ParallelComputing CSCE 435 Group project

TAMU GitHub Repo

1. *due 10/27* Group members

1. Rohit Barichello
2. Christopher Tran
3. Vicente Trevino
4. Kelton Chesshire

---

2. *due 11/3* Project topic

We'll be implementing 3 different parallel sorting algorithms in MPI and CUDA and comparing them.

We'll also be measuring communication time, computation time, and how much data is sent, with varied inputs. Our inputs will include sorted, random, and reversed lists.

Our main communication method will be through Discord.

2. *due 11/3* Brief project description (what algorithms will you be comparing and on what architectures)

Source - Tutorials Point

1. Odd-Even Transposition Sort (MPI)

```
procedure ODD-EVEN_PAR (n)

begin
id := process's label

for i := 1 to n do
begin

    if i is odd and id is odd then
        compare-exchange_min(id + 1);
    else
        compare-exchange_max(id - 1);

    if i is even and id is even then
        compare-exchange_min(id + 1);
    else
        compare-exchange_max(id - 1);

end for

end ODD-EVEN_PAR
```

2. Enumeration Sort (MPI)

```
procedure ENUM_SORTING (n)

begin
for each process P1,j do
    C[j] := 0;

for each process Pi, j do

    if (A[i] < A[j]) or A[i] = A[j] and i < j) then
        C[j] := 1;
    else
        C[j] := 0;

for each process P1, j do
    A[C[j]] := A[j];

end ENUM_SORTING
```

3. Parallel Merge Sort (MPI)

```
procedure PARALLEL_MERGE_SORT(id, n, data, newdata)

begin
data = sequentialmergesort(data)

    for dim = 1 to n
        data = parallelmerge(id, dim, data)
    endfor

newdata = data
end PARALLEL_MERGE_SORT
```

4. Hyper Quick Sort (MPI)

```
procedure HYPER_QUICKSORT (B, n)

begin

id := process's label;

for i := 1 to d do
    begin
    x := pivot;
    partition B into B1 and B2 such that B1 ≤ x < B2;
    if ith bit is 0 then
```

```
        begin
            send B2 to the process along the ith communication link;
            C := subsequence received along the ith communication link;
            B := B1 U C;
        endif

        else
            send B1 to the process along the ith communication link;
            C := subsequence received along the ith communication link;
            B := B2 U C;
            end else
        end for

    sort B using sequential quicksort;

    end HYPER_QUICKSORT
```

---

3. *due 11/12* Pseudocode for each algorithm and implementation

Source - Tutorials Point

1. Odd-Even Transposition Sort (MPI)

   Source - selkie-macalester.org

   "Odd-Even Transposition Sort is based on the Bubble Sort technique. It compares two adjacent numbers and switches them, if the first number is greater than the second number to get an ascending order list. Odd-Even transposition sort operates in two phases − odd phase and even phase. In both the phases, processes exchange numbers with their adjacent number in the right."

   The Odd-Even Transposition Sort was implemented using MPI. The data being sorted is read from a binary file that contains *x* random numbers. The way this algorithm is written, on startup, each process is assigned a rank and based on this rank and the total number of processes, the processes each will read a chunk of the numbers from the binary file. The size of the chunk and the chunk each process reads is determined by their rank and the total number of processes.

   Now the actual Odd-Even portion of the sort occurs. Each process will sort locally its chunk of values using quick sort. Then based on whether it is the Odd phase or the Even phase, each process will pair up with a partner process to swap data is necessary. The swapping continues as many times as there are processes. This is to ensure that we swap all local process data if necessary and we end up with the final data as being sorted.

2. Enumeration Sort (MPI)

   The enumeration sort is a parallel sorting algorithm that comparesevery element of an array to every other element in the array. Each element is assigned a count. An element's count is incremented by one when compared to an element with a smaller value. At the end of the comparisons, each element is placed in an array at the index of its count. This algorithm indexes elements based on how many elements they're greater than.

This would usually be a naive approach to sorting. However, because we're leveraging multiple processes, the task is really only slightly less efficient than an O(n) algorithm.

The MPI implementation starts by having the main process send each element in the array to all other processes. Each subprocess then takes the value it recieves from the main process and compares it to the entire array using a for-loop. Each process has its count variable which is incremented as the comparison takes place.

An MPI Barrier is used to stall all processes until all comparison is done. Then, each process broadcasts it's count variable and element value. These pairs of data are compiled into a sorted list by the main process and printed.

3. Parallel Merge Sort (MPI)

   Parallel Merge Sort is a parallel version of the well-known mergesort algorithm. This algorithm assumes that the sequence to be sorted is distributed and so generates a distributed sorted sequence. To simplify things, we distribute the data evenly among the processors, and make sure that the processors are an integer power of two.

   In the MPI implementation, each process first reads from a binary file containing a predetermined amount of random numbers. Each process takes a specific chunk from the amount of random numbers assigned by rank, with the size of the chunk being determined by the rank and the total number of processes. Once this occurs, the merge begins. Each process performs a mergesort on their specific sections of the original data set, which is then combined back into a larger array on the primary process that is merged again. Then the sorted array is output to the console.

4. Hyper Quick Sort (MPI)

   Hyper Quick Sort is an implementation of the quick sort algorithm using parallelism. The fundamental of quicksort is choosing a value and partitioning the input data set to two subsets. One subset contains input data smaller in size than the chosen value and the other contains input data greater than the chosen value. This chosen value is called the pivot value. And in each step, these divided data sets are sub-divided choosing pivots from each set. Quicksort stop conditions are met when no sub division is possible.

   The Hyper Quick Sort algorithm was implemented using MPI. Initially, the data is read in from a binary file with a predetermined amount of random numbers. Once the data is stored within an array the head process will start quicksort with each process waiting to receive a subarray. Once the process receives a subarray it divides the array into smaller parts and distributes it to available processes. If there aren't any other available, then it sorts it sequentially. It then sends back the subarray.

3. *due 11/12* Evaluation plan - what and how will you measure and compare

We will compare the performance of 4 MPI sorting algorithms. We will be measuring not only the total computation/sorting time, but we will also consider things like the communication time, and how much data is sent between nodes/processes.

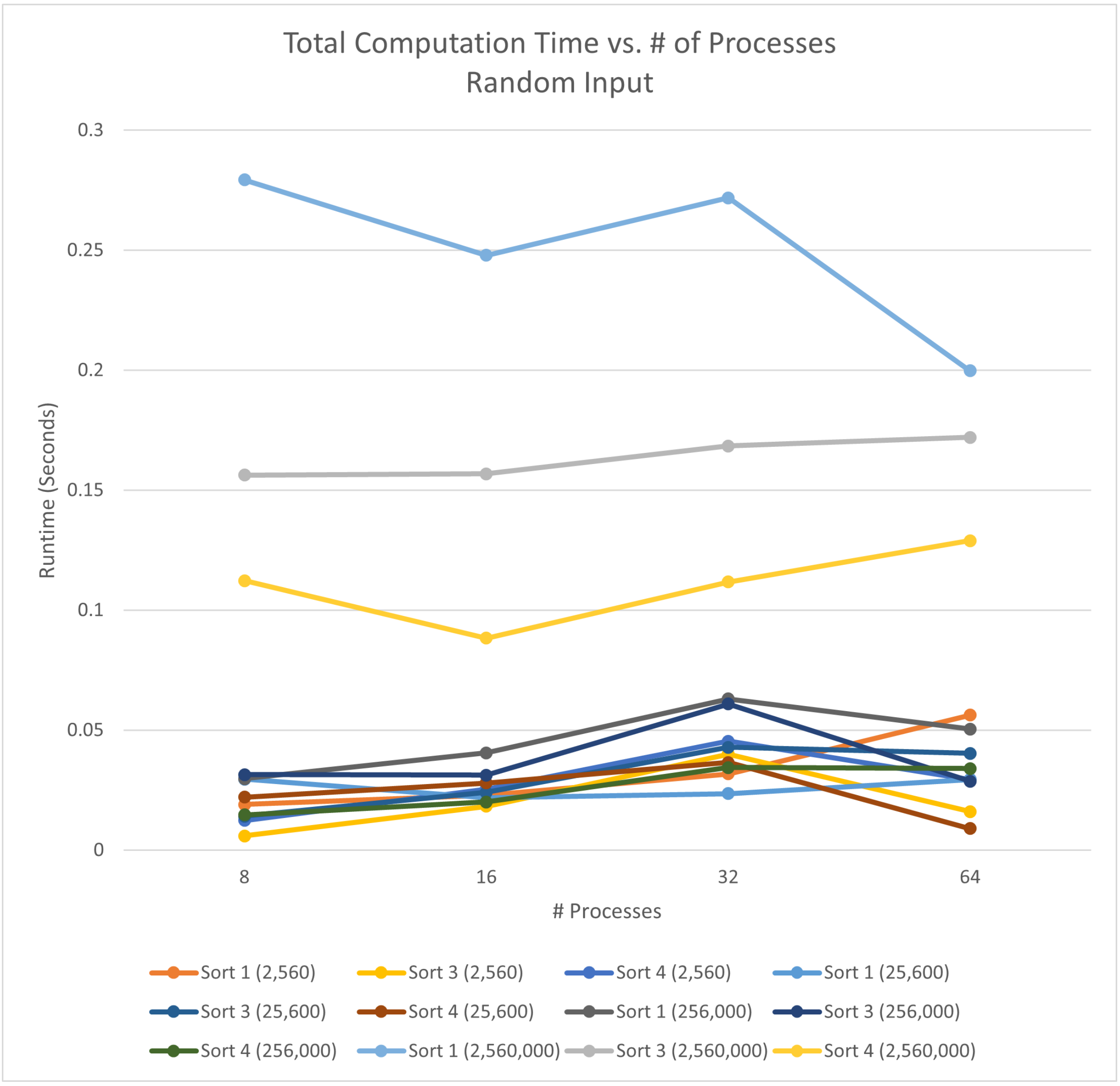
In addition to measuring these items with various size and random inputs, we will also be measuring with various sizes of sorted and reversed inputs.

