waiting which adds time. For bigger array sizes (2^22, 2^24, 2^26, 2^28) speedup is still happening even at 1024 processes which is reasonable as these array sizes have more work that needs to be done which can be leveraged by more processes. Like before the whole computation follows generally the same trends, as the whole_computation region is dominated by comp/comp_large. The only thing different about the plots when looking at the whole computation is that there is a dip in the speedup graph for sizes 2^16 and 2^18 however this is most likely due to resource allocation when queuing jobs as those particular jobs may have been allocated nodes that were father apart causing the unusual dip.





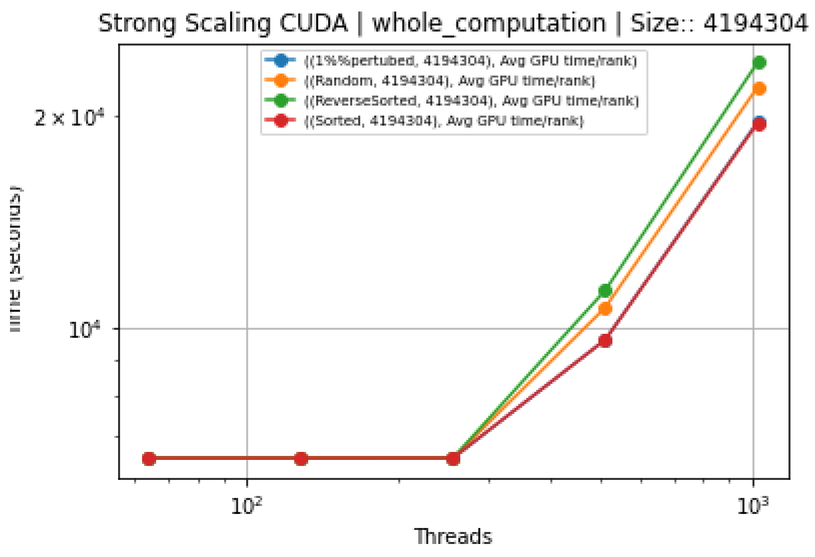
- Weak Scaling: Generally a weak scaling graph should be flat, as the load per process is the same as you increase process and array size. This is generally the case for odd-even sort. When looking at a weak scaling graph that has similar results to the speedup graphs, for smaller array sizes (2^16, 2^18, 2^20) there is a valley at 64 processes which is the same as the optimal number of processes found in the speedup graph. For bigger arrays the scaling is slowly going down which also reflects what the speedup graph was showing. As for the whole_computation the trends are almost identical, smaller array sizes (2^16, 2^18, 2^20) start to skew up a process's increase where bigger array sizes (2^22, 2^24, 2^26, 2^28) are basically flat across. The same dip occurs which adds more evidence to that being a resource allocation anomaly rather than a problem with the algorithm.



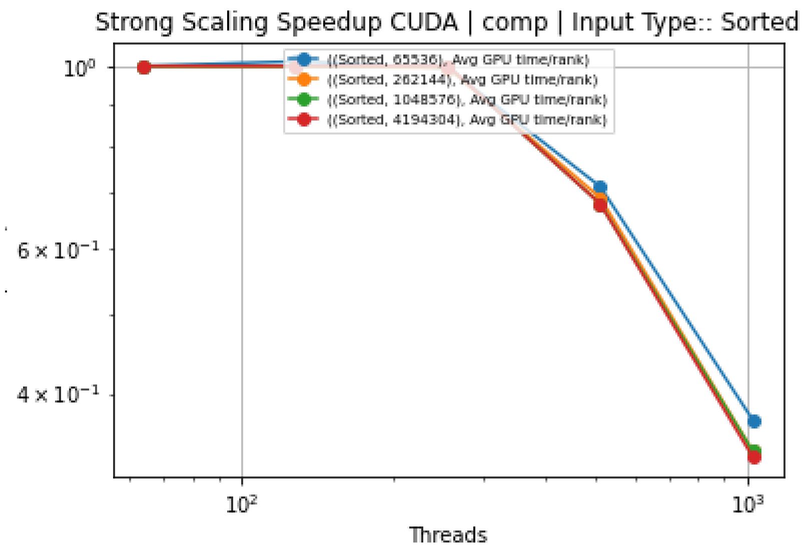


- CUDA:

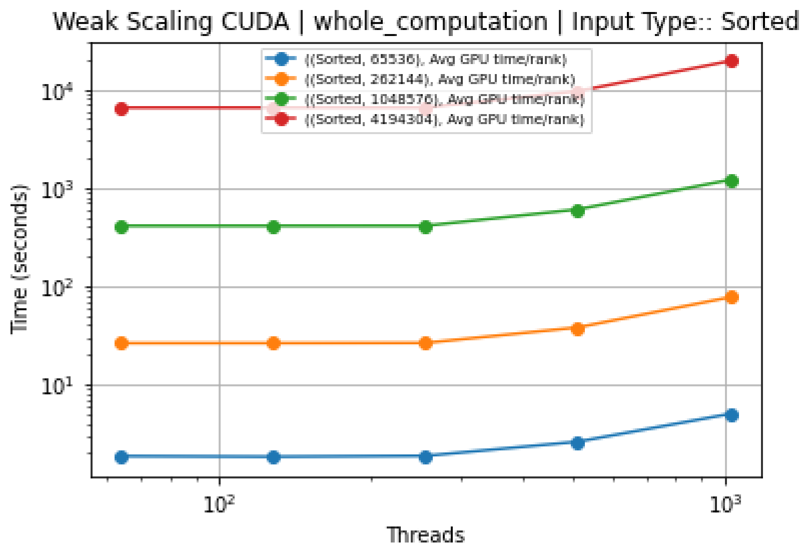
- Strong Scaling: The CUDA implementation of odd-even sort does not scale well as evidenced by the strong scaling plot. The reason I think this is due to the personal implementation of the algorithm. The number of blocks used in the kernel call is InputSize/2, it was originally attempted with n/threads number of blocks however the array would end up unsorted. InputSize/2 blocks is significantly more blocks than InputSize/threads which means more blocks are trying to access the same memory and resources which could be the reason for the poor scaling in the CUDA implementation of odd-even sort. Ultimately the sort could only be scaled to a size of 2^22.



- Speedup: Because of the implementation and how poorly the odd-even sort scales there is really no expected speedup. That is evidenced by the plots. As mentioned before this is most likely because the number of blocks is so high and they are all fighting over memory due to the amount of access that is being done.



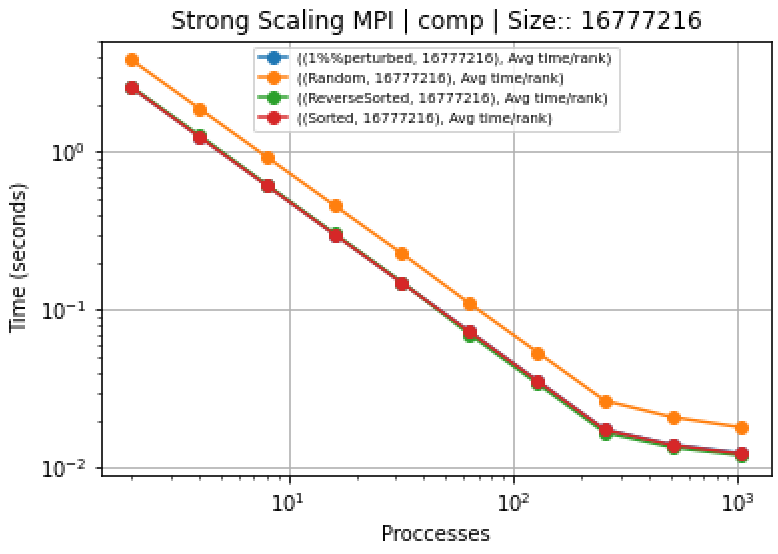
- Weak Scaling: As mentioned before the load per thread should be the same as both array size and thread amount increases. This is reflected in the odd-even weak scaling plot. With a slight upward trend at array size 2^22. However this really does not give any insight into why it is scaling poorly.



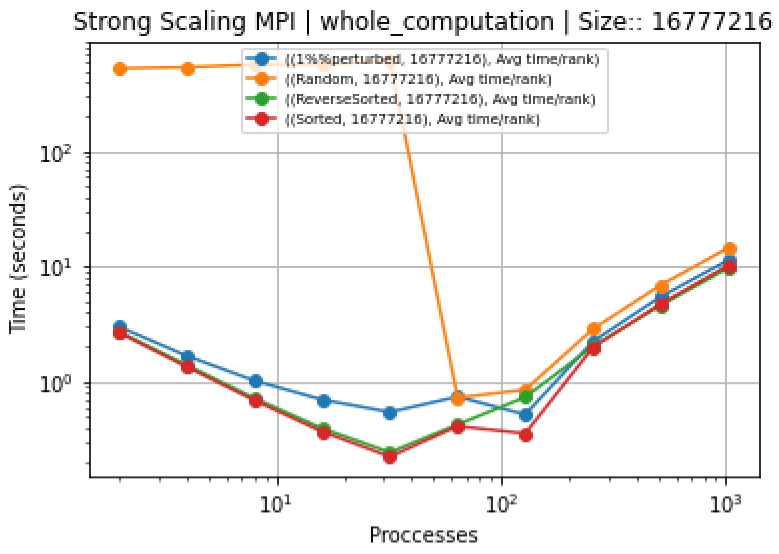
The plots linked in the report are the ones that clearly and concisely represent the algorithm's behavior. If more context is wanted or needed all of the MPI and CUDA odd-even plots are located within the repo.

