CSCE 435 Group project

0. Group number: 20

1. Group members:

1. Victoria Chen
2. Arielle Shaver
3. Victoria Chiang
4. Bonnie Wu

We are using a group chat over text to communicate.

2. Project topic (e.g., parallel sorting algorithms)

Parallel Sorting Algorithms

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

- Bitonic Sort (Bonnie):
- Sample Sort (Arielle):
- Merge Sort (Victoria):
- Radix Sort (Vic):

2b. Pseudocode for each parallel algorithm

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

```
// Initialize MPI
MPI_Init()
Get the number of processes (n_procs)
Get the rank of each process (rank)

// Main process (rank 0), read and distribute the data
if rank == 0 then
    Read the input array
    Split the input data into n_procs parts
end if

// Scatter the data to all processes given the main proc
MPI_Scatter(data, local_data, root=0)

// For each digit LSD to MSD:
for each digit in LSD to MSD do
    // Count the frequency of each digit in the local dc
    local_histogram = calculate_histogram(local_data, di

    // Share and combine all local histograms to create
    global_histogram = MPI_Allreduce(local_histogram, MF

    // Use the global histogram to figure out where datc
    prefix_sum = calculate_prefix_sum(global_histogram)

    // Exchange data between processes based on prefix_s
    MPI_Alltoallv(local_data, send_counts, recv_counts,

    // Update local data with the received sorted data
    local_data = sorted_data
end for

// Gather all the sorted data back to process 0
MPI_Gather(local_data, sorted_array, root=0)

// Finalize MPI (clean up and end the program)
MPI_Finalize()
```

- Merge Sort:

Main:

1. Initialize MPI - MPI_Init()
    a. MPI_INIT(&argc, &argv)
2. Rank and size of MPI COMM_WORLD
    a. MPI_Comm_rank, MPI_Comm_size
3. If rank == 0 then: read in the input array, s
    a. Use a fixed array for sorting so it is th
    b. no variable is our size of array
4. Broadcast that size of array to all the proce
    a. MPI_Bcast(no, 1, MPI_INT, 0, MPI_COMM_WOF
5. Find the size of each subarray for each of th
6. Allocate a memory for each process's subarray
7. Scatter the original array into all the subar
    a. MPI_Scatter()
8. Allocate temp array for merging with subSize
9. Every process will perform merge sort on its
    a. mergeSort(subarray, temparray, 0, subSize
10. Root process - gathers all the sorted subarr
    a. World rank is 0, we want to make a sortec
11. Perform MPI_Gather to grab all the sorted su
12. Final merge with the root process so world r
13. Add a print statement that prints out the sc
14. Free any memory and finalize the MPI enviror

Merge Function: * merge two sorted halves of one arr

1. Set lh= to the start index to the left half,
    set rh= to the starting index of the right h
2. While loop to start merging and compare the e
    While (lh <= ending index of first half && r
        a. If current element array[lh] is <= to
        b. Else we want to do the opposite so ar
3. Remaining elements will be copied:
    a. If there are elements in the right half l
    b. Check the index and the end of side index
4. Copy the sorted elements from the temp array
    a. using a for loop and the indexes

MergeSort Function: *recursive merge sort algorithm

1. Check if the starting index is less than the
    has more than one element
        a. Find the middle index: mid = (LI + RI
        b. Recursively call mergeSort so it can
        c. Recursively call mergeSort so it can
        d. Merge Function called so we can combi

- Sample Sort:

```
Main Function:

    1. Get user input for data_type, size (total number

    2. // Initialize MPI
       MPI_Init()  // Set up MPI environment for paralle
       task_id = MPI_Comm_rank(MPI_COMM_WORLD)  // Get r
       num_tasks = MPI_Comm_size(MPI_COMM_WORLD)  // Get

    3. // Check if there are enough tasks available
       If num_tasks < 2:
           Print "Need at least two MPI tasks. Quitting.
           MPI_Abort(MPI_COMM_WORLD, error_code = -1)
           Exit program

    4. Set the number of buckets (m) = num_tasks  // m i

    5. // Synchronize all processes
       MPI_Barrier(MPI_COMM_WORLD)

    6. // Master Process
       If task_id == MASTER:
           Print "Parallel samplesort with master-worker
           Print "Initializing data..."

           // Generate input data of specified data_type
           Generate size amount of input_data of type da

           // Draw a sample of size s
           // Choose s based on some multiple of m (s =
           s = m * oversampling_factor
           Sample s elements from input_data  // Randoml

           // Sort the sampled elements
           Sort the sampled elements using quicksort
           QuickSort(sampled_elements, length(sampled_el

           // Select m-1 splitters from the sorted sampl
           Select the s/m, 2*(s/m), ..., (m-1)*(s/m) ele
           // These m-1 splitters will be used to partit

       // Broadcast the splitters to all processes
       MPI_Bcast(splitters, m-1, data_type, root=MASTER,

    7. // All Processes (Master and Workers)

       // Scatter the input data to all processes
       local_size = size / num_tasks  // Calculate the s
       local_data = Array[local_size]
```

```
MPI_Scatter(input_data, local_size, data_type, lo
// Each process now has its portion of the data t

// Each process partitions its data into m bucket
Local_buckets = Create m empty buckets for partit

// Assign data to buckets based on splitters
For each element in local_data:
    Determine the correct bucket for the element
    Append element to the corresponding bucket ir

// Prepare data for sending to other processes
// Convert Local_buckets to an appropriate struct
send_counts = [Number of elements in each bucket
send_displacements = [Offsets for each bucket to

// Use MPI_Alltoallv to exchange bucket data amor
recv_counts = [Number of elements to receive from
recv_displacements = [Offsets for each received b

total_recv_size = sum(recv_counts)  // Total size
recv_data = Allocate array of size total_recv_siz

MPI_Alltoallv(Local_buckets, send_counts, send_di
// Each process now has all the elements that bel
local_bucket = recv_data  // This is the bucket e

// Sort the received bucket using iterative quick
Print "Task", task_id, "sorting its bucket..."
QuickSort(local_bucket, length(local_bucket))

8. // Gather sorted buckets back to the master

   // Use MPI_Gather to gather all sorted buckets
   sorted_bucket_size = length(local_bucket)
   sorted_buckets = None
   If task_id == MASTER:
       sorted_buckets = Array[size]  // Master will

   MPI_Gather(local_bucket, sorted_bucket_size, datc

9. // Master Process
   If task_id == MASTER:
       // Concatenate all sorted buckets to get the
       Final_sorted_data = concatenate(sorted_bucket
       Print "Parallel samplesort completed."

10. // Finalize MPI environment
```

```
        MPI_Finalize()

End Main Function


// Iterative Quicksort Algorithm Function
QuickSort Function (arr, n):
    // arr: array to be sorted
    // n: size of the array

    Create an empty stack stack
    Push (0, n - 1) onto stack  // Push initial subarray

    While stack is not empty:
        // Pop high and low indices from stack
        (low, high) = Pop(stack)

        // Partition the array
        If low < high:
            // Choose the pivot element as the last elem
            pivot = arr[high]
            i = low - 1

            // Rearrange elements based on pivot
            For j from low to high - 1:
                If arr[j] <= pivot:
                    i = i + 1
                    Swap arr[i] and arr[j]

            // Put the pivot in its correct position
            Swap arr[i + 1] and arr[high]
            pivot_index = i + 1

            // Push the left and right subarrays onto th
            // Left subarray: elements less than the piv
            If (pivot_index - 1) > low:
                Push (low, pivot_index - 1) onto stack

            // Right subarray: elements greater than the
            If (pivot_index + 1) < high:
                Push (pivot_index + 1, high) onto stack

End Function
```