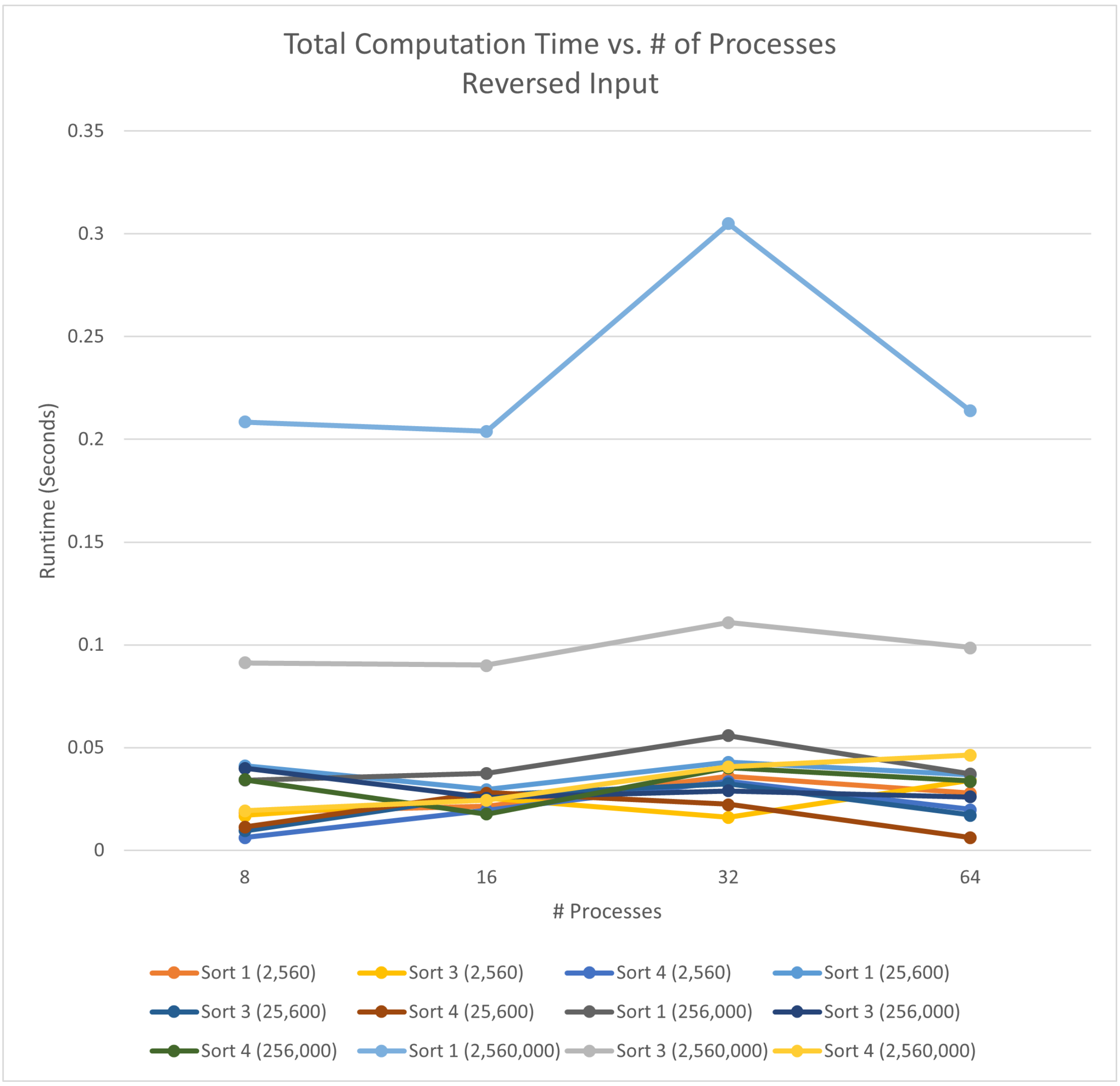
Another source of variation that we will be testing is varying the number of nodes/processes to look for either strong or weak scaling.

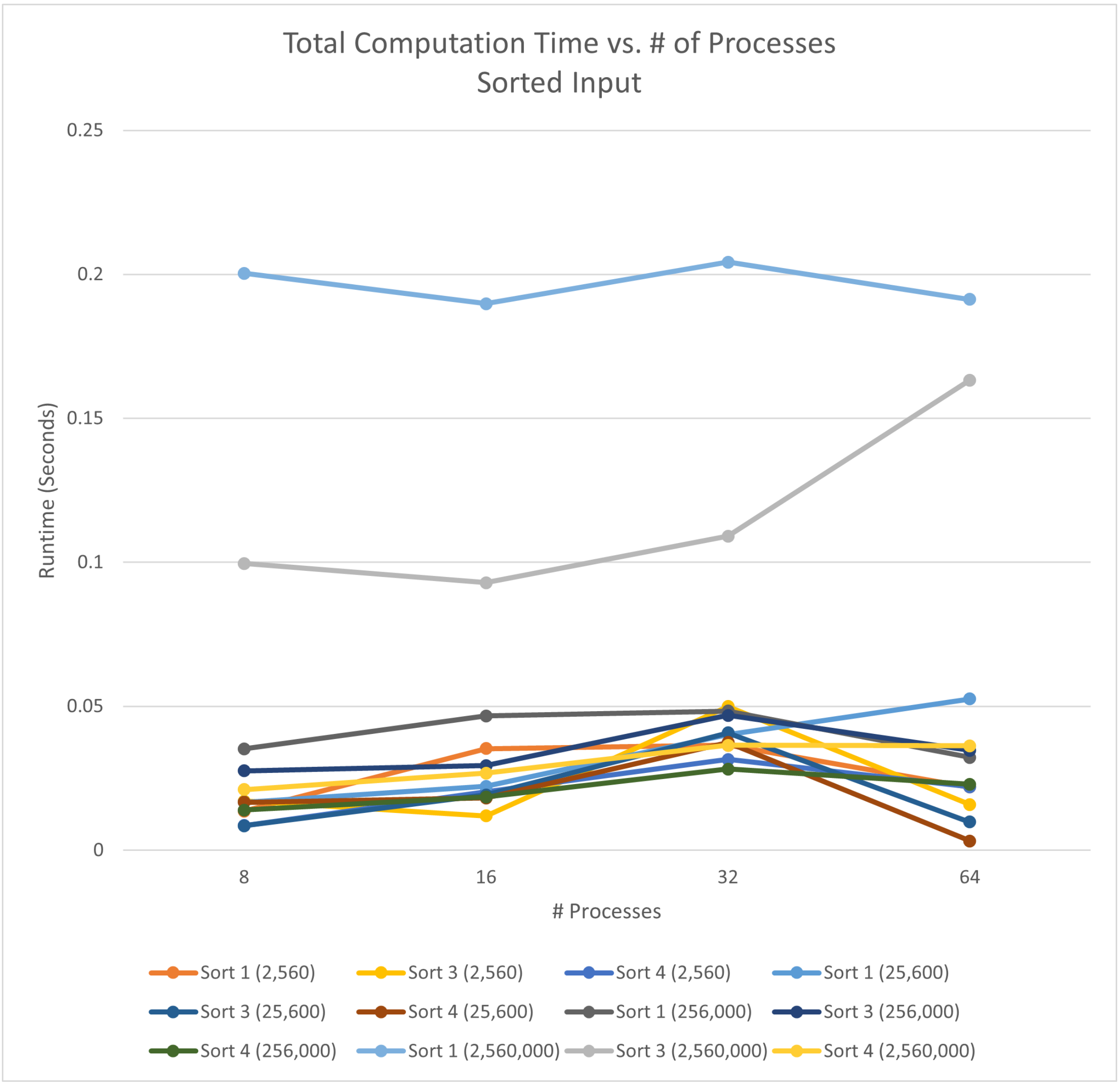
Overall we will be looking for trends in the sorting algorithms with various inputs and input sizes as well as with various numbers of nodes/processes.