- Merge Sort: - Nebiyou Ersabo
  - MPI:

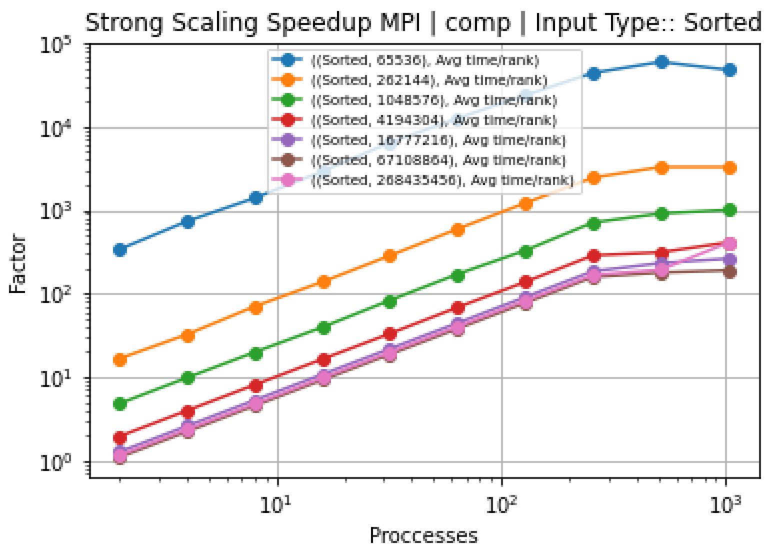
- Strong Scaling: With constant problem size and increased number of processors we expected to see a better time. We also expected that there could be diminishing results due to communication overheads. This expectaion was validated with trend from our computation graph below for array size of 2^24. As we can see from the plot the general trend is that computaiton time goes down up to 128 threads and then it starts declining at lower rates possibly due to communication overhead.



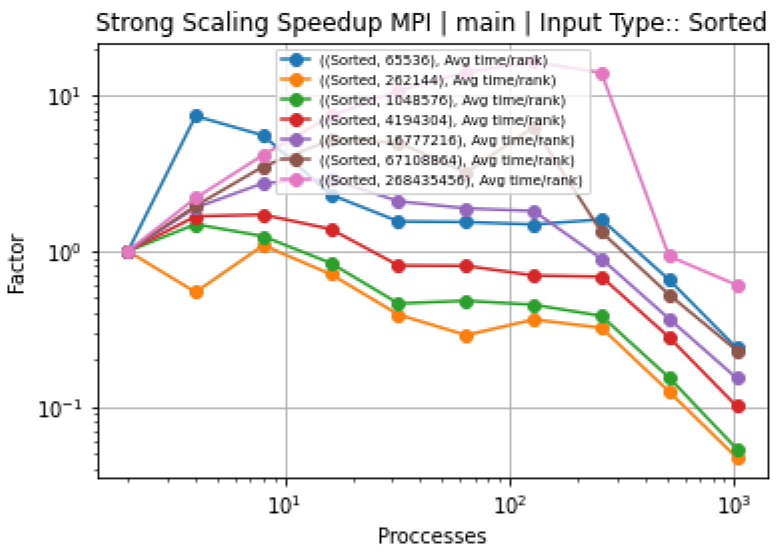
For whole computation, from the graph below of array size 2^24, we can see that as the number of processes increases the overall time decreases upto 64 threads, and then we start getting diminishing returns due to communication overhead.



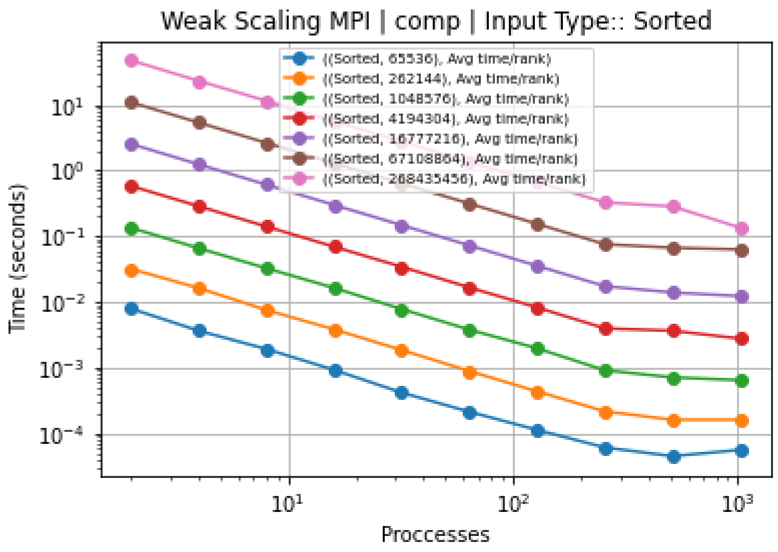
computation graph for array size 2^24 with sorted input type.



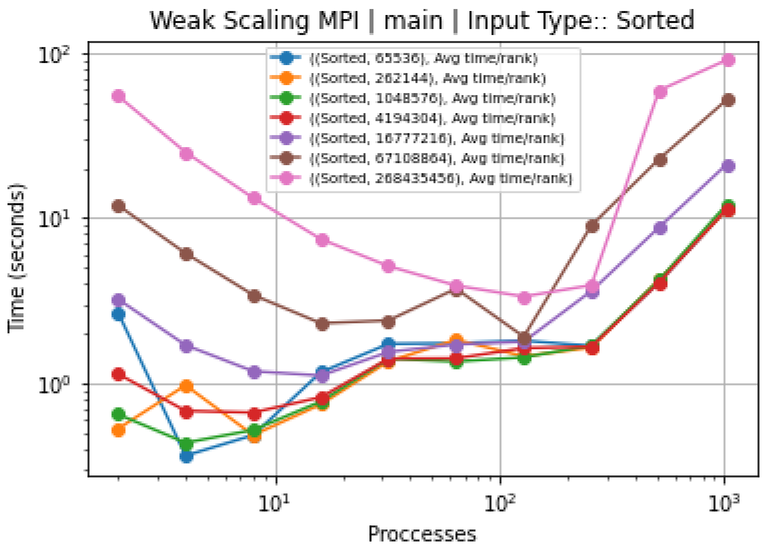
However, the the main graph below for all array sizes with sorted input did not speed up as expected due to inefficient data initialization methods which took a longer computation time for larger arrays and communication overhead.



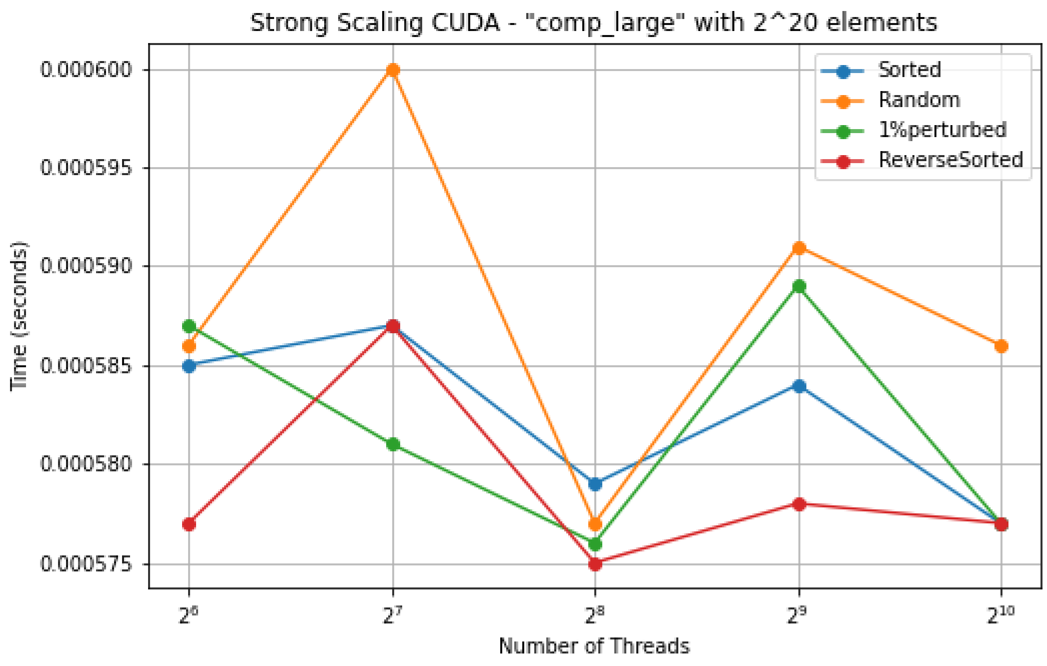
- Weak Scaling: when we increase both the problem size and threads proportionally we expected the runtimes to be relatively constant since it would be a constant load per processor. We observed the relatively constant trend towards higher processorts for computation graph of sorted input below, despite its initial decline.

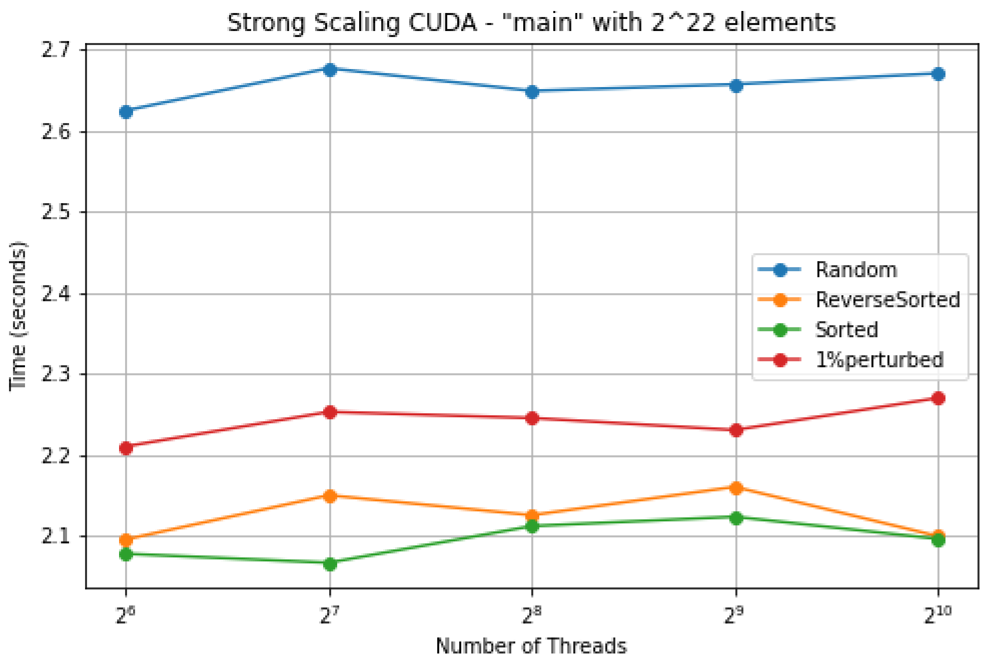


For main part of sorted input Type, the constant expected trend only exist somewhere in the between 64 and 128 threads despite the inconsistent data points on left and right edges. The causes for those could be resource limitations and communication overheads due to larger array sizes as well.