Bitonic Sort

```
MAIN BEGIN
    // Initialize MPI
    MPI_Init()

    // Get the number of processes
    MPI_Comm_size()

    // Get the rank/ID of the process
    MPI_Comm_rank()

    // Scatter data among processes
    MPI_Scatter()

    // Bitonic Sort process
    for (i = 0; i < log(number processes); i++) {  // lo
        for (j = i; j >= 0; j--) {      // Comparing pai
            if ((process_rank >> (i + 1)) % 2 == 0 && (p
        // Use rightshift operator to determine which pr
                // Both ranks are even or both are odd
                Swap elements in ascending order (smalle
                (send maximum value to partner
                receive minimum value from partner)
            }
            else if ((process_rank >> (i + 1)) % 2 != 0
                // One rank is even and one is odd
                Swap elements in descending order (vice
                (send minimum value to partner
                receive maximum value from partner)
            }
        }
    }

    // Gather results back to the root process
    MPI_Gather()

    // Finalize MPI
    MPI_Finalize()
MAIN END
```

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

- Input sizes: 2^16, 2^18, 2^20, 2^22, 2^24, 2^26, 2^28
- Input types: Sorted, Random, Reverse sorted, 1% perturbed
- Strong scaling (same problem size, increase number of processors/nodes)

- Weak scaling (increase problem size, increase number of processors)
- For number of processors: 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024

For each algorithm (Bitonic Sort, Sample Sort, Merge Sort, Radix Sort), the following metrics will be measured:

- **Execution Time**: Track the time required to complete the sort for various input sizes, input types, and processor counts.
- **Speedup**: Compare the parallel execution time with the sequential baseline to assess the improvement for each algorithm.
- **Efficiency**: Determine how effectively each algorithm uses additional processors by calculating speedup divided by the number of processors.

Strong Scaling:

- **Objective**: Analyze how each algorithm performs as the number of processors increases while keeping the input size constant.
- **Method**: For a fixed input size (e.g., 2^24 elements), run the algorithm with 2, 4, 8, 16, 32, and more processors. We will measure execution time and calculate speedup and efficiency for each case.
- **Comparison**: Evaluate how each algorithm's execution time changes with different processor counts.

Weak Scaling:

- **Objective**: Evaluate how well each algorithm handles increasing input sizes as the number of processors grows proportionally, while ensuring each processor has sufficient work to do.
- **Method**: For each input type (Sorted, Random, Reverse Sorted, 1% Perturbed), we will start with a small input size and a reasonable number of processors. As we increase the input size, we plan to proportionally increase the number of processors:
    - 2^16 elements for 2 processors
    - 2^18 elements for 4 processors
    - 2^20 elements for 8 processors
    - 2^22 elements for 16 processors
    - 2^24 elements for 32 processors
    - 2^26 elements for 64 processors
    - 2^28 elements for 128 processors
    - 2^28 elements for 256 processors
    - 2^28 elements for 512 processors

    - 2^28 elements for 1024 processors
- **Comparison**: Measure the execution time for each combination of input size and processor count. The goal is to maintain a stable execution time as both the input size and the number of processors increase, showing strong weak scaling properties.