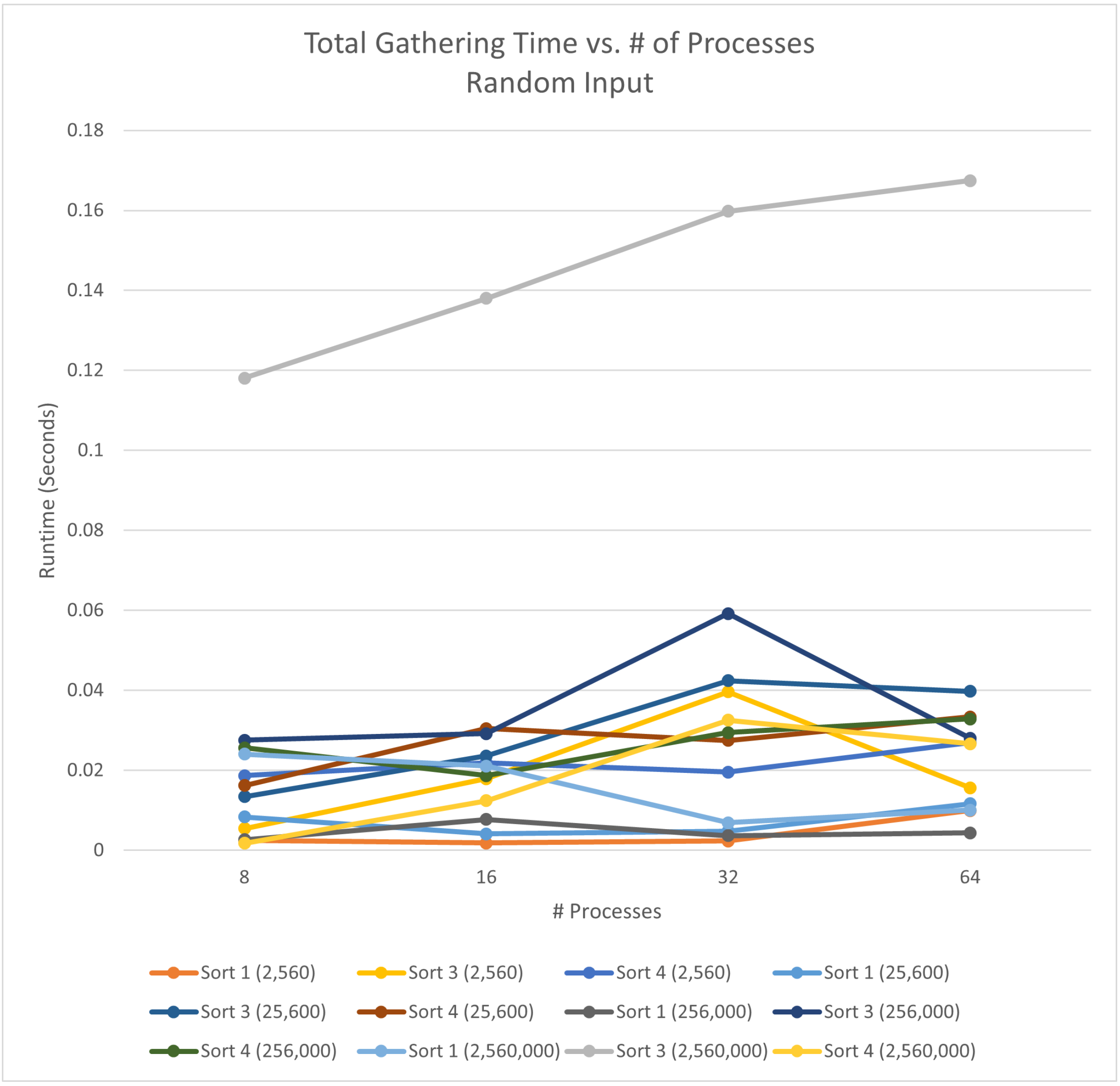
4. *due 11/19* Performance evaluation

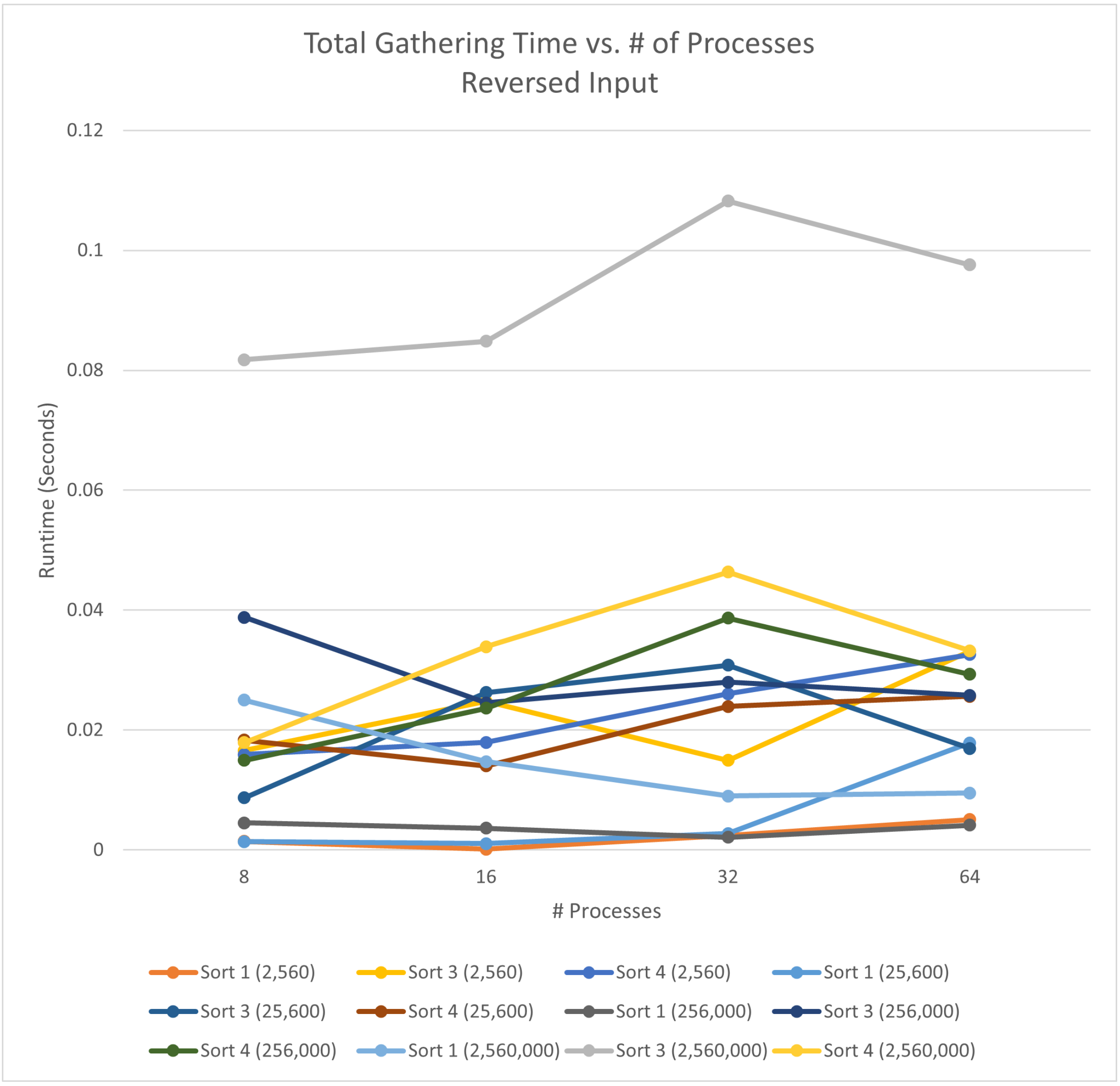
- Input sizes: 2,560; 25,600; 256,000; 2,560,000;
- Process sizes: 8; 16; 32; 64;
- Input types:
  - Random
  - Sorted
  - Reversed (Sorted)
- Measurement points:
  - Total computation time
    - Measure the total time it takes to ingest, sort, and gather sorted data.
  - Total data gathering time
    - Measure the total time it takes to gather sorted data.
  - Individual computation time (min, max, avg)
    - Measure the indvidual computation time of each process.
  - Individual data ingestion time (min, max, avg)
    - Measure the individual data ingestion time of each process. Or the time taken by each process to read in their respective data to be sorted.
- All Measurment Figures
  - Sort 1 is Odd-Even Transposition Sort, Sort 2 is Enumeration Sort, Sort 3 is Parallel Merge Sort, Sort 4 is Hyper Quick Sort
  - The figures do NOT include the information for "Sort 2" aka "Enumeration Sort." This is because data collection from enumeration sort required the use of values where the number of processes were equal to the number of values being sorted. This is attributed simply to the nature of how the algorithm operates.

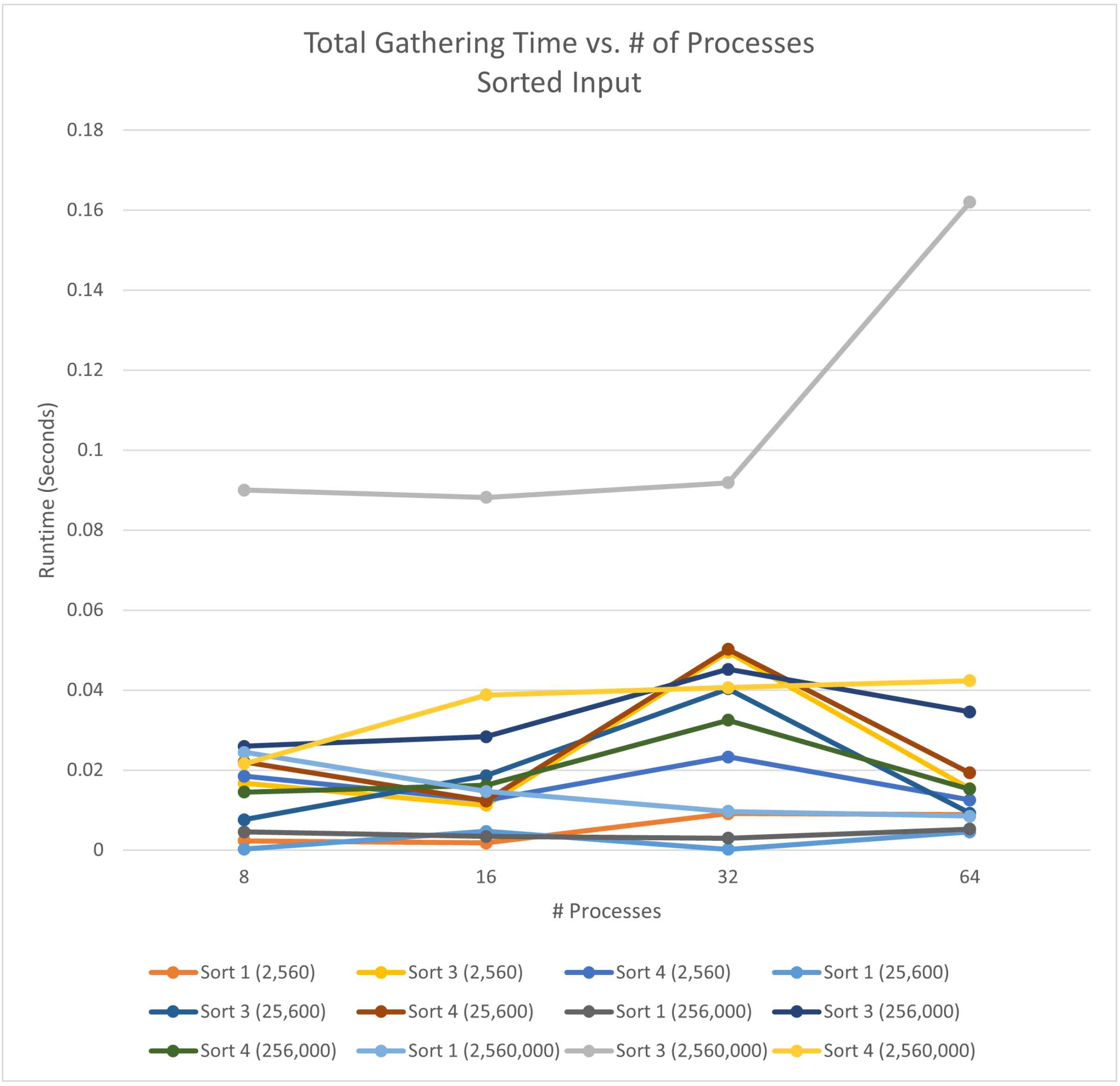
- 
Total Computation Time vs. # of Processes
Random Input
Runtime (Seconds)
0.3
0.25
0.2
0.15
0.1
0.05
0
8
16
32
64
# Processes
Sort 1 (2,560)
Sort 3 (2,560)
Sort 4 (2,560)
Sort 1 (25,600)
Sort 3 (25,600)
Sort 4 (25,600)
Sort 1 (256,000)
Sort 3 (256,000)
Sort 4 (256,000)
Sort 1 (2,560,000)
Sort 3 (2,560,000)
Sort 4 (2,560,000)

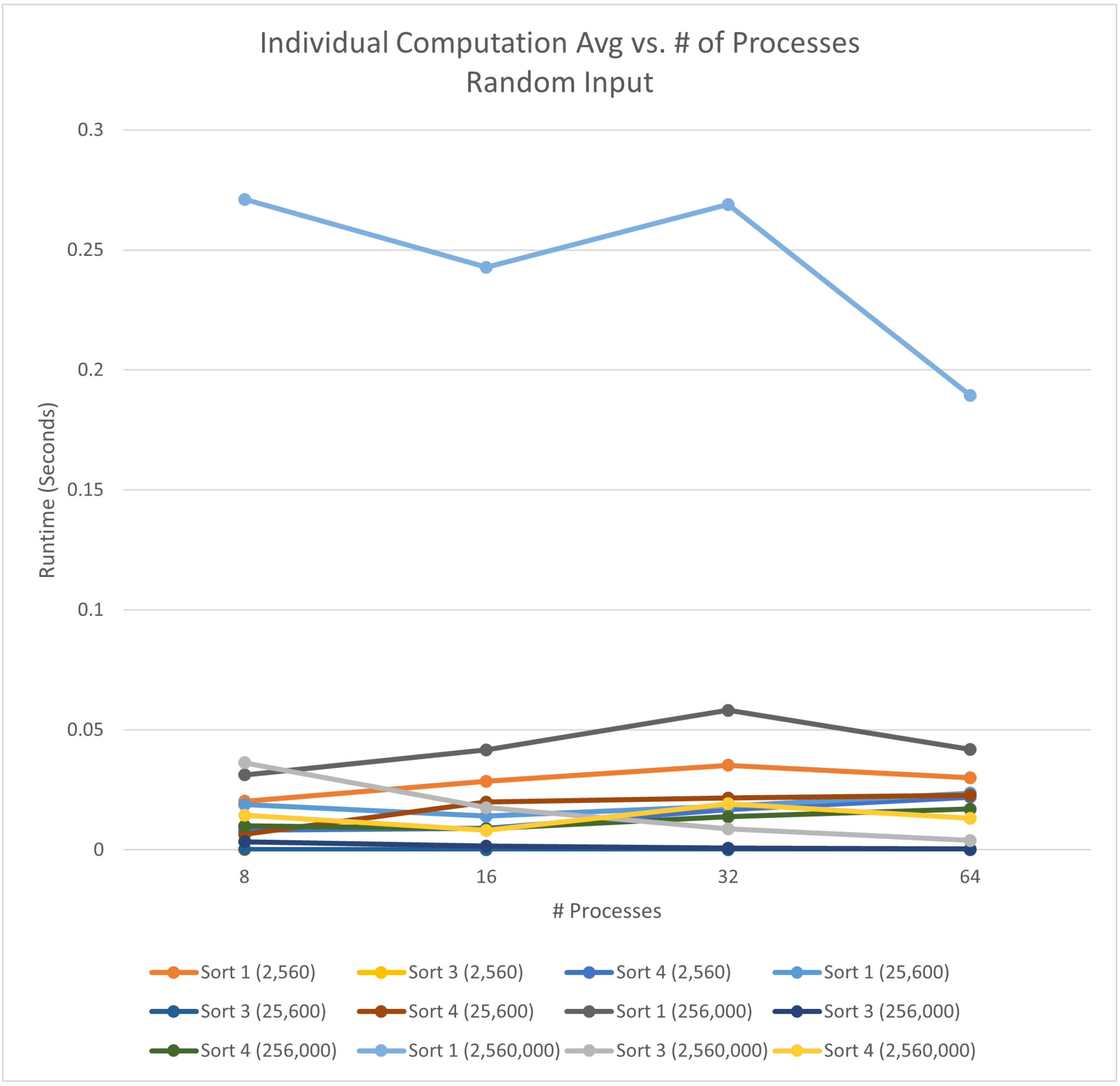
- 
Total Computation Time vs. # of Processes
Reversed Input
Runtime (Seconds)
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
8
16
32
64
# Processes
Sort 1 (2,560)
Sort 3 (2,560)
Sort 4 (2,560)
Sort 1 (25,600)
Sort 3 (25,600)
Sort 4 (25,600)
Sort 1 (256,000)
Sort 3 (256,000)
Sort 4 (256,000)
Sort 1 (2,560,000)
Sort 3 (2,560,000)
Sort 4 (2,560,000)