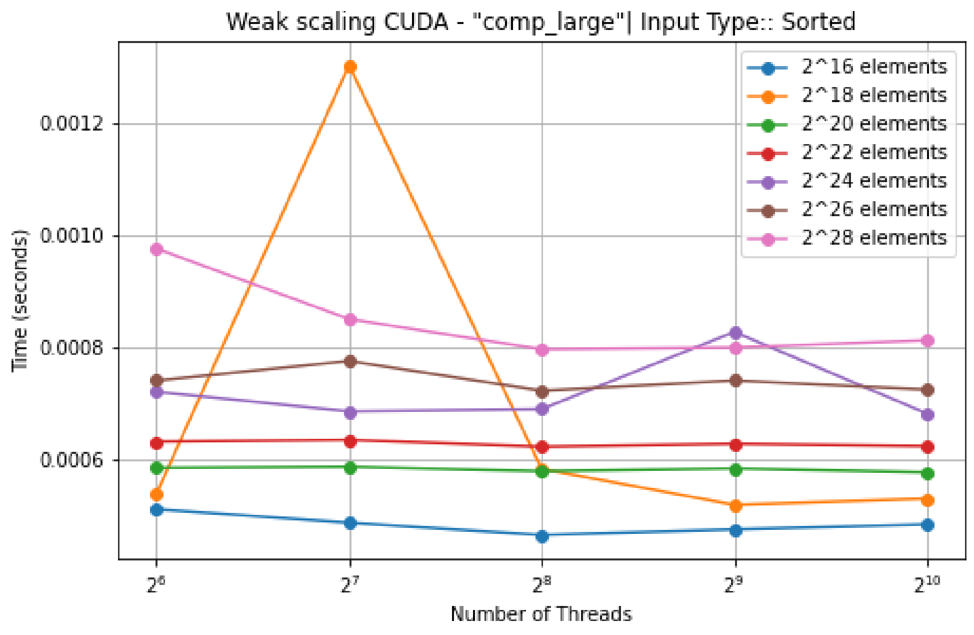


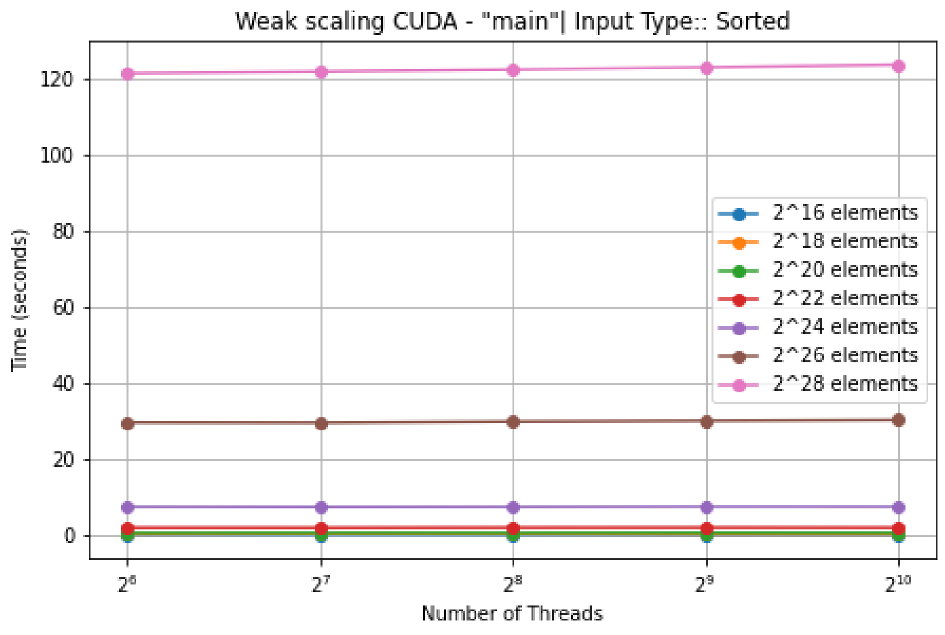
- CUDA:
  - Strong Scaling: as more GPUs are utilized we expected to see less computation time for strong scaling. As we can see from the first graph below, the large computation graph for 2^20 array size did not have less time as the number of processors increases. The main function region for array size of 2^22 in the second plot below also didn't exhibit less runtime with increases number of processors. This is probably due to algorithm implementation errors, memory bandwith limitaions, and communication overheads.


Strong Scaling CUDA - "comp_large" with 2^20 elements
Time (seconds)
Number of Threads
Sorted
Random
1%perturbed
ReverseSorted
0.000575
0.000580
0.000585
0.000590
0.000595
0.000600
2^6
2^7
2^8
2^9
2^{10}

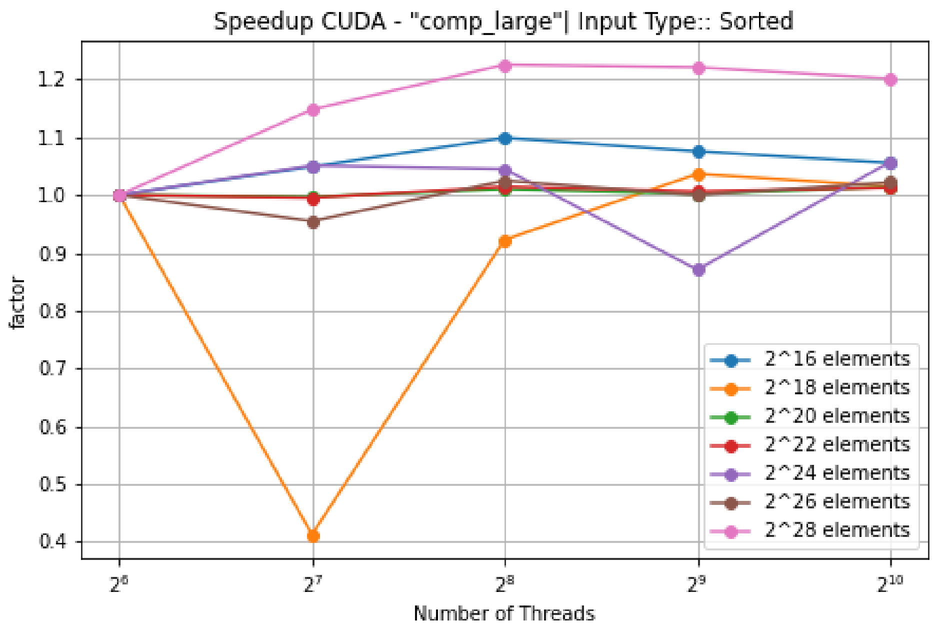

Strong Scaling CUDA - "main" with 2^22 elements
Time (seconds)
Number of Threads
Random
ReverseSorted
Sorted
1%perturbed
2.1
2.2
2.3
2.4
2.5
2.6
2.7
2^6
2^7
2^8
2^9
2^{10}

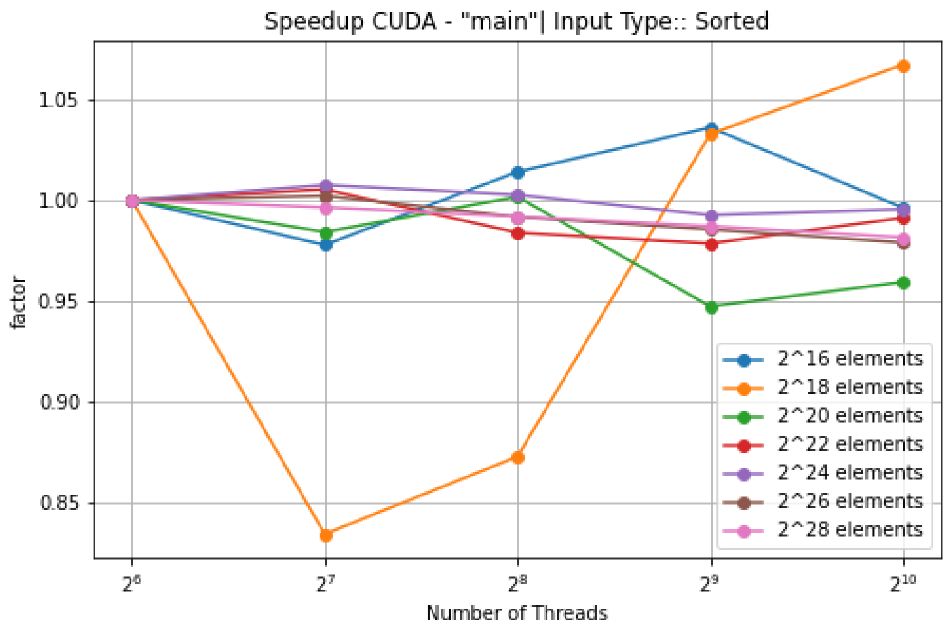
- Weak Scaling: With proportional increase of array size and processors, we expected to see a nearly constant runtime plot. Both graphs below exhibit the expected trends since the load per processor is constant when both array size and processors are increased proportionally. The first graph below is the weak scaling for comp_large for all array sizes with sorted input type. The second graph below is the weak scaling plot for main for all array sizes with sorted input type.





- Speedup: we expected to see a more linear or logarithmic trend showing faster runtimes with more processors. For comp_large in the first plot below, it did scale for the biggest array size but stayed relatively constant and even lower with increased number of processors. For main plot in the second plot below it mostly stayed relatively constant and even lower for some with increase processors. The reason why it is not scaling as expected shows me that there could have been inefficiencies in the implementation of mergesort cuda algorithm, data generation, and communication overheads.