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
```

Implementations

Bitonic Calltree:

```
1.720 main
├─ 0.000 MPI_Init
├─ 0.000 data_init_runtime
├─ 0.026 comp
|  ├─ 0.006 comp_small
|  └─ 0.079 comp_large
├─ 0.006 comm
|  └─ 0.006 MPI_Gather
├─ 0.000 correctness_check
├─ 0.000 MPI_Finalize
├─ 0.000 MPI_Initialized
├─ 0.000 MPI_Finalized
└─ 0.080 MPI_Comm_dup
```

Sample Sort Calltree:

```
30.491 main
├─ 0.000 MPI_Init
├─ 0.228 data_init_runtime
├─ 7.454 comp
|  ├─ 3.980 comp_small
|  └─ 3.474 comp_large
├─ 1.377 comm
|  ├─ 0.021 comm_small
|  |  ├─ 0.004 MPI_Gather
|  |  └─ 0.016 MPI_Bcast
|  └─ 1.356 comm_large
|     ├─ 0.015 MPI_Alltoall
|     └─ 0.222 MPI_Alltoallv
├─ 0.115 correctness_check
├─ 0.000 MPI_Finalize
├─ 0.000 MPI_Initialized
├─ 0.000 MPI_Finalized
└─ 19.820 MPI_Comm_dup
```

Merge Sort Calltree:

```
2.422 main
├─ 0.000 MPI_Init
├─ 0.091 data_init_runtime
├─ 0.017 comm
|  ├─ 0.012 MPI_Barrier
|  └─ 0.005 comm_large
|     ├─ 0.004 MPI_Scatter
|     └─ 0.001 MPI_Gather
├─ 0.048 comp
|  └─ 0.048 comp_large
├─ 0.016 correctness_check
├─ 0.000 MPI_Finalize
├─ 0.000 MPI_Initialized
├─ 0.000 MPI_Finalized
└─ 0.674 MPI_Comm_dup
```

Radix Sort Calitree

```
1.664 main
├─ 0.000 MPI_Init
├─ 0.000 data_init_runtime
├─ 0.060 comp
|  ├─ 0.000 comp_small
|  └─ 0.060 comm
|     ├─ 0.054 comm_small
|     |  └─ 0.054 MPI_Alltoall
|     └─ 0.006 comm_large
|        └─ 0.006 MPI_Alltoallv
├─ 0.000 correctness_check
├─ 0.000 MPI_Finalize
├─ 0.000 MPI_Initialized
├─ 0.000 MPI_Finalized
└─ 0.000 MPI_Comm_dup
```

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 t
adiak::clustername();   // Name of the cluster
adiak::value("algorithm", algorithm); // The name of the
adiak::value("programming_model", programming_model); //
adiak::value("data_type", data_type); // The datatype of
adiak::value("size_of_data_type", size_of_data_type); //
adiak::value("input_size", input_size); // The number of
adiak::value("input_type", input_type); // For sorting,
adiak::value("num_procs", num_procs); // The number of p
adiak::value("scalability", scalability); // The scalabi
adiak::value("group_num", group_number); // The number o
adiak::value("implementation_source", implementation_sou
```

Bitonic Metadata: The metadata collected for bitonic sort includes the following information:

```
launchdate: 1729117054,
libraries: [/scratch/group/csce435-f24/Caliper/caliper/l
cmdline: [./mergesort, 65536, 4, Random],
algorithm: bitonic_sort,
programming_model: mpi,
data_type: int,
size_of_data_type: 4,
input_size: 65536,
input_type: Random,
num_procs: 4,
scalability: strong,
group_num: 20,
implementation_source: handwritten
```

Sample Sort Metadata:

```
cali.caliper.version: 2.11.0
mpi.world.size: 32
spot.metrics: min#inclusive#sum#time.duration, max#inclu
spot.timeseries.metrics: (not provided)
spot.format.version: 2
spot.options: time.variance, profile.mpi, node.order, re
spot.channels: regionprofile
cali.channel: spot
spot:node.order: true
spot:output: p32-a268435456.cali
spot:profile.mpi: true
spot:region.count: true
spot:time.exclusive: true
spot:time.variance: true
launchdate: 1729094595
libraries: [/scratch/group/csce435-f24/Caliper/caliper/l
cmdline: [./samplesort, 268435456, 32, Random]
cluster: c
algorithm: sample_sort
programming_model: mpi
data_type: int
size_of_data_type: 4
input_size: 268435456
input_type: Random
num_tasks: 32
scalability: strong
group_num: 20
implementation_source: handwritten
```

Merge Sort Metadata:

```
cali.caliper.version: 2.11.0
mpi.world.size: 32
spot.metrics: min#inclusive#sum#time.duration, max#inclu
spot.timeseries.metrics: (not provided)
spot.format.version: 2
spot.options: time.variance, profile.mpi, node.order, re
spot.channels: regionprofile
cali.channel: spot
spot:node.order: true
spot:output: p32-a4194304.cali
spot:profile.mpi: true
spot:region.count: true
spot:time.exclusive: true
spot:time.variance: true
launchdate: 1729123305
libraries: [/scratch/group/csce435-f24/Caliper/caliper/l
cmdline: [./mergesort, 4194304, 32, random]
cluster: c
algorithm: merge
programming_model: mpi
data_type: int
size_of_data_type: 4
input_size: 4194304
input_type: random
num_procs: 32
scalability: strong
group_num: 20
implementation_source: handwritten
```