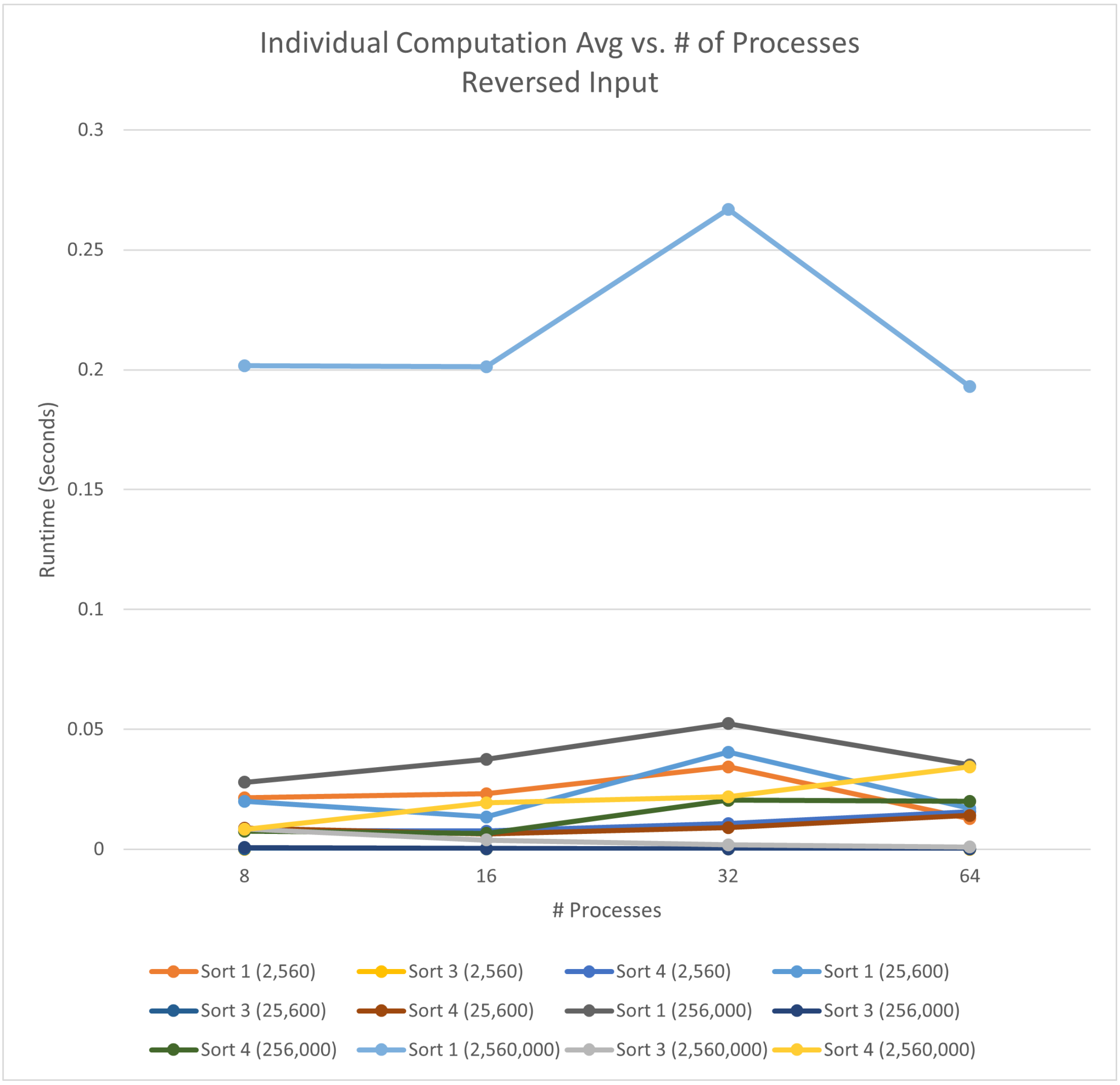
- 
Total Computation Time vs. # of Processes
Sorted Input
Runtime (Seconds)
0.25
0.2
0.15
0.1
0.05
0
8
16
32
64
# Processes
Sort 1 (2,560)
Sort 3 (2,560)
Sort 4 (2,560)
Sort 1 (25,600)
Sort 3 (25,600)
Sort 4 (25,600)
Sort 1 (256,000)
Sort 3 (256,000)
Sort 4 (256,000)
Sort 1 (2,560,000)
Sort 3 (2,560,000)
Sort 4 (2,560,000)

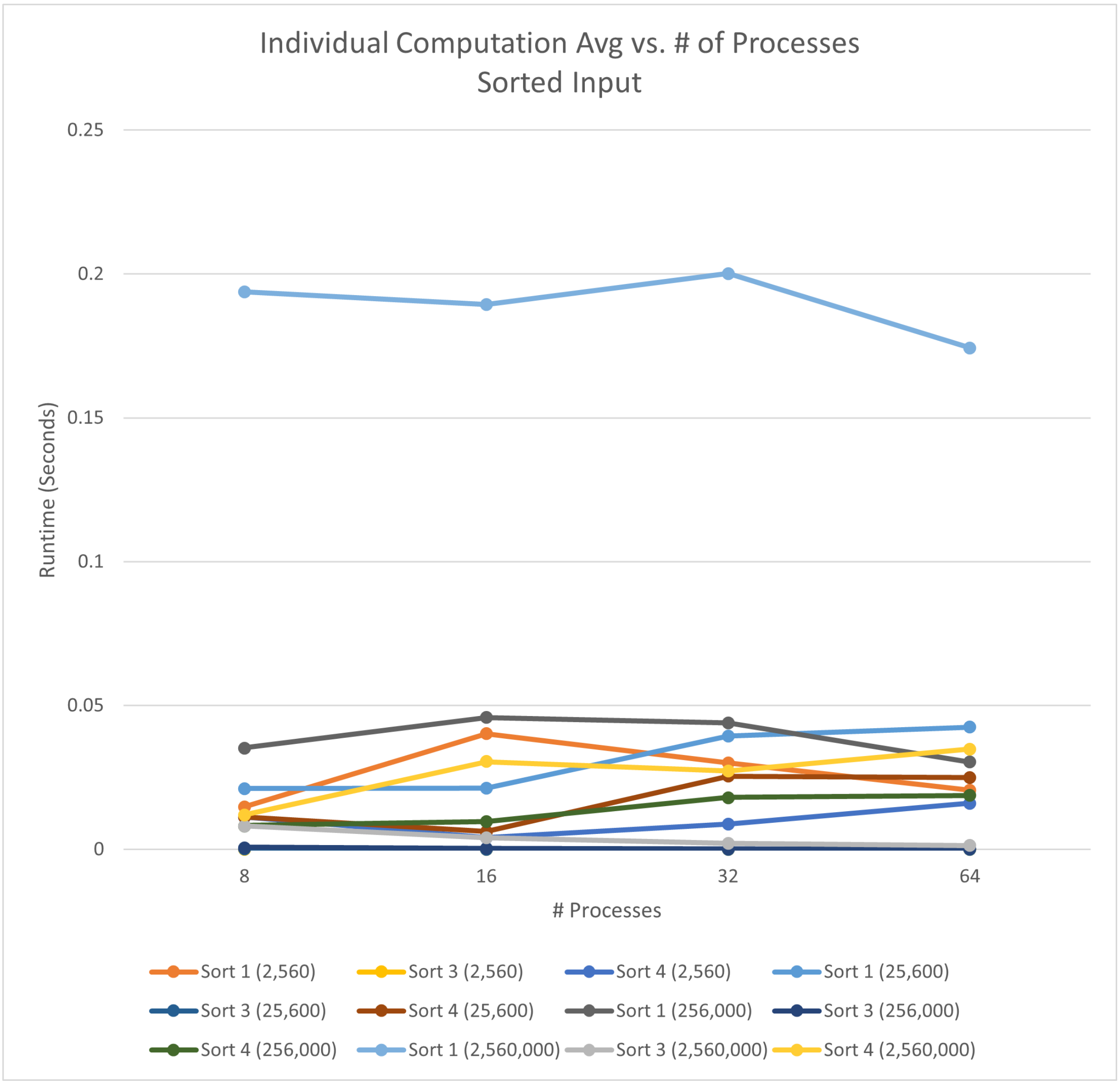
- 
Total Gathering Time vs. # of Processes
Random Input
Runtime (Seconds)
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
8
16
32
64
# Processes
Sort 1 (2,560)
Sort 3 (2,560)
Sort 4 (2,560)
Sort 1 (25,600)
Sort 3 (25,600)
Sort 4 (25,600)
Sort 1 (256,000)
Sort 3 (256,000)
Sort 4 (256,000)
Sort 1 (2,560,000)
Sort 3 (2,560,000)
Sort 4 (2,560,000)