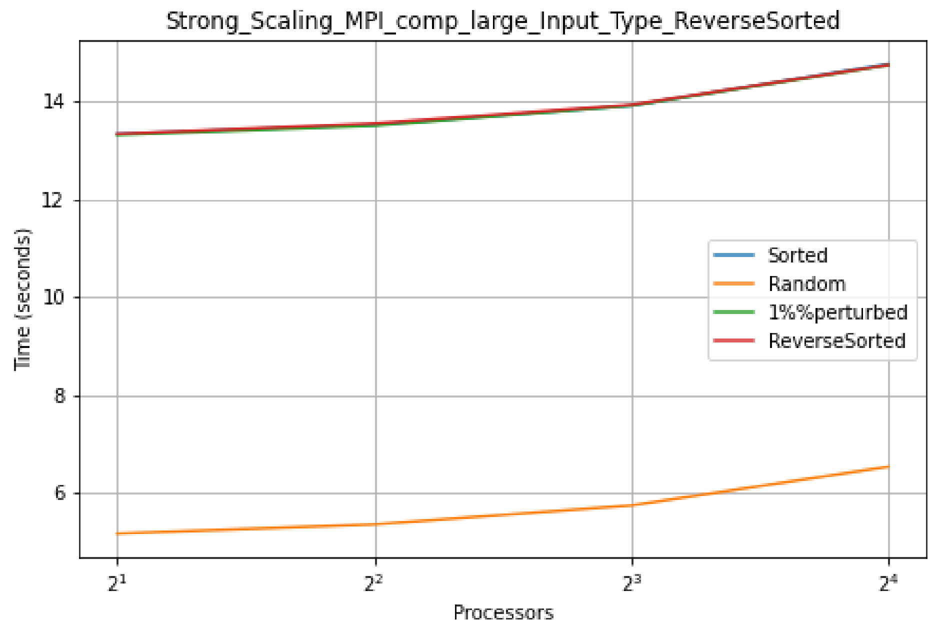


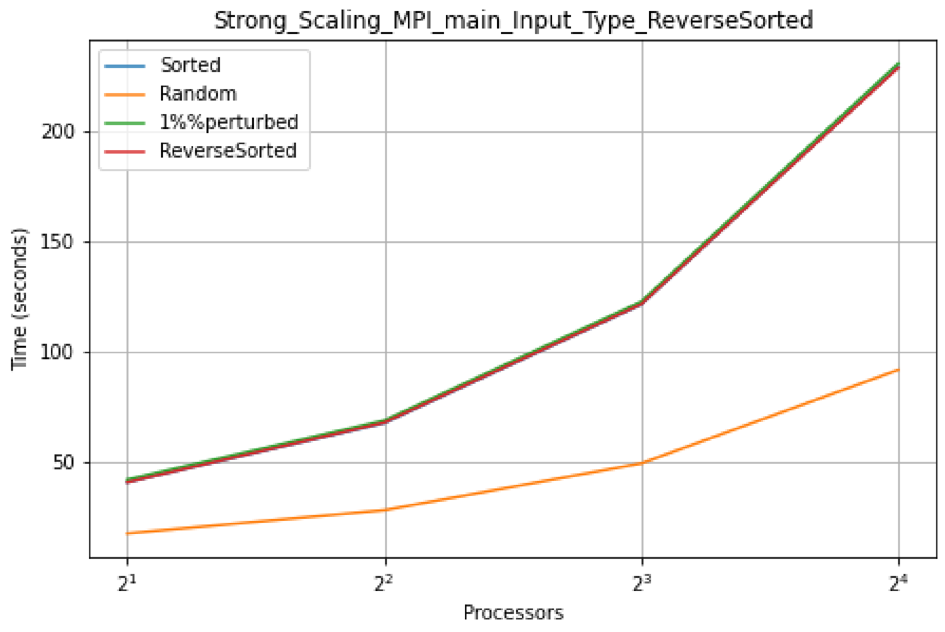


Some of the plot here are selected as they show the algorithms behavior in both MPI and CUDA implementation. All the other 190+ plots for each implementation are found here in this repo. MPI_Plots and CUDA_Plots

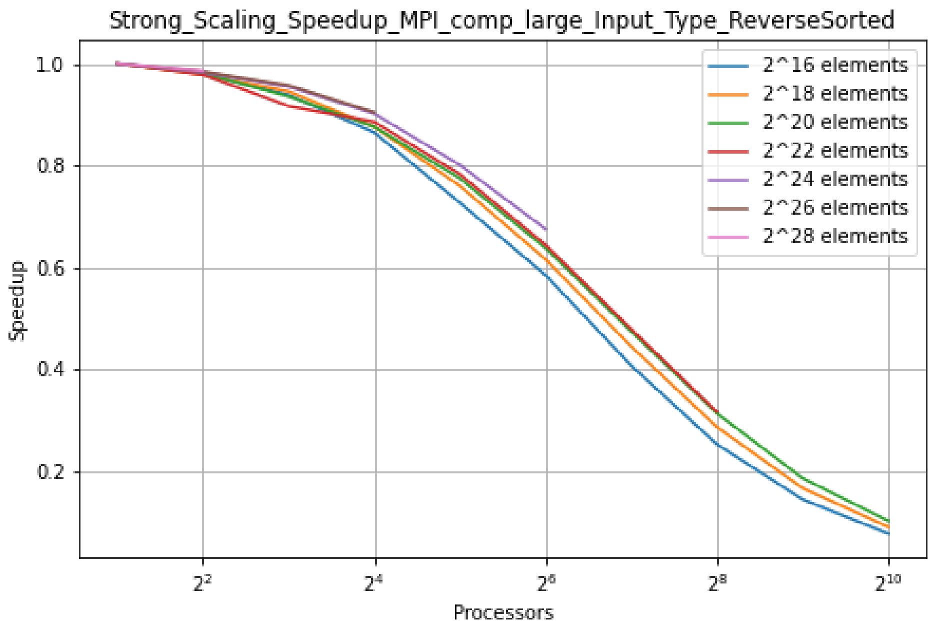
- Radix Sort: - Jonathan Kutsch
    - MPI:

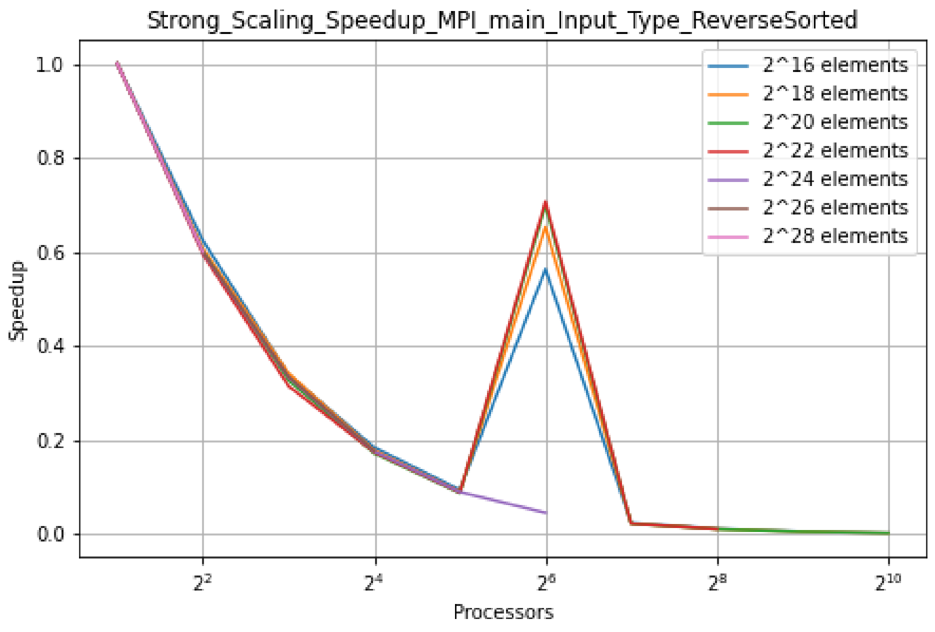
- Strong Scaling: In my graphs, I notice an inverted correlation between computational runtime and processes. We expect to see a decrease in time as the number of processes increase, however, we see that the more workers there are, the more communication overhead present leading to issues with dividing up the work and aggregating the result back. The optimal number of processes in my case is when the time is the least with the smallest array sizes. When looking at the main portion of the sorting algorithm below, you can see that this trend is occuring due to the way that I designed the algorithm as well as in the comm graph that shows the amount of data being communicated leading to overhead.



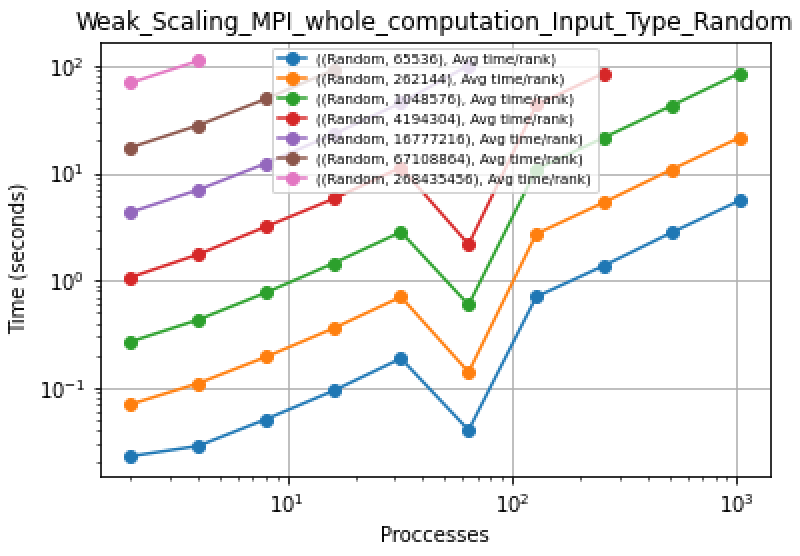


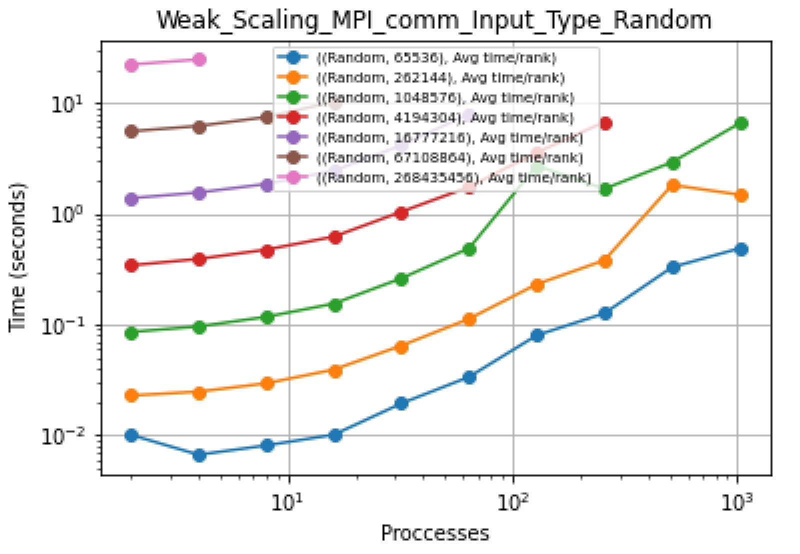
- Speedup: When looking at the speedup graphs below, there is a gradual dip as the number of processes increase. Similar to the strong scaling above, this is due to how the number of processes are handled on a single core. As the workers increase, the communication overhead increases and spikes the speedup. It is clear that the optimal number of processes is at the lowest matrix size at 2^16, 2^18, and 2^20. As the matrix size increases, the overhead increases, and results in more processes waiting as they are hitting one core, effectively adding time and killing the speedup. Additionally, it is important to note the jump in speedup at 64 processes, which can be possible through a change in communicaiton overhead, resource allocation, and the varied nodes in each computation.