Radix Sort

```
cali.caliper.version: 2.11.0
mpi.world.size: 32
spot.metrics: min#inclusive#sum#time.duration, max#inclu
spot.timeseries.metrics: true
spot.format.version: 2
spot.options: time.variance, profile.mpi, node.order, re
spot.channels: regionprofile
cali.channel: spot
spot:node.order: true
spot:output: p32-a1024.cali
spot:profile.mpi: true
spot:region.count: true
spot:time.exclusive: true
spot:time.variance: true
launchdate: 1728970597
libraries: [/scratch/group/csce435-f24/Caliper/caliper/l
cmdline: [./radixsort, 1024, Random]
cluster: c
algorithm: radix_sort
programming_model: mpi
data_type: int
size_of_data_type: 4
input_size: 1024
input_type: Random
num_tasks: 32
scalability: strong
group_num: 20
implementation_source: 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.

Algorithm Descriptions:

- Sample Sort Algorithm:

Helper Functions:

1. void quicksort(std::vector<int>& arr);
    - Sorts the array given as an argument using stc

2. std::vector<int> generateSortedData(int size);
    - Creates a vector called data.
    - Through a for loop adds values from 0 to input
    to create a sorted array and returns it.

3. std::vector<int> generatePerturbedData(int size);
    - Creates a vector called data using the generat
    - Calculates 1% of the size given.
    - For 1% of the total size of values swap random
    array to generate perturbed data.

4. std::vector<int> generateRandomData(int size);
    - Creates a vector called data
    - Through a for loop from index 0 to inputted_si
    generates random data and adds it to vector.

5. std::vector<int> generateReverseSortedData(int si
    - Creates a vector called data.
    - Through a for loop adds values from inputted s
    to create a reversed sorted array and returns it

6. std::vector<int> generateRandomInput(int size, cc
    - Uses the given input_type to decide which gene
    - Calls data generation function based on input_
      If the input_type = "Random", it will call ger

Main Function:

1. Initalize MPI
    The algorithm starts by initializing the MPI env
    using MPI_Init and setting up task ids for each
    and the total number of processors (num_tasks).
    Adiak is then initialized to collect metadata ab

2. Set variables from job script arguments
    The input parameters passed to the program are e
        input_size: Total number of elements in the
        num_tasks: Number of processors.
        input_type: The type of data input (sorted,

3. Ensure there are at least two processors
    If statement to make sure num_tasks is at least
    then the program will abort.

4. Adiak collects various data about the program
   This includes algorithm, input_type, etc.

5. Generate data for sorting
   We start off by marking the region of init with
   A vector for local_data is created to generate d
   off of the input type. Each processor will recei
   size of data to generate. This will be evenly di
   For example, if there are 8 processors, each prc
   generates input_size / 8 elements based on the s

6. Sort local data
   Each processor will sort the data that it create
   This will once again be timed by caliper but
   it will be noted as a small computation since we
   are only sorting local data and not the whole ar

7. Drawing sample size
   Each processor will draw a sample size to determ
   The size s is computed as the logarithm of the r
   elements (log2(local_data.size())). The purpose
   communication overhead by ensuring that the samp
   to the size of the data. Based on this sample si
   randomly pick s samples from its data.

8. Gathering samples
   All processors send their local samples to a des
   using MPI_Gather. MPI_Gather will collect the sc
   processors and send them to a designated root pr
   The root processor receives all the samples and
   them into a single array. This process will be m
   as a small communication by caliper.

9. Sample Sorting and Splitting
   The root process now sorts the samples. It then
   from the sorted samples, which will be used to p
   into buckets across all processors. Therefore th
   the number of processors should be equivalent. T
   to secure indices ensuring a good distance that
   The splitters are then broadcast to all processc
   This ensures that every processor knows the rang
   the samples is marked and timed as a small compu
   Bcast function will be marked and timed as a smc

10. Putting data into buckets
    We create a 2D vector, buckets, where each inne
    The number of buckets is equal to the number of
    Each processor has its own bucket where data wi
    Each inner vector (buckets[i]) holds the values

be sent to processor i during the data exchange
over each value in the local data. Each value w
based on the splitters. This loop goes through
to determine where the current value should be
is less than or equal to the current splitter.
is set to i, meaning the value belongs in the b
The loop then breaks as the correct bucket is f
greater than the current splitter, the code mov
check the next splitter and increments.

11. Assigning buckets to processors
    We then initialize our counts. We will use thes
    to assign sizes of each buckets to sendcounts t
    how much each processor will be sending to othe
    and receiving from other processors. Using all
    send these values to all of the processors. Nex
    calculate the total size of data the current pr
    by summing up all elements in recvcounts and ir
    recvbuf to hold the data. The send_index and
    receive_index arrays are then set up to indicat
    for sending and receiving data in the buffers f
    We then convert the 2D buckets vector into a si
    vector so that it can be processed by MPI, enab
    to send its data to the appropriate destinatior

12. Sending data to buckets and sorting data
    Using MPI_Alltoallv, each processor sends its d
    to the appropriate processors. This and steps 1
    large communication and timed by caliper. We th
    to time the sorting of all arrays by the proces
    partitioned data, each processor sorts the rece

13. Correctness Check
    After sorting is completed, the alorithm checks
    sorted data is in order using std::is_sorted. I

14. Finalizing MPI
    The MPI environment is finalized, marking the er

- Merge Sort Algorithm:

Sorting Functions:
1. void merge(int *arr1, int *arr2, int LI, int mid,
    - This function allows for combination of two sc
    - It takes the left index, midpoint, right index
    - As we go through each of the index for the lef
    - If we have any leftover elements after the inc
    - Following the subarray sorting, the temporary
2. void mergeSort(int *arr1, int *arr2, int LI, int
    - This function will recursively divide the arrc
    - As long as the left index is less than the rig
    - This leads to the recursive call of mergeSort
    - After the dividing is complete, the merging wi

Main Function:
1. Initialize MPI/Adiak
    - Begin by initializing the MPI environment and
    - Initialize Adiak as well for performance monit
2. Input Validation
    - Ensure that there is three command-line argume
3. Declare and Initialize Variables:
    - Read and convert the command-line arguments ir
    - These arguments are the array size, number of
    - Make sure there are at least 2 MPI processes r
4. Collect Data with Adiak:
    - Collect the needed information with Adiak: alg
5. Array Allocation:
    - Allocate memory for the original array based c
6. Array Population with Input Type:
    - Populate the original array with the input typ
    - Sorted: provide a basic sorted array based on
    - Random: randomly generate numbers for the arrc
    - Reverse: provide a basic reversed array based
    - 1%perturbed: randomly select a few values and
    * this also marks the end of data initializatior
7. Subarray Size and Memory:
    - Determine each subarray size for process distr
    - Allocate memory for the subarray to be able tc
8. Scatter
    - Start tracking comm and comm_large performance
    - Use a barrier to synchronize all the processes
    - Distribute subarrays from the original array t
9. Merge Sort
    - Start tracking comp and comp_large performance
    - Allocate a temporary memory for the tempArray
    - Call the mergeSort function to sort that subAr
10. Allocate memory for final merge:
    - Create a pointer to be used for the final sort
    - In the master process, we want to allocate mem
11. Gather:

```
        - Start tracking comm and comm_large performance
        - Use MPI_Gather to collect all the sorted subar
    12. Final Merge:
        - Make sure its all the master process before al
        - Start tracking comp and comp_large performance
        - Call the mergeSort function for the sorting of
    13. Correctness Check:
        - Start correctness_check to measure performance
        - Use a boolean to check if its sorted with the
        - If the boolean is true then it has been sortec
    14. Finalize and Cleanup
        - Free all memory allocated areas
        - Call MPI_Finalize() to finalize the MPI envirc
```

Bitonic Sort Algorithm:

```
Helper Functions:
1. quicksort(std::vector<int>& arr): Uses standard libra
2. generateSortedData(int size): Generates a sorted vect
3. generatePerturbedData(int size): Starts by creating a
4. generateRandomData(int size): Produces a vector of ra
5. generateReverseSortedData(int size): Generates a reve
6. generateRandomInput(int size, const std::string& inpu

Sorting Functions:
1. compare_and_swap(std::vector<int>& data, int i, int j
2. bitonic_merge(std::vector<int>& data, int low, int cr
3. bitonic_sort(std::vector<int>& data, int low, int cnt

Main Function:
1. MPI Initialization: We use MPI_Init to start MPI, MPI
2. Metadata Collection: adiak statements - init, launcha
3. Input Handling: Handles command line arguments passed
4. Metadata Collection: adiak statements - algorithm . .
5. Local Data Distribution: Using the generateRandomInpu
6. Local On Each Process: Sort each array with quicksort
7. MPI Gather: Part of comm, it collects the locally sor
8. Global Sorting: The root process performs the final b
9. Correctness Check: Verifies that the sorted data is a
10. MPI Finalize: Cleans up and shuts down the MPI envir
```

Radix Sort

```
Initialization:
MPI is initialized, and process ranks are retrieved.
Metadata about the run is recorded using Adiak.

Input Generation:
Each process generates a portion of the input data basec

generateSortedData(size): Generates an array of integers
generatePerturbedData(size): Perturbs the sorted array b
generateRandomData(size): Generates an array of random i
generateReverseSortedData(size): Generates an array of i

Radix Sort Setup
Bits per Pass: The program calculates how many bits are
Radix: The number of "bins" or buckets is determined by
Total Passes: The number of passes required for sorting


Parallel Radix Sort:
Radix Sort is performed in parallel. Each process calcul

The program runs Radix Sort in parallel across multiple
Loop Through Radix Passes: For each pass:
Computation Phase (CALI_MARK_BEGIN("comp_small")):
Each process determines which "bucket" or range of value
Data is organized into local buckets, with each bucket t
Communication Phase:
Processes exchange data with one another to ensure that
MPI_Alltoall() is used to exchange the number of element
MPI_Alltoallv() is used to exchange the actual data betw
Local Data Update: After receiving data from other proce

Correctness Verification:
After sorting, each process verifies that its data is sc