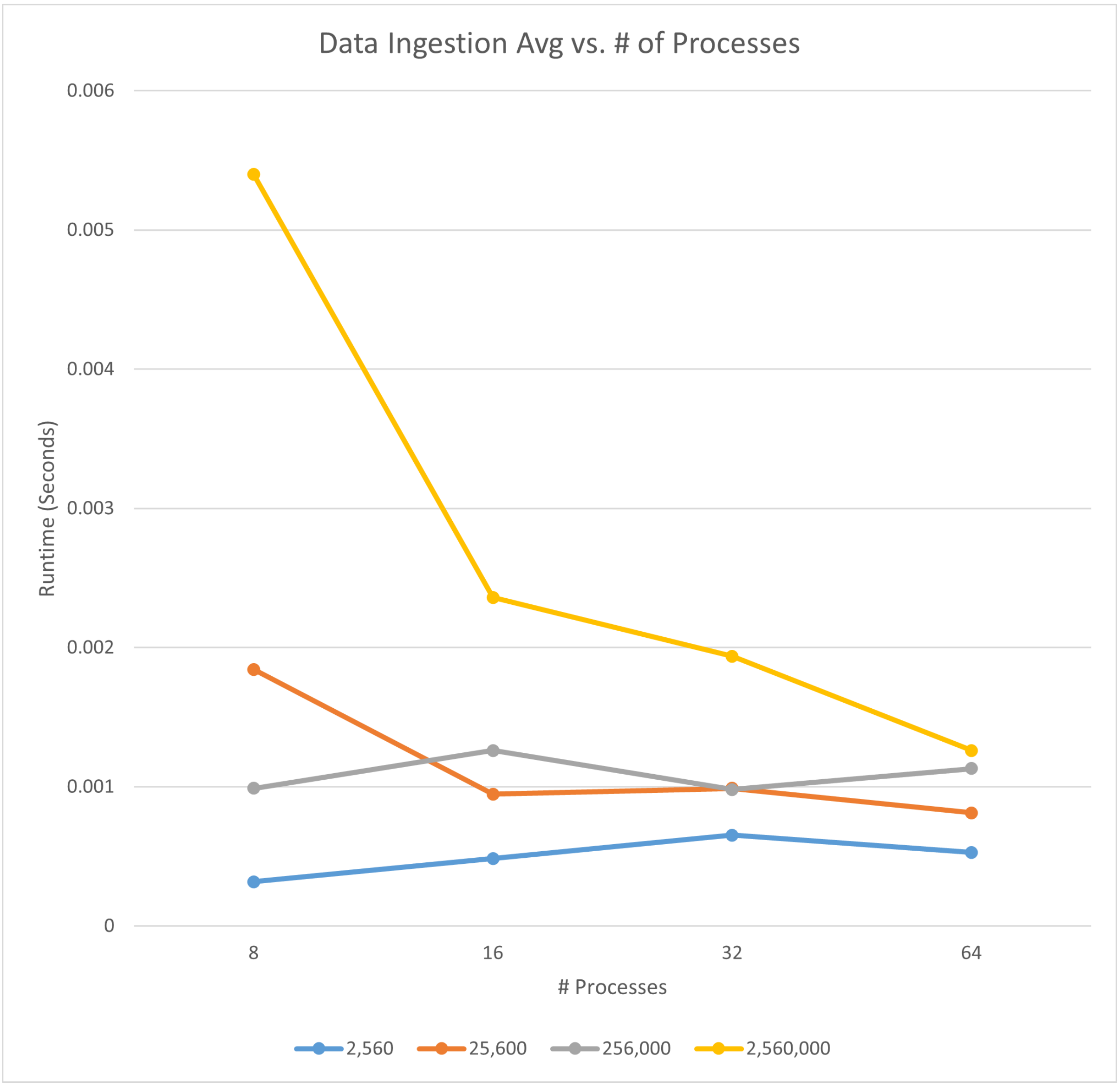
- 
Total Gathering Time vs. # of Processes
Reversed Input
Runtime (Seconds)
0.12
0.1
0.08
0.06
0.04
0.02
0
8
16
32
64
# Processes
Sort 1 (2,560)
Sort 3 (2,560)
Sort 4 (2,560)
Sort 1 (25,600)
Sort 3 (25,600)
Sort 4 (25,600)
Sort 1 (256,000)
Sort 3 (256,000)
Sort 4 (256,000)
Sort 1 (2,560,000)
Sort 3 (2,560,000)
Sort 4 (2,560,000)

- 
Total Gathering Time vs. # of Processes
Sorted Input
Runtime (Seconds)
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
8
16
32
64
# Processes
Sort 1 (2,560)
Sort 3 (2,560)
Sort 4 (2,560)
Sort 1 (25,600)
Sort 3 (25,600)
Sort 4 (25,600)
Sort 1 (256,000)
Sort 3 (256,000)
Sort 4 (256,000)
Sort 1 (2,560,000)
Sort 3 (2,560,000)
Sort 4 (2,560,000)

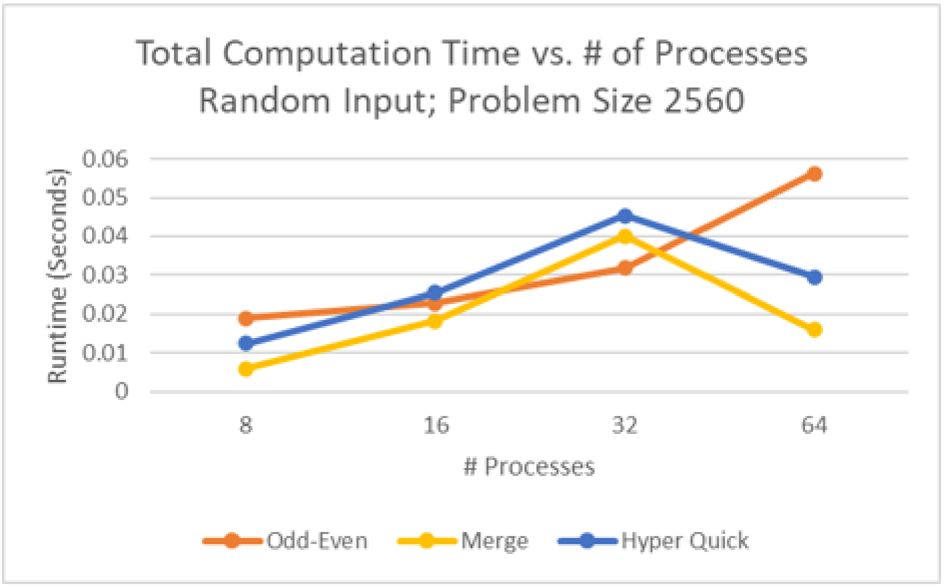
- 
Individual Computation Avg vs. # of Processes
Random Input
Runtime (Seconds)
0.3
0.25
0.2
0.15
0.1
0.05
0
8
16
32
64
# Processes
Sort 1 (2,560)
Sort 3 (2,560)
Sort 4 (2,560)
Sort 1 (25,600)
Sort 3 (25,600)
Sort 4 (25,600)
Sort 1 (256,000)
Sort 3 (256,000)
Sort 4 (256,000)
Sort 1 (2,560,000)
Sort 3 (2,560,000)
Sort 4 (2,560,000)

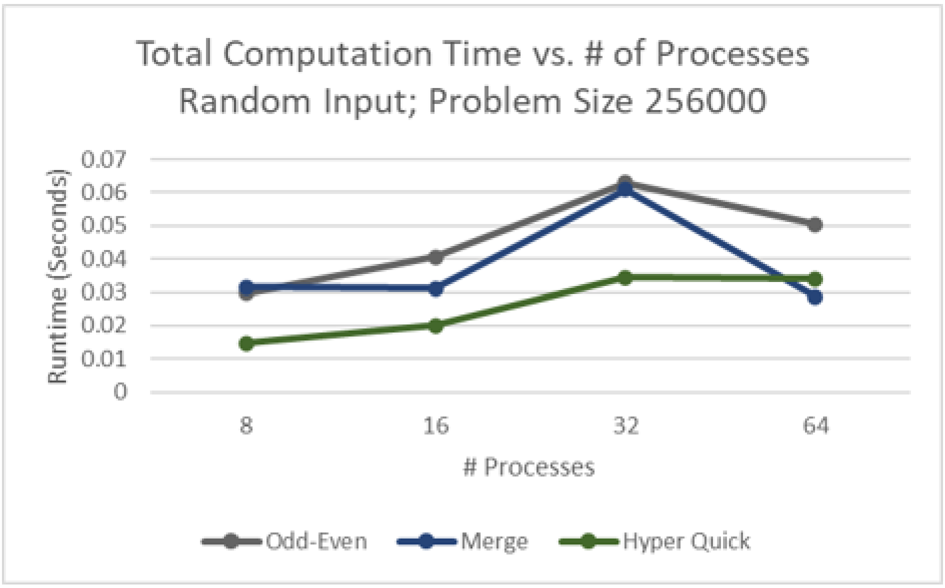
- 
Individual Computation Avg vs. # of Processes
Reversed Input
Runtime (Seconds)
0.3
0.25
0.2
0.15
0.1
0.05
0
8
16
32
64
# Processes
Sort 1 (2,560)
Sort 3 (2,560)
Sort 4 (2,560)
Sort 1 (25,600)
Sort 3 (25,600)
Sort 4 (25,600)
Sort 1 (256,000)
Sort 3 (256,000)
Sort 4 (256,000)
Sort 1 (2,560,000)
Sort 3 (2,560,000)
Sort 4 (2,560,000)

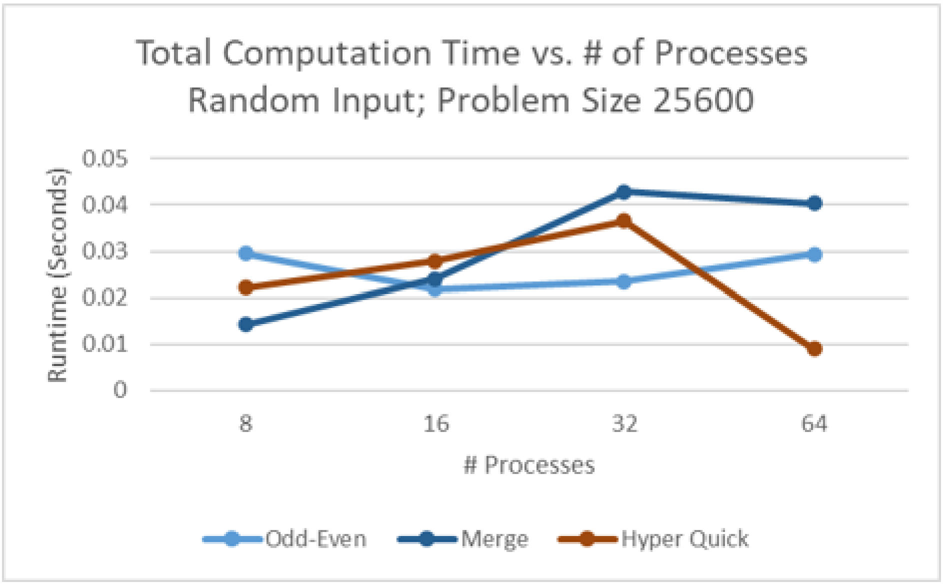
- 
Individual Computation Avg vs. # of Processes
Sorted Input
Runtime (Seconds)
0.25
0.2
0.15
0.1
0.05
0
8
16
32
64
# Processes
Sort 1 (2,560)
Sort 3 (2,560)
Sort 4 (2,560)
Sort 1 (25,600)
Sort 3 (25,600)
Sort 4 (25,600)
Sort 1 (256,000)
Sort 3 (256,000)
Sort 4 (256,000)
Sort 1 (2,560,000)
Sort 3 (2,560,000)
Sort 4 (2,560,000)

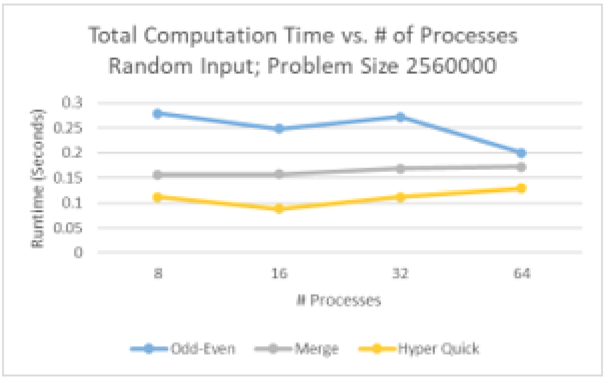
- 

  - **Data Ingestion Time Avg is measured as the avg time each indvidual process took to read in their data**