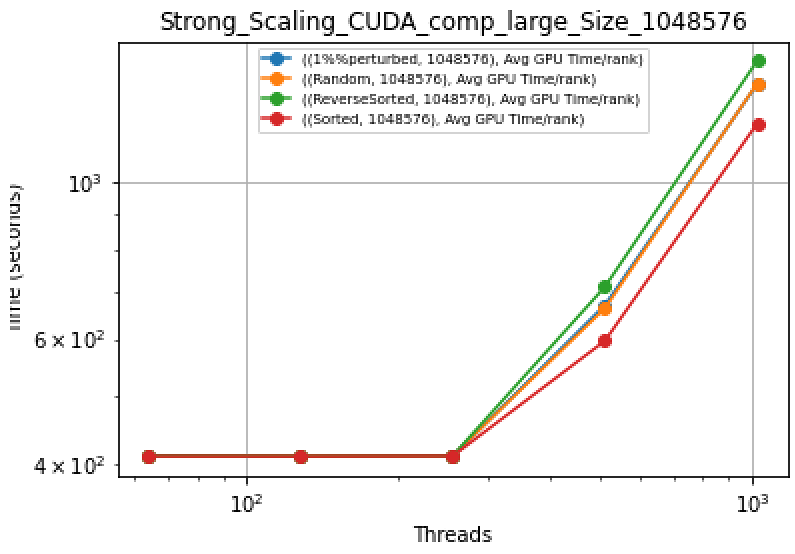
- Weak Scaling: In a weak scaling plot, we anticipate a steady level as both the array size and the number of processes increase, signifying a uniform load per process. As seen in the graphs below, this is acurately the case for radix sort. Similar to the speeedup graphs where we seed a change at 64 processes, we see a sharp dip in time as it changes from 32 to 64 processes. This is important as 64 processes represent the optimal configuration for the fastest computation. Lastly, we see in the graph below that it is consistently flat across the board for all array sizes and input types. This holds true for all array sizes and input types as it does in the speedup graph. With this dip, I see that there are issues with my algorithm and the way I allocated the resources for each computation.

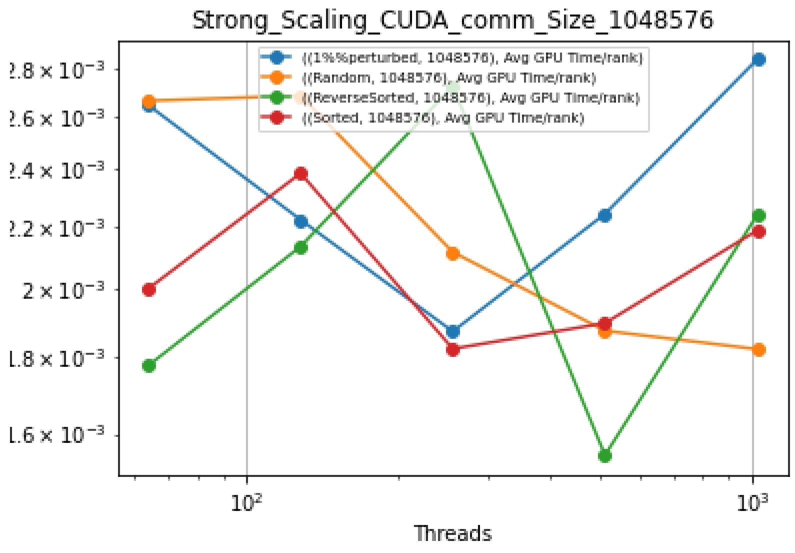




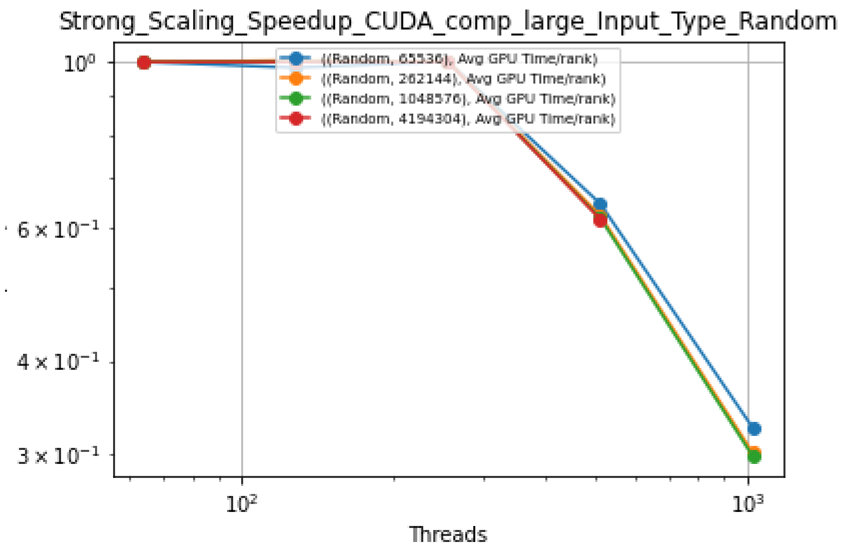
- CUDA:

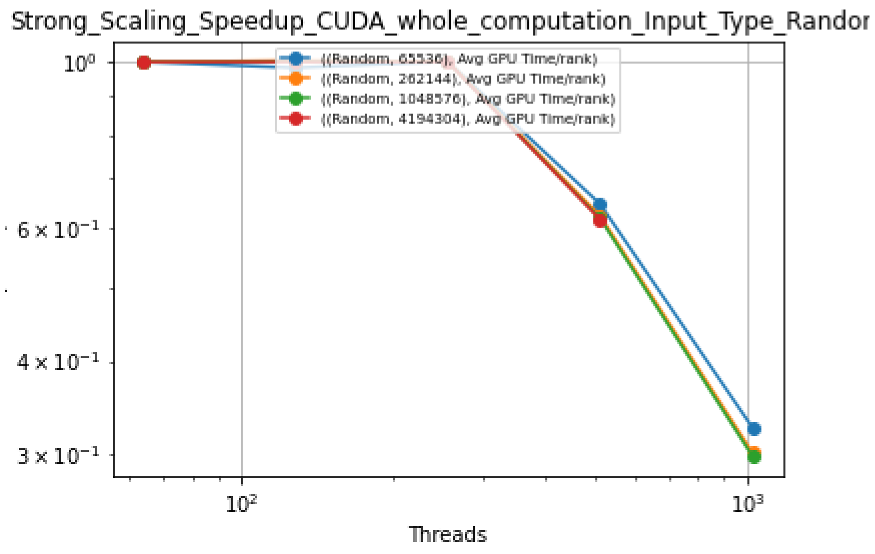
- Strong Scaling: The radix sorting algorithm did not scale well in the CUDA implementation as seen in the strong scaling plots below. I attribute this limitation to the algorithm's construction and the associated communication overhead. In the Jupyter code, the number of blocks used was set to InputSize/2. Originally, it was executed with n/threads number of blocks, resulting in the initial array not being sorted correctly after compilation. The choice of InputSize/2 blocks represents a significantly higher number than n/threads, causing more blocks to contend for the same memory and resources, which I think ultimately leads to the poor scaling of the radix sorting algorithm in the CUDA implementation. Additionally, the algorithm's scalability was constrained, with the sort only accommodating array sizes up to 4194304, unable to handle sizes greater than 2^22.



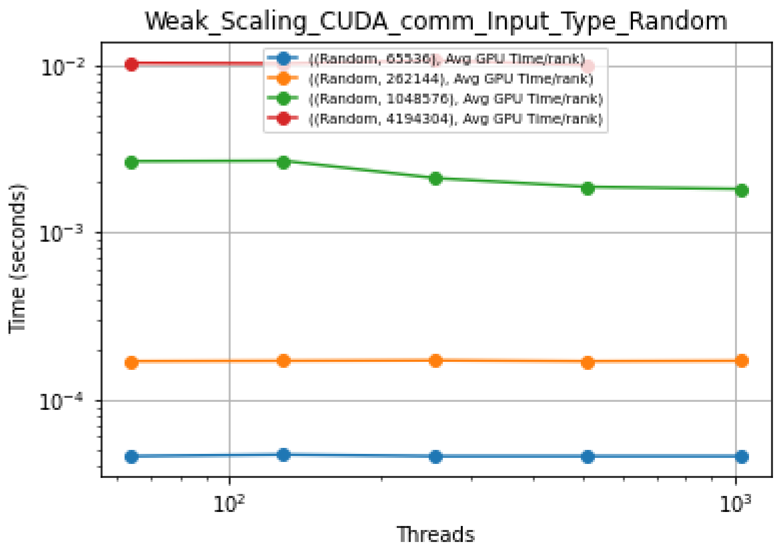


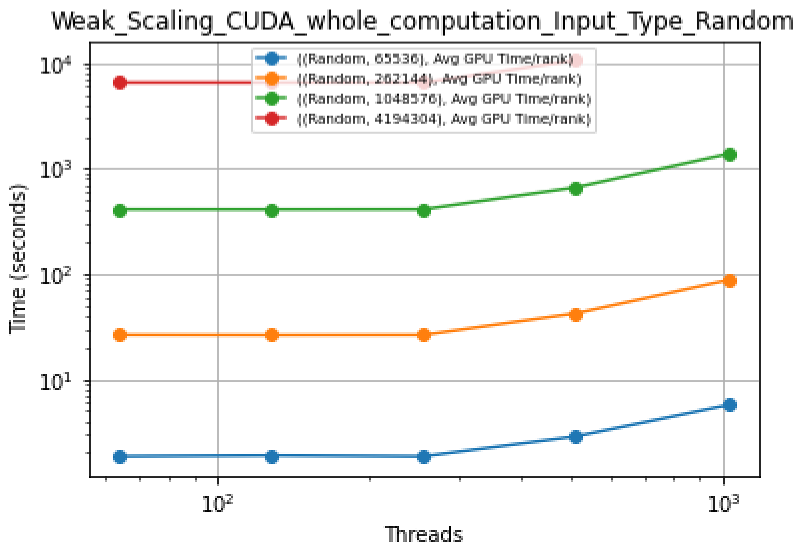
- Speedup: Due to the suboptimal implementation and the limited scalability of the radix sort, any anticipated speedup is essentially non-existent as seen below by the plotted data. As I described in the strong scaling above, this lack of improvement is likely attributed to the excessively high number of blocks, leading to a battle for memory resources due to the large amount of access that is required.