Profiling and Metadata:
The algorithm is profiled using Caliper to measure compu

Finalization:
MPI is finalized, and the program terminates.
```

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).

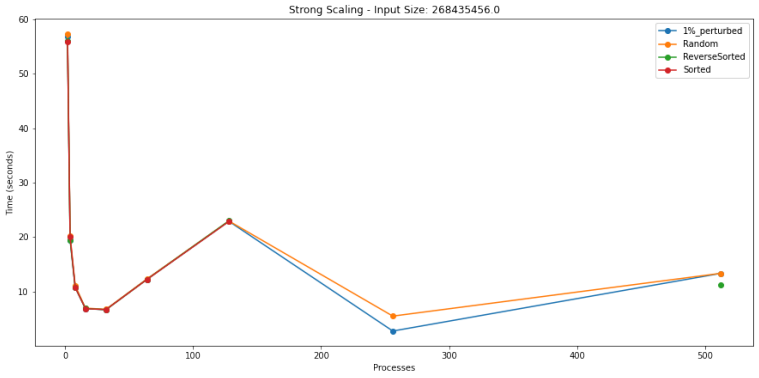
4c. You should measure the following performance metrics

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

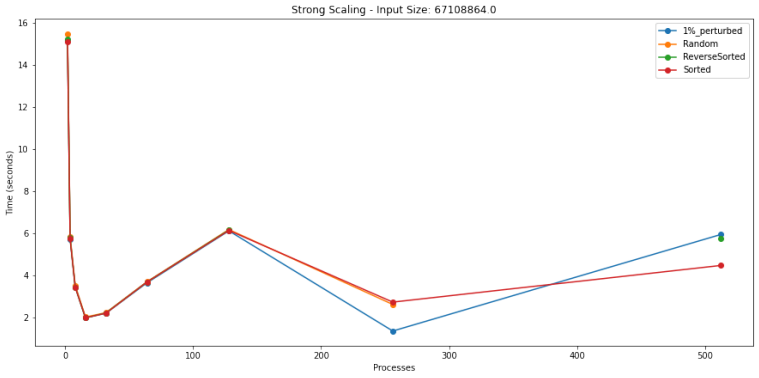
Plots and Analysis:

- Radix Sort:

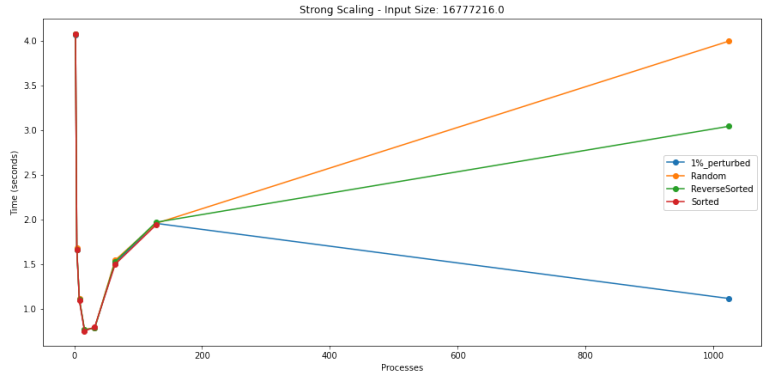
Strong Scaling - Input Sizes



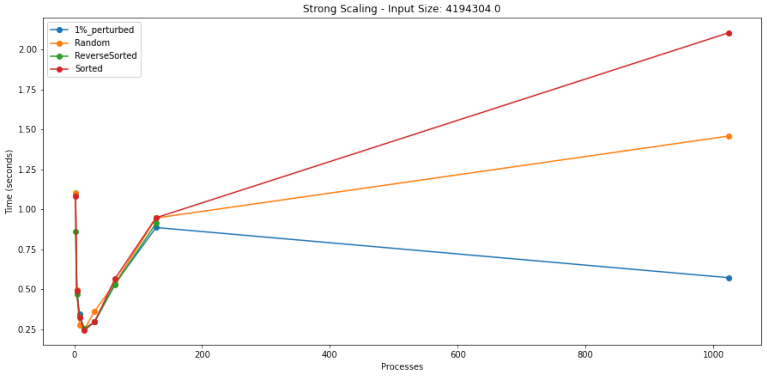
The plot shows that for a very large input size, increasing the number of processes initially reduces computation time significantly, but beyond 100-200 processes, the benefits taper off. This is likely due to communication and synchronization overheads, which will start to outweigh the advantages of adding more processors. Past this point, performance stabilizes or worsens (but very slightly), indicating that further increases in the number of processors offer diminishing returns. The different input types show similar performance patterns, suggesting that for large inputs, the sorting state of the data doesn't really impact scalability.



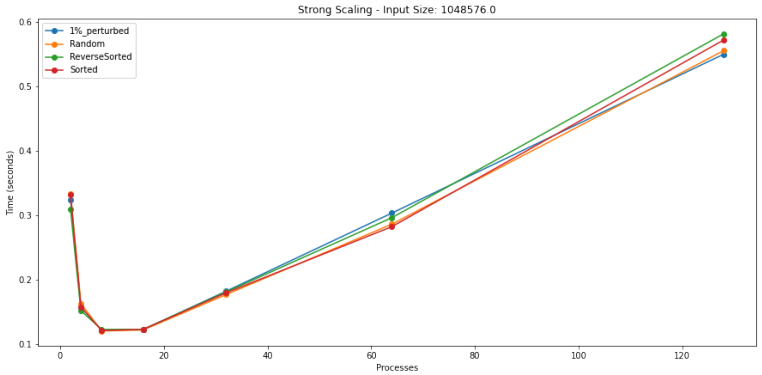
This graph exhibits a similar pattern to the above graph. Increasing the number of processes initially reduces computation time significantly, but beyond 100-200 processes, the benefits taper off. This is likely due to communication and synchronization overheads, which will start to outweigh the advantages of adding more processors. However, 1% perturbed does have a slighter higher execution time at the highest processor count. This could be due to increased irregularity in the distribution of the data.



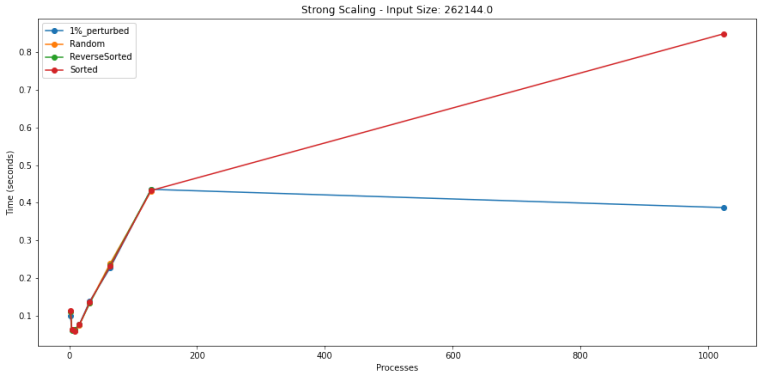