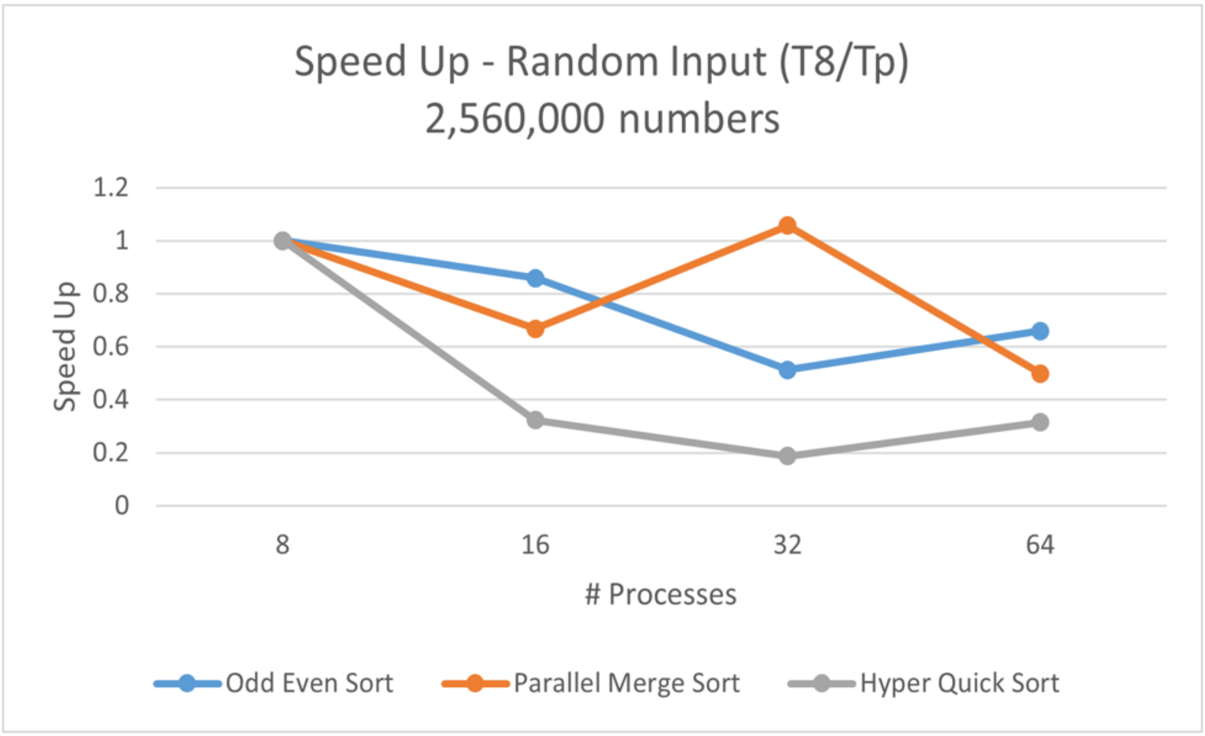
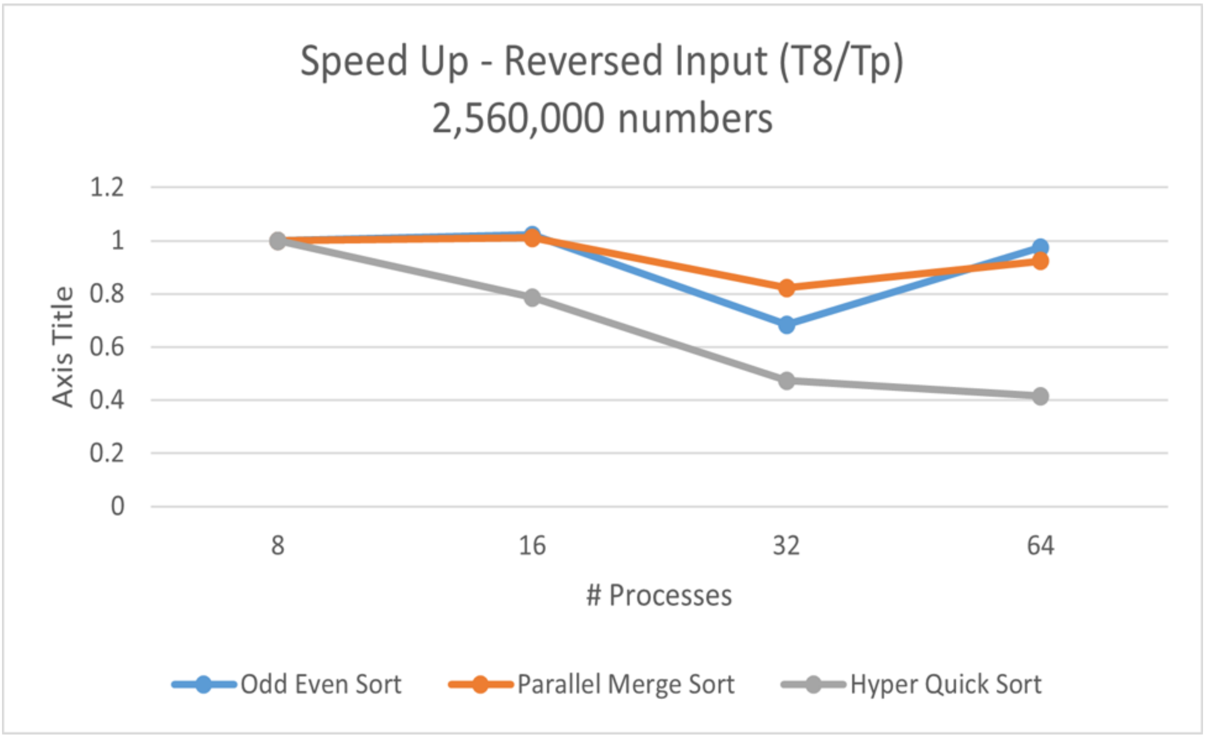
- Total Computation Time Comparison Figures
  - 