- Weak Scaling: As mentioned before the load per thread should be the same as both array size and thread amount increases. This is reflected in the radix weak scaling plot. We can see that there is a marginal uptick at 2^22 array size, but it is unclear as to why it is scaling poorly. Upon further analyzing the algorithm, it could be possible to sort at larger array sizes and have a clearer picture of scaling.

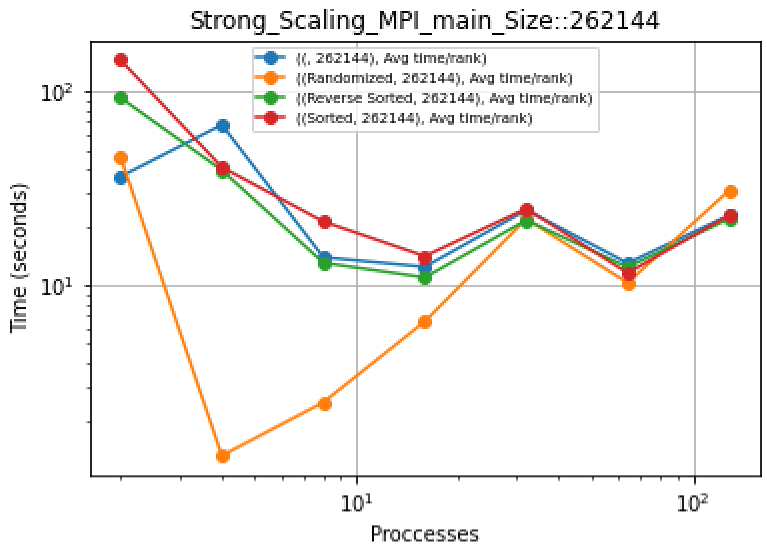


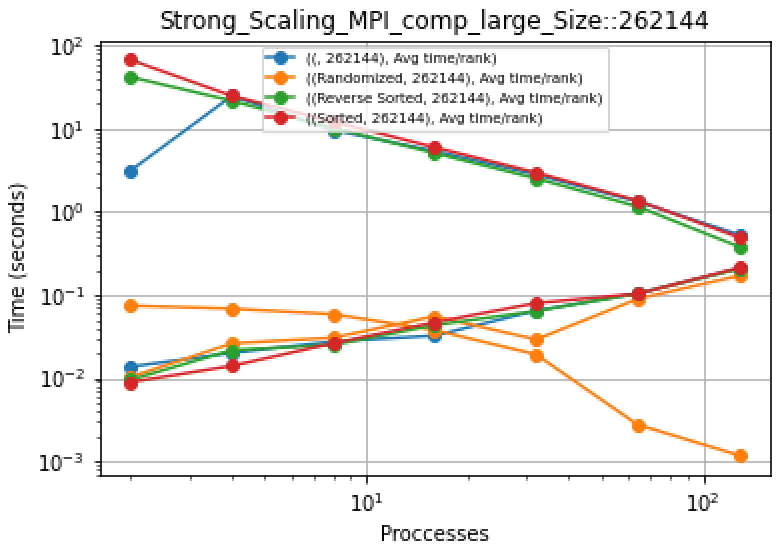


Some of the plot here are selected as they show the algorithms behavior in both MPI and CUDA implementation. All the other 190+ plots for each implementation are found here in this repo. MPI_Plots and CUDA_Plots

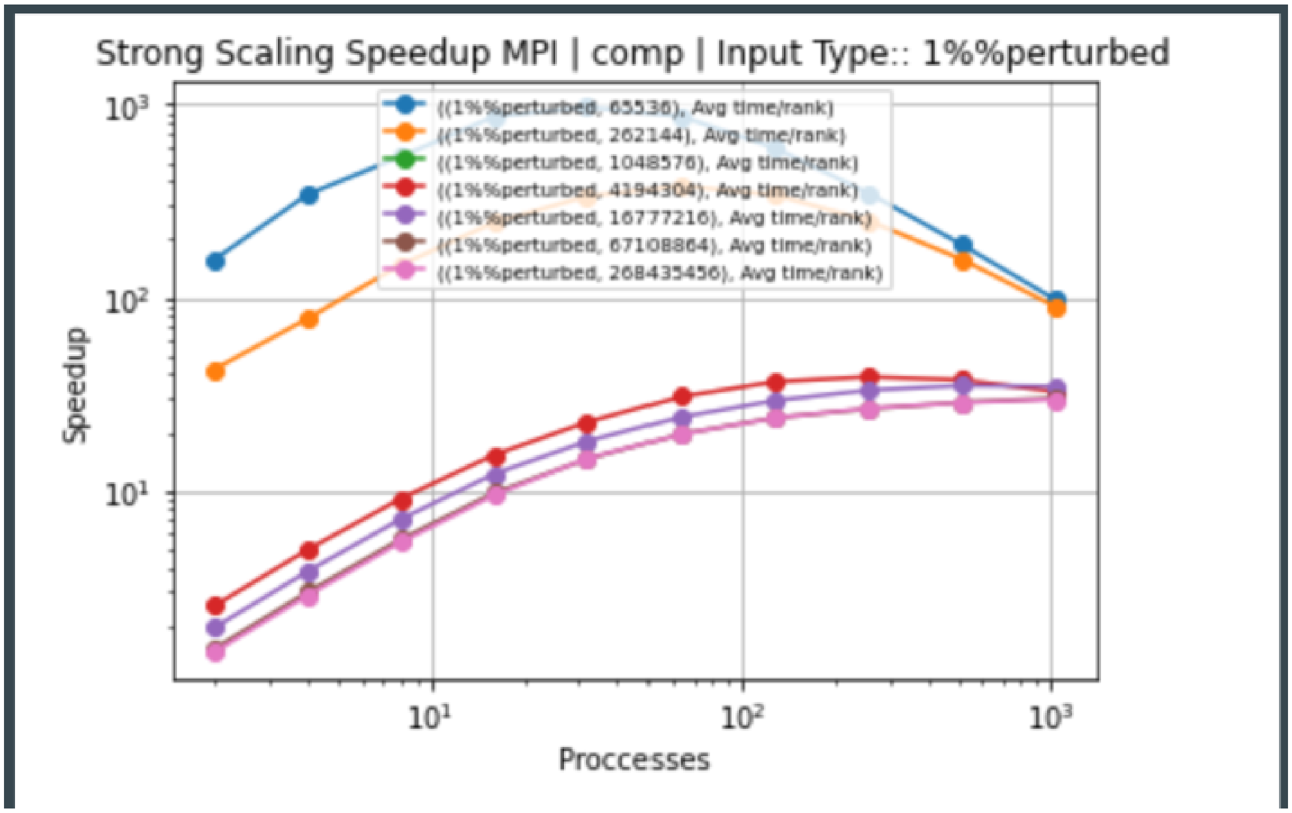
- Sample Sort: - Vincent Lobello
  - MPI:
    - Strong scaling for sample sort MPI was consistent with what was expected. With keeping array size 2^18 and increasing the number of processes, it is expected to see a decrease in time which is consistent with the "main" and

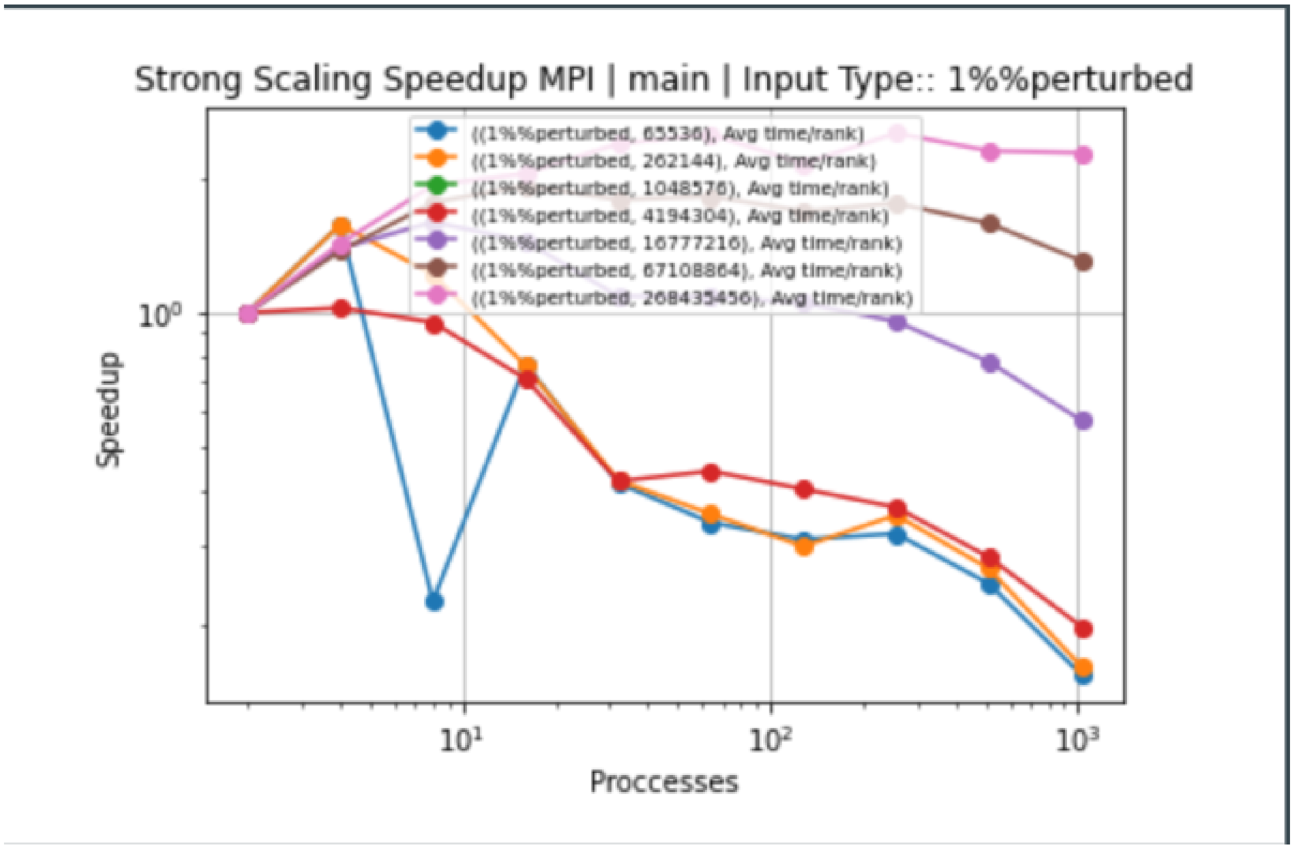
"comp large" graphs below. Main does see a spike back up in time once the processes reach a certain point because of the overhead and communication being inefficient. My claim is supported by seeing the constant decrease in time in the "comp large" graph. Both graphs support the expected outcome of a strong scaling experiment.