This graph shows a significant decrease in overall computational time beyond processors 2-16. However, with more processors beyond that count, the benefits increasingly taper off. This is likely due to overwhelming communication and synchronization overheads. These factors can cause certain input types to become less efficient as more processors are used. Interestingly enough, 1% perturbed comes out as the winner with the highest number of processors with the lowest execution count, while random has the largest computational time.



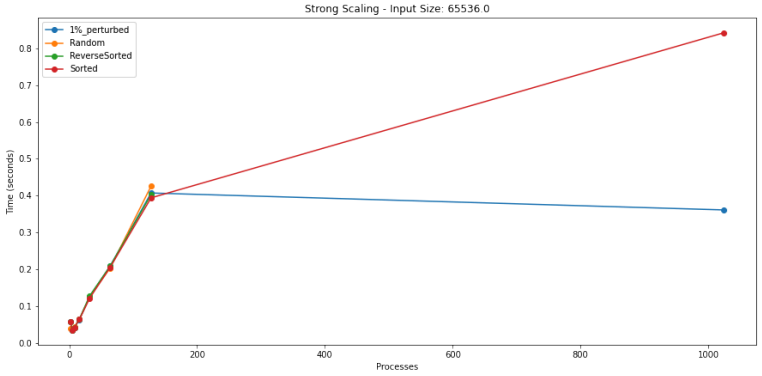
Similar to the previous graph, this graph shows a significant decrease in overall computational time beyond processors 2-16. However, with more processors beyond that count, the benefits increasingly taper off. The points at which the different algorithms start to see diminishing returns differ very slightly, compared to the graph above. This is likely due to overwhelming communication and synchronization overheads. This is more clearly shown that these factors can cause certain input types to become less efficient as more processors are used.



Similar to the previous graph, this graph shows a significant decrease in overall computational time beyond processors 2-8.



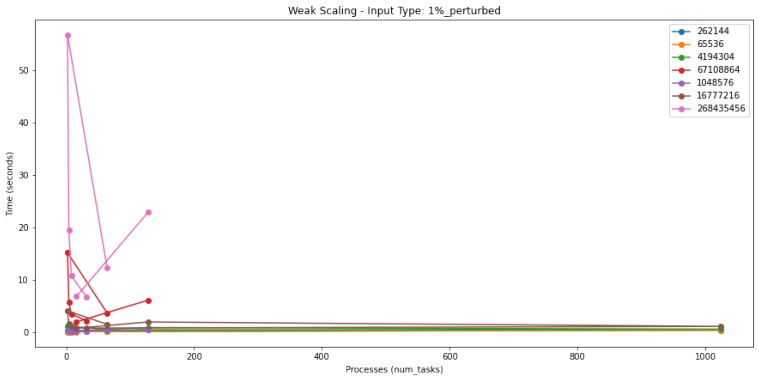
Similar to the previous graph, this graph shows a significant decrease in overall computational time beyond processors 2-4.

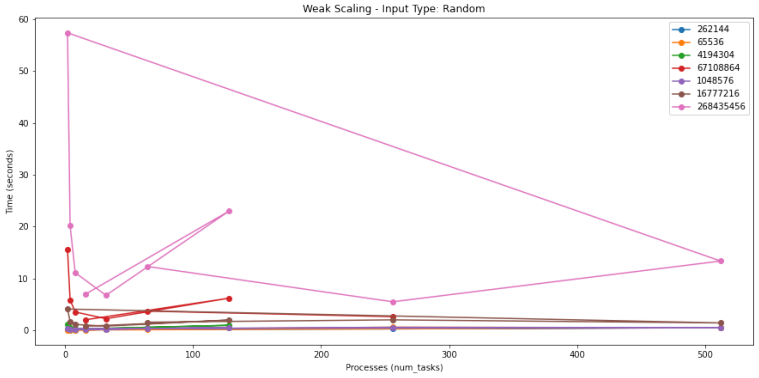


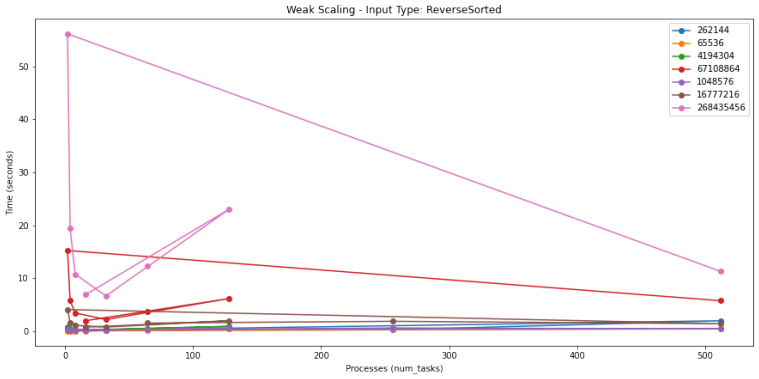
Similar to the previous graph, this graph shows a significant decrease in overall computational time beyond processors 2.

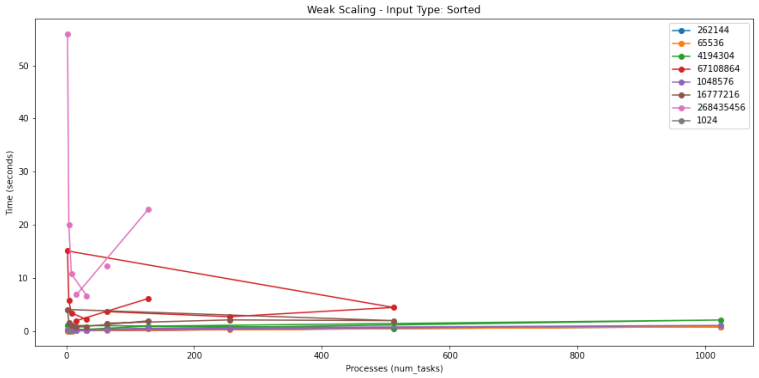
Overall, the analysis of the strong scaling in regards to input size shows how, with smaller input sizes, there are diminishing returns (and even detrimental effects due to the increased overhead of processor communication and synchronization overheads) as processors are scaled up.