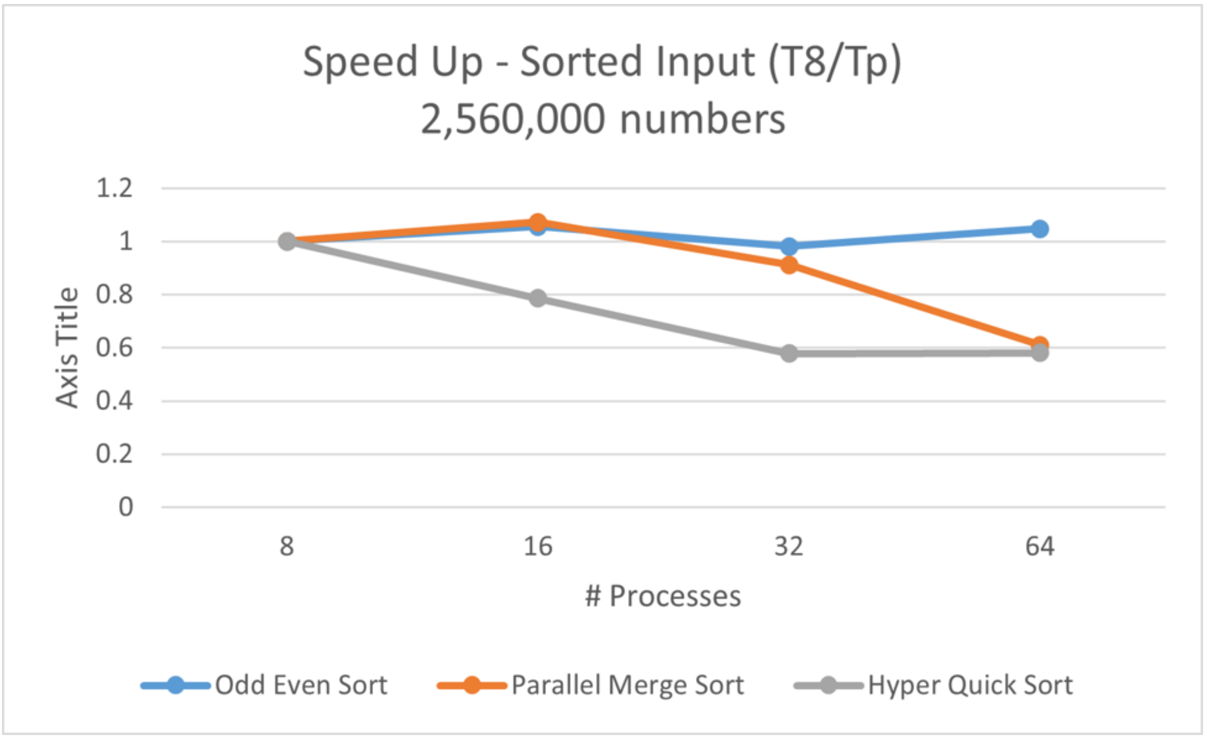
- 

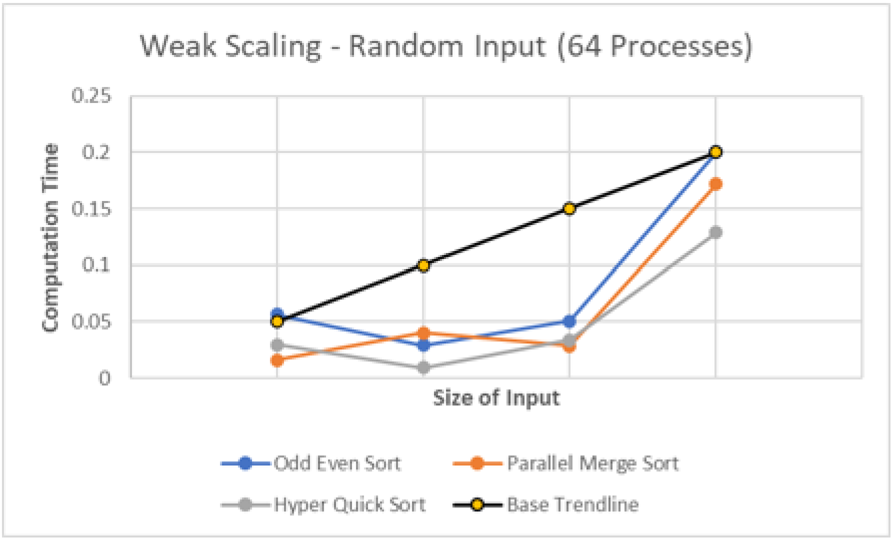
- 

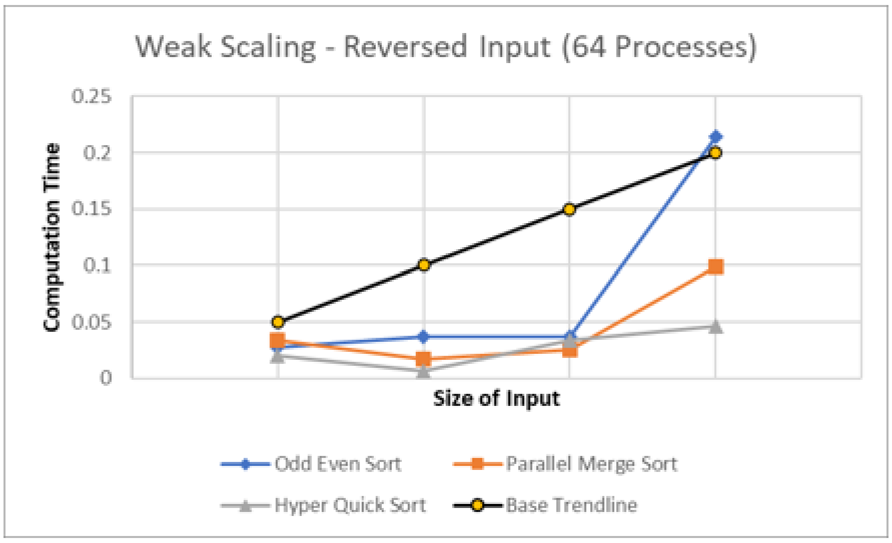
- 

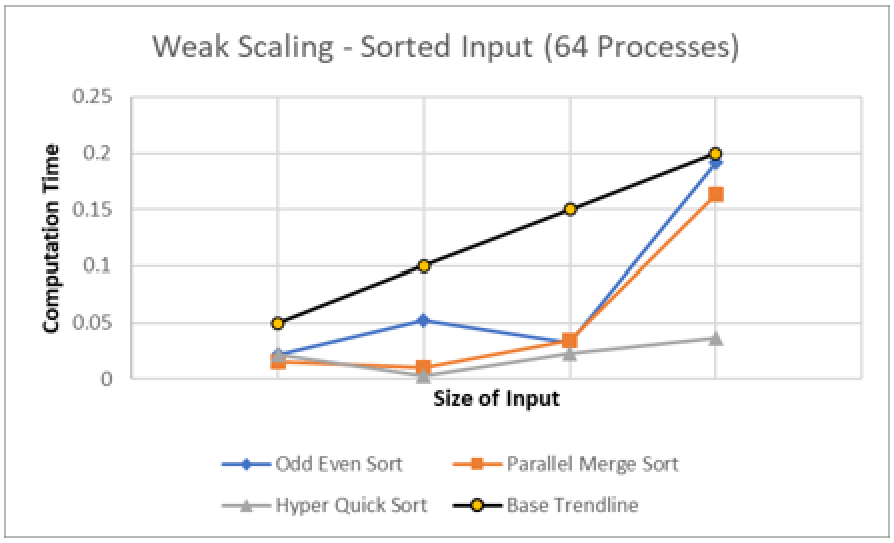
- Strong Scaling Speedup Figures
    - There was limited evidence of strong scaling that was able to be found with any of the implemented sorting algorithms. This is most likely due to the fact that we were unable to get larger sets of data to be sorted due to the limitation of how we chose to create and ingest the data.

- 
Speed Up - Random Input (T8/Tp)
2,560,000 numbers
Speed Up
1.2
1
0.8
0.6
0.4
0.2
0
8
16
32
64
# Processes
Odd Even Sort
Parallel Merge Sort
Hyper Quick Sort
- 
Speed Up - Reversed Input (T8/Tp)
2,560,000 numbers
Axis Title
1.2
1
0.8
0.6
0.4
0.2
0
8
16
32
64
# Processes
Odd Even Sort
Parallel Merge Sort
Hyper Quick Sort