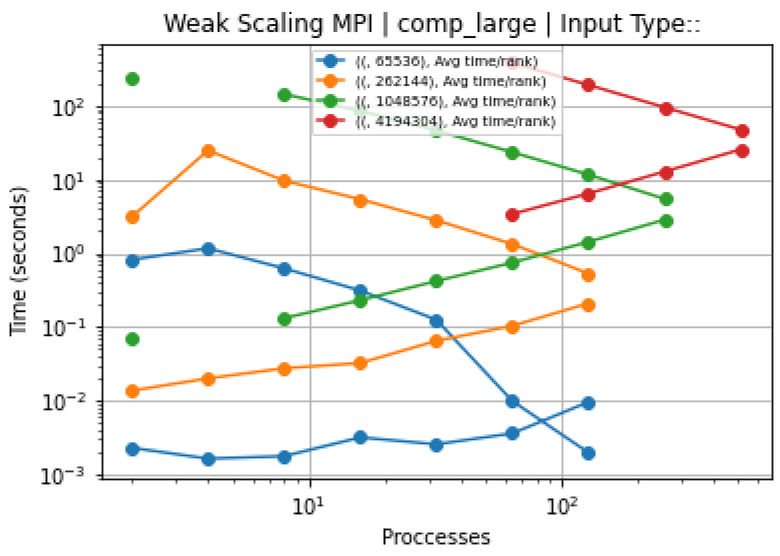


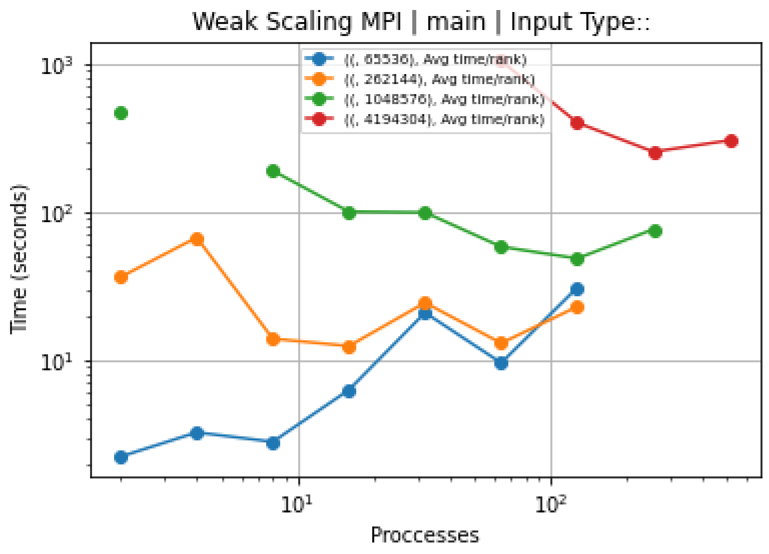
- Speedup for sample sort MPI was seen to increase as the number of processes increases. Looking at the 1% pertubed input, you can see that the peak speed up for "comp" was around 128 processes, and similarly for "main". The "main" graph has smaller array sizes decrease in speedup as processes increase because of the communication portion and overhead of the algorithm. The larger input sizes for "main" followed a similar trend as "comp", increasing in speedup as processes increased.



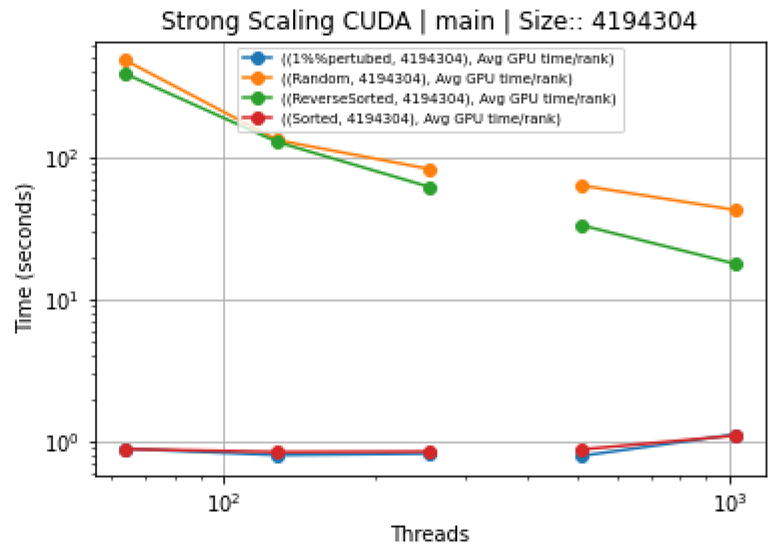


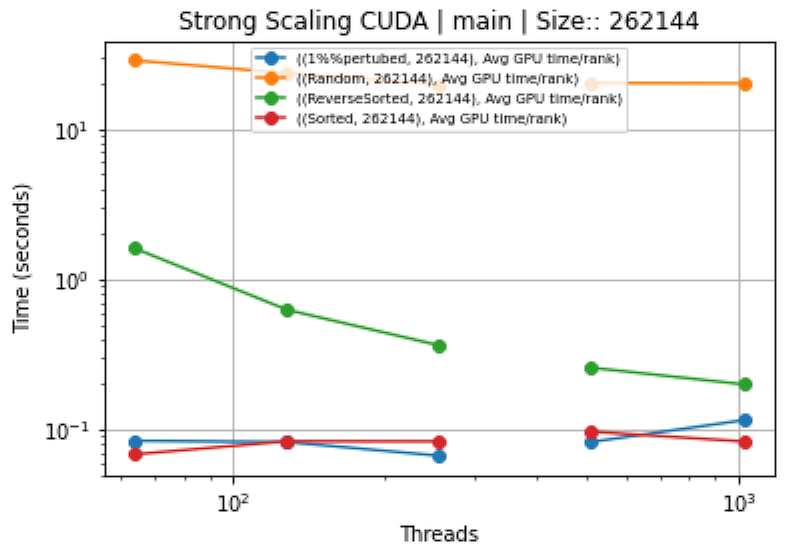
- Weak Scaling for sample sort MPI was looked at by viewing the "main" and "comp_large" sections of 1% pertubed input of the algorithm. Weak scaling is supposed to see a constant time as the array size and processes increase. The graphs don't show a completely constant run time, but they are not seen with very large discrepency. My algorithm most likely does not communicate properly or efficiently, making the graphs not as constant as they should be.



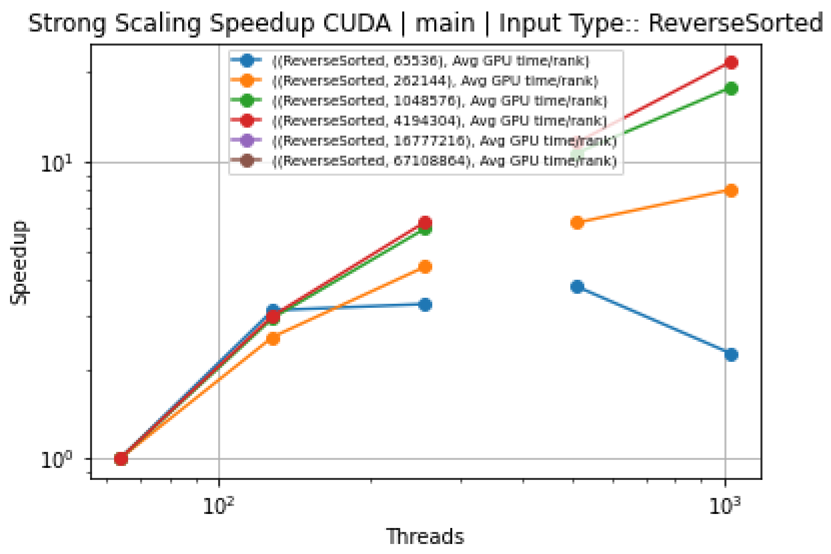


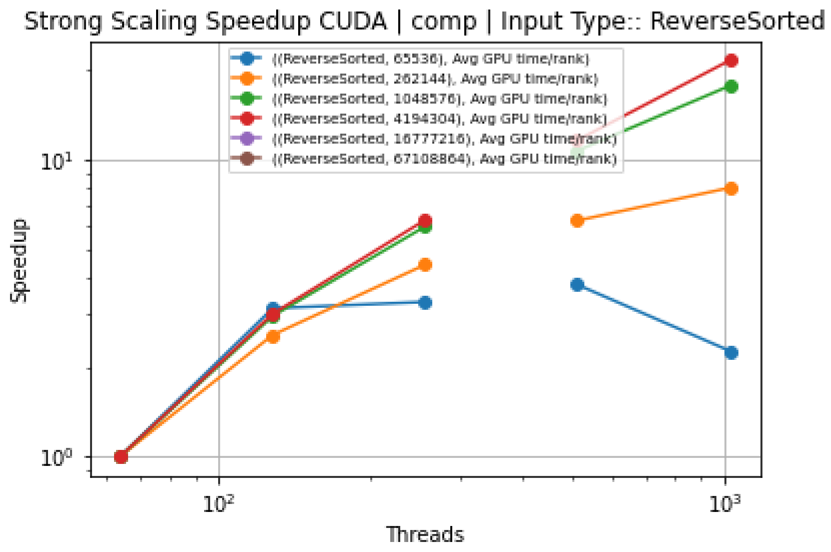
- CUDA:
  - Strong Scaling for sample sort CUDA saw a general decrease in time while increasing the threads ran on. The largest change in times was for input types random and reversed, seeing a decrease in time in "main" for both array sizes, 2^22 and 2^18. This is consistent with what is expected because as the thread count increses, the overhead decreases, increasing speed of the algorithm.



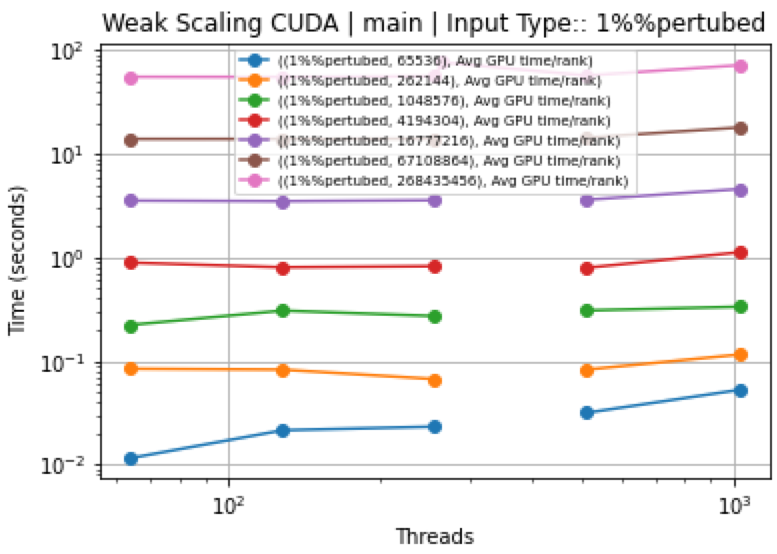


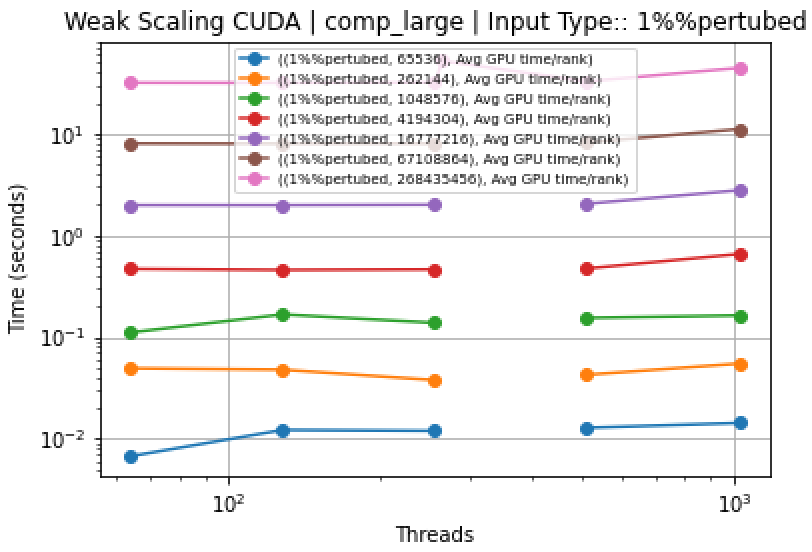
- Speedup for sample sort CUDA was very conistsent with what was expected. The "main" and "comp" for input type reverse, saw increases in speedup as the threads increased. The speedup was most prominent in the larger array sizes, with the smallest array, 2^16, flattening out when reaching the highest thread counts. The larger inputs maintained a consistent speedup throughout the experiment.



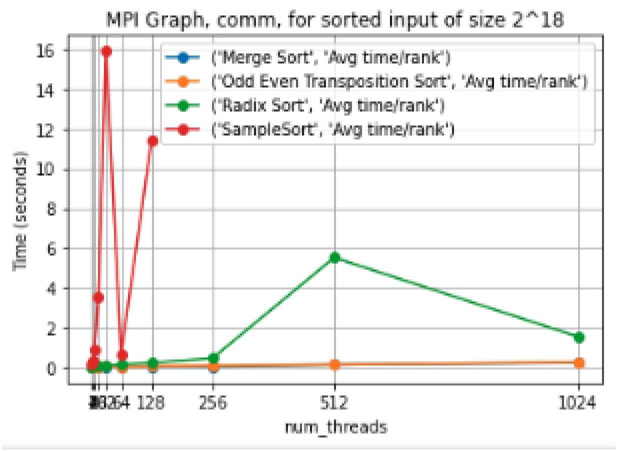


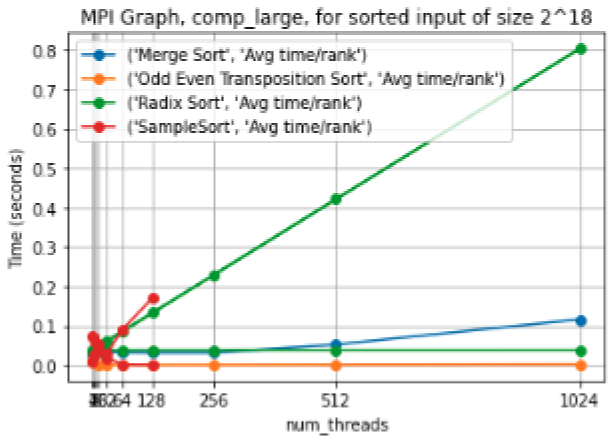
- Weak Scaling for sample sort CUDA was very consistent with theory, as the array size increases with thread count also increasing, time remains very constant. As seen in the graphs the time maintained a very flat line with some up and down most likely related to hardware or network discrepencies. The weak scaling of sample sort on CUDA was performed very well by my algorithm, keeping the constant time.

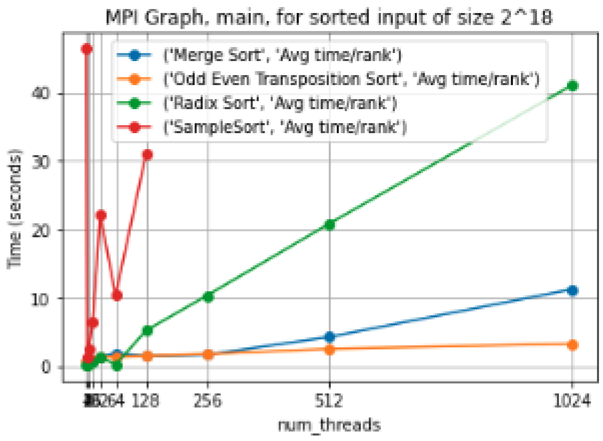




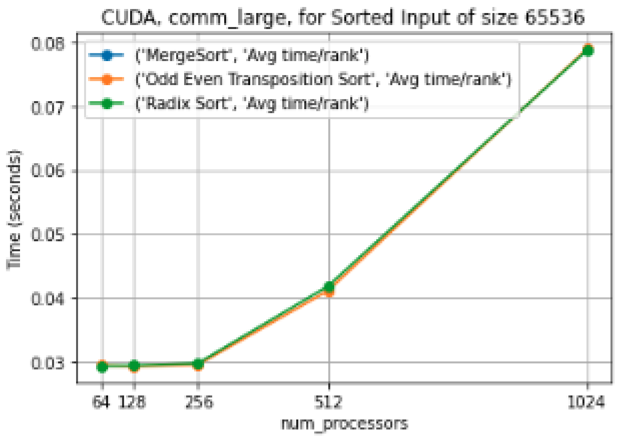
- **Comparison Plots**
  - MPI (comm, comp_large, main) - comparing sorted input of array size 2^18

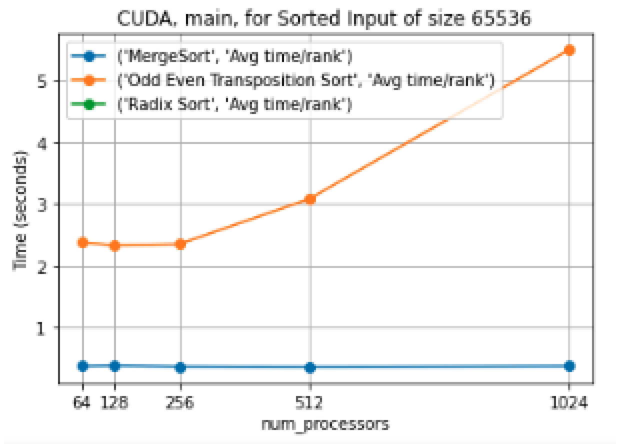


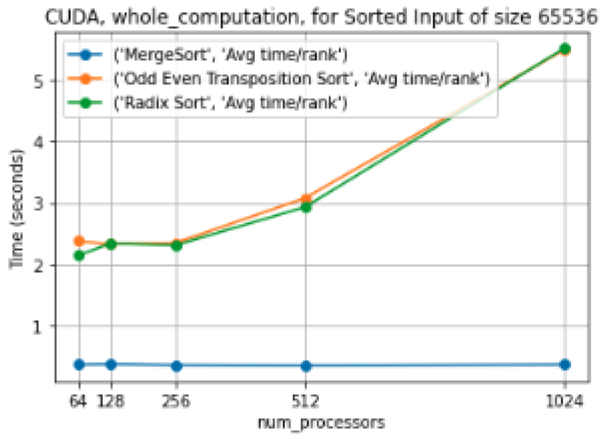




- CUDA (comm, main, whole_comp) - comparing sorted input of array size of 2^16




CUDA, main, for Sorted Input of size 65536
('MergeSort', 'Avg time/rank')
('Odd Even Transposition Sort', 'Avg time/rank')
('Radix Sort', 'Avg time/rank')
Time (seconds)
1
2
3
4
5
64
128
256
512
1024
num_processors


CUDA, whole_computation, for Sorted Input of size 65536
('MergeSort', 'Avg time/rank')
('Odd Even Transposition Sort', 'Avg time/rank')
('Radix Sort', 'Avg time/rank')
Time (seconds)
1
2
3
4
5
64
128
256
512
1024
num_processors