- 

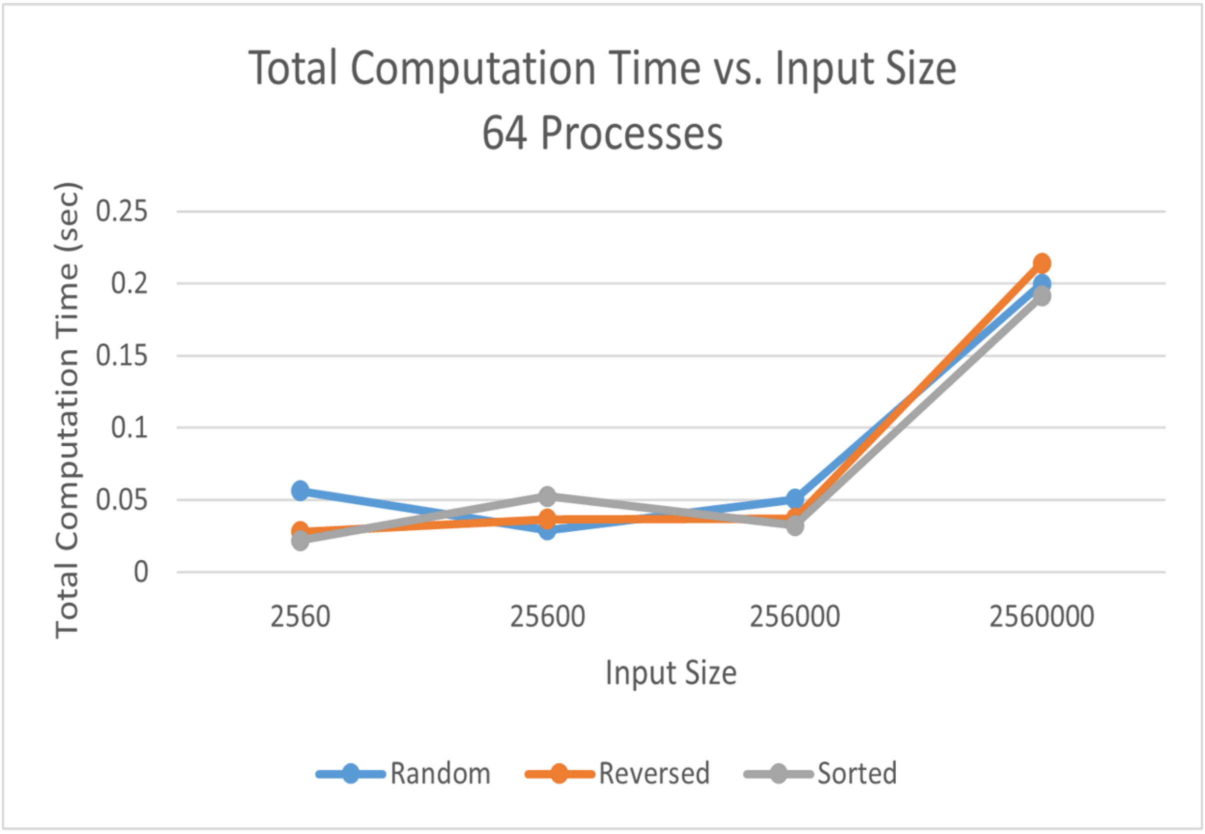
- Weak Scaling Speedup Figures
  - 

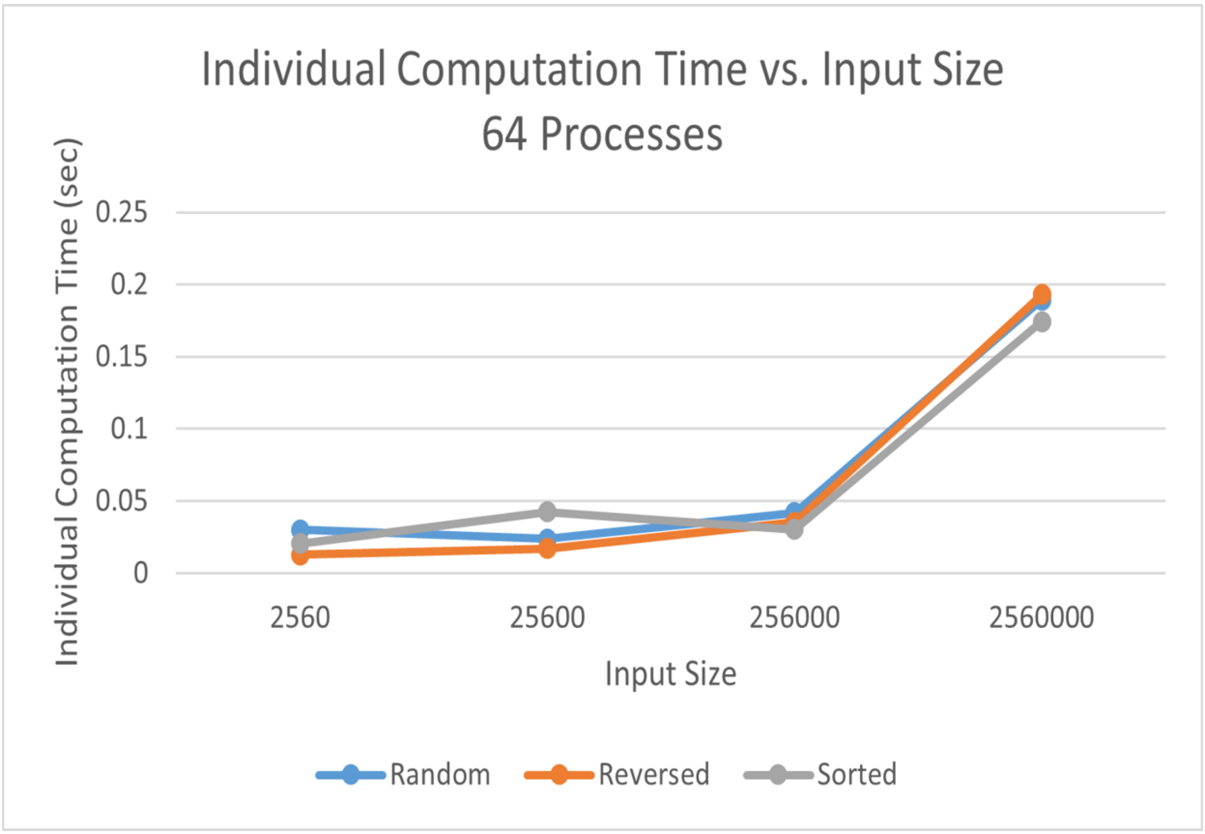
  - 

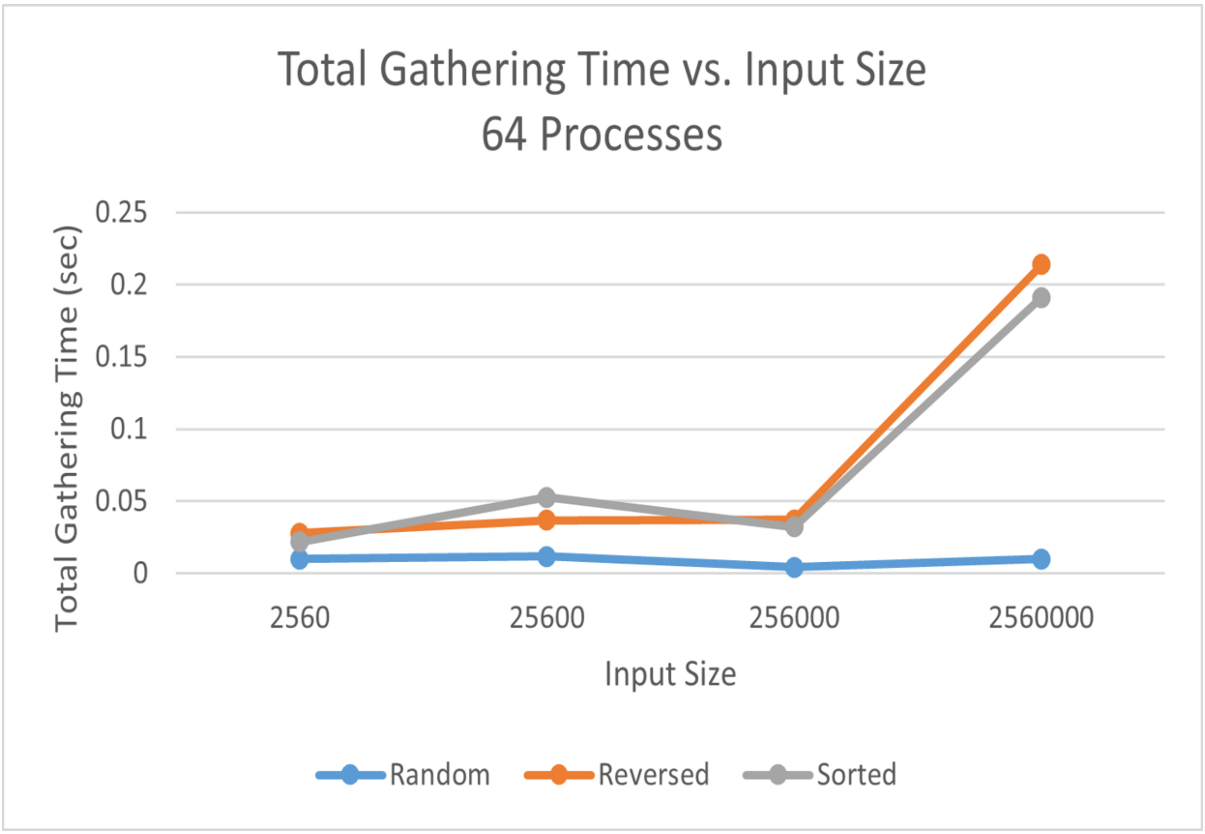
- 

1. Odd-Even Transposition Sort (MPI)

   The Odd-Even Transposition Sort performed comparably to the Enumeration Sort as opposed to the Parallel Merge Sort and the Hyper Quick Sort. This sort saw increases in total computation time as the number of processes increased for a fixed problem set. This means it doesn't exhibit strong scaling. This can be mainly attributed to the communication overhead between processes. This is because according to the algorithm, between each odd or even "phase" each process must communicate with one another the necessary information on whether or not a swap should be made between them. There was also the overhead that it takes to swap values between processes. Similarly to all the other sorts, this sort suffered from the communication overhead of gathering the sorted data at the end as well because a single process was handling the gathering of data from multiple processes. Additionally, individual process times were decreased as the # of processes increase as expected because each process has to handle much less values.

   - 

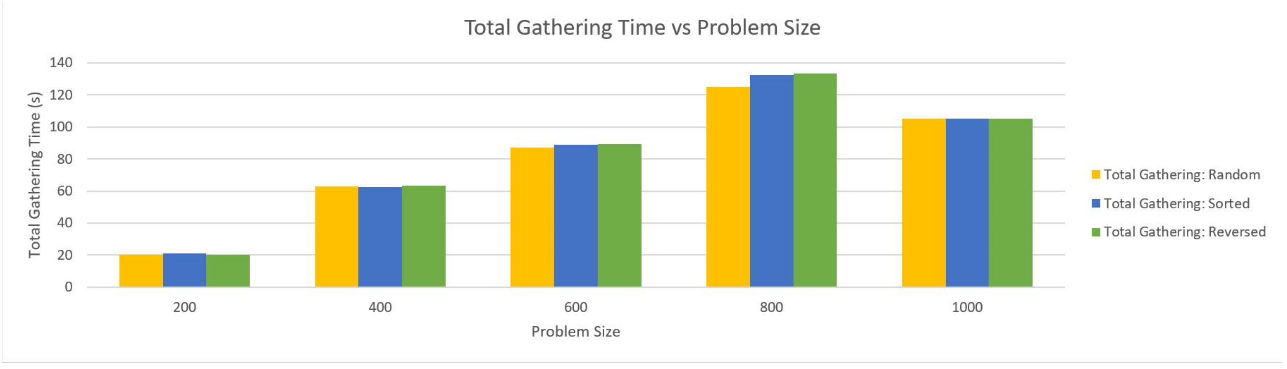
- 

- 

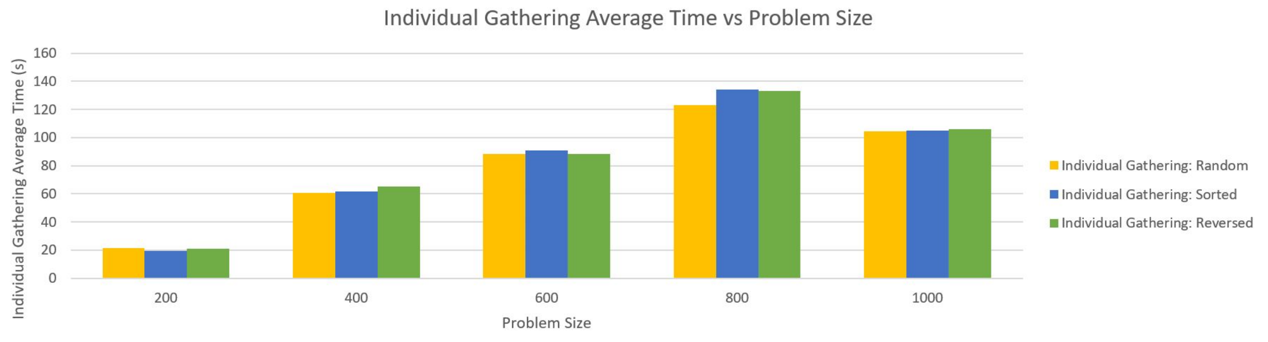
2. Enumeration Sort (MPI)

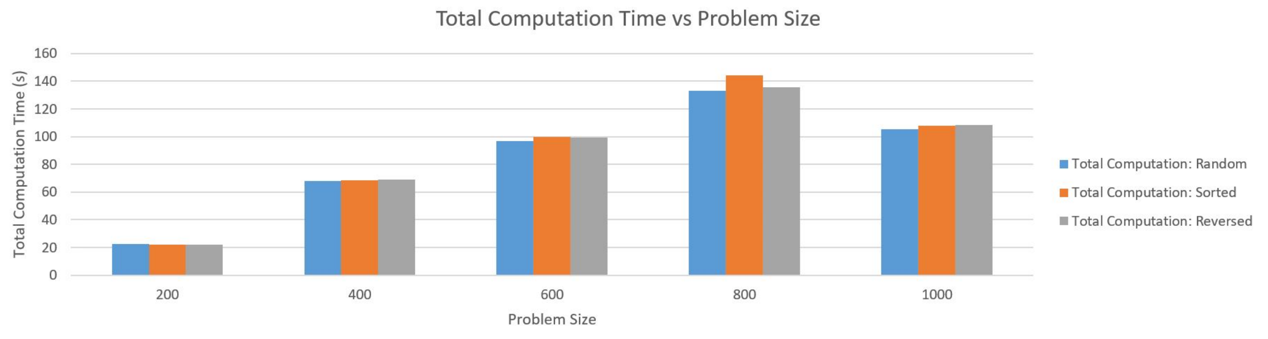
   For the enumeration sort, we were only able to demonstrate weak scaling due to the way the algorithm is implemented. For a given problem size, the algorithm, implemented as it is in this project, must have the same number of processes. This only allows for varying the number of processors for a fixed problem size.

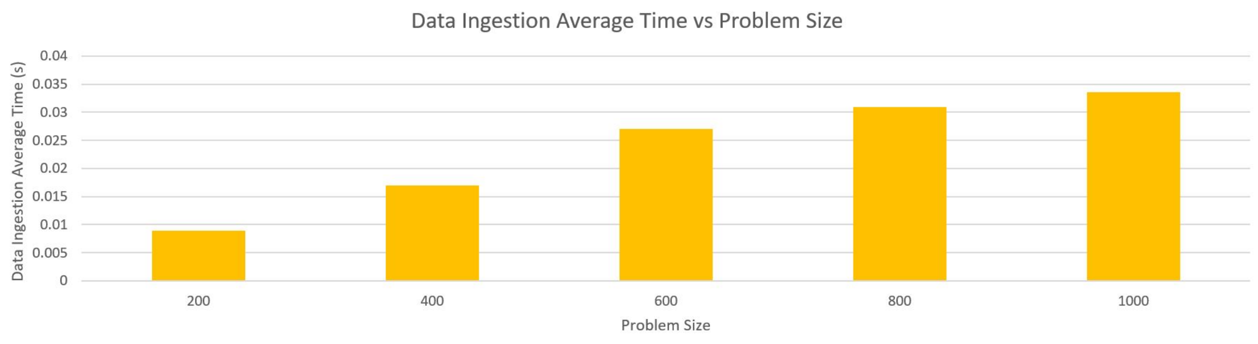
The execution times for the enumeration sort were much higher than the other three sorts in this project despite smaller problem sizes. This is because the algorithm has to leverage extensive blocking to make sure that the main process doesn't broadcast too early. This algorithm also followed more of an arc in execution time than the other algorithms. The other ones sharply increased in execution time with problem size, but the enumeration sort saw a decline after a certain problem size. This was only for weak scaling since the enum sort doesn't have applications with regards to strong scaling or GPU performance.



This first figure demonstrates a steady increase in the time taken to gather all the broadcasted data from the back part of the algorithm, with a decline past 800. This is likely due to the fact that even though we have more overhead from the processes and the problem size the algorithm has more similar values to compare to due to the increase in spread of the initial random data array. This allows for small speedups throughout the comparison phase of the algorithm. The three input types demonstrated similar performance, indicating no real difference in operation. This is most likely due to the fact that the enumeration sort is a naive algorithm that runs in O(k*n) time no matter the input.







We can see from the slight downward linear shift in execution time of the individual gathering time from the overall gathering time that the individual gathering time represented the majority of the computation time in the program. This is because multiple blocking broadcast statements had to be made in addition to a second O(n) pass through the original array.

3. Parallel Merge Sort (MPI)

   The Parallel Merge Sort performed comparably to the Hyper Quick Sort as opposed to the Odd-Even Sort and the Enumeration Sort. This sort saw decreases in total computation time as the number of processes increased for a fixed problem set. This means it exhibits strong scaling. However, the gather time increased as the number of processes increased as expected, since a single process has to manage gathering sorted data from all other processes. This was a bottleneck in the algorithm. This caused the runtimes of higher # of processes to actually be higher than the runtime of those with lower # of processes as the communication overhead during the gathering phase became very significant. Additionally, individual process times were decreased as the # of processes increase as expected because each process has to handle much less values.

4. Hyper Quick Sort (MPI)

   The Hyper Quick Sort performed comparably to the Parallel Merge Sort as opposed to the Odd-Even Sort and the Enumeration Sort. This sort saw decreases in total computation time as the number of processes increased for a fixed problem set. This means it exhibits strong scaling. However, the gather time increased as the number of processes increased as expected, since a single process has to manage gathering sorted data from all other processes. This was a bottleneck in the algorithm. This caused the runtimes of higher # of processes to actually be higher than the runtime of those with lower # of processes as the communication overhead during the gathering phase became very significant. Additionally, individual process times were decreased as the # of processes increase as expected because each process has to handle much less values.

5. *due 12/1* Presentation, 5 min + questions

Presentation Preview

6. *due 12/8* Final Report

Report Preview