Weak Scaling





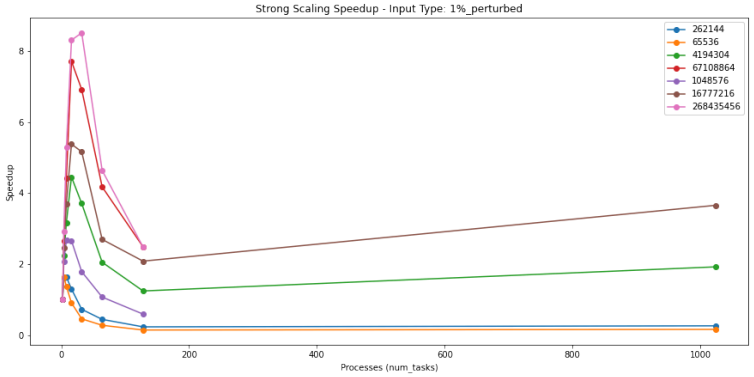




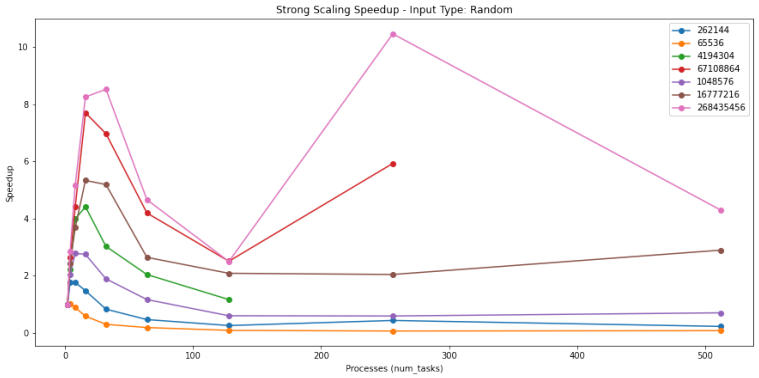
Overall: For smaller input sizes, the time remains relatively stable as the number of processes increases, indicating good scalability. However, for the larger input sizes (like 67108864 and 268435456), there is an initial spike in execution time, which then declines as the number of processes increases. This suggests while smaller input sizes scale efficiently, larger input sizes face more significant overhead at least initially, likely due to communication and synchronization costs. As more processes are added, this overhead reduces, though the larger inputs still take longer to process compared to smaller ones.

The trend persists across the different input types.

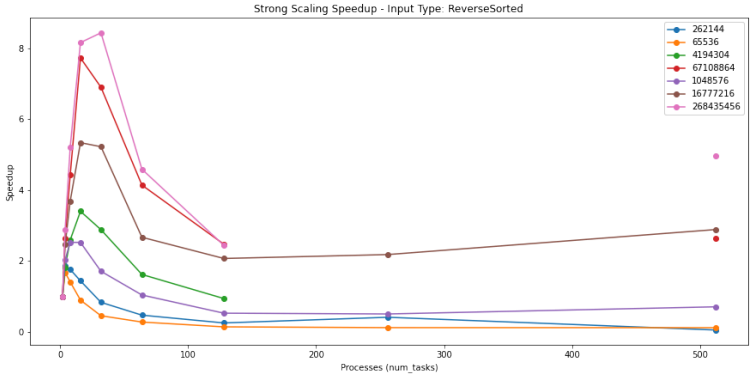
Strong Scaling - Speedup



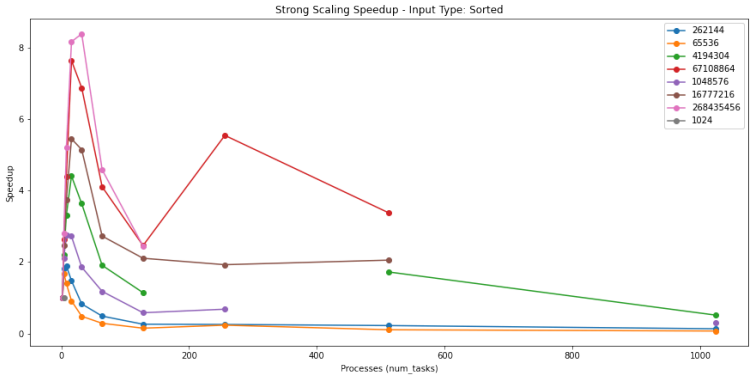
In this plot, strong scaling seems effective up to a certain number of processes, after which the overhead of coordination reduces the speedup, particularly for smaller input sizes. Larger input sizes maintain better scalability, but still diminished speedup as they continue to benefit from additional processors.



Similar to the above, in this plot, strong scaling seems effective up to a certain number of processes, after which the overhead of coordination reduces the speedup, particularly for smaller input sizes. Larger input sizes maintain better scalability, but still diminished speedup as they continue to benefit from additional processors.



Similar to the above, in this plot, strong scaling seems effective up to a certain number of processes, after which the overhead of coordination reduces the speedup, particularly for smaller input sizes. Larger input sizes maintain better scalability, but still diminished speedup as they continue to benefit from additional processors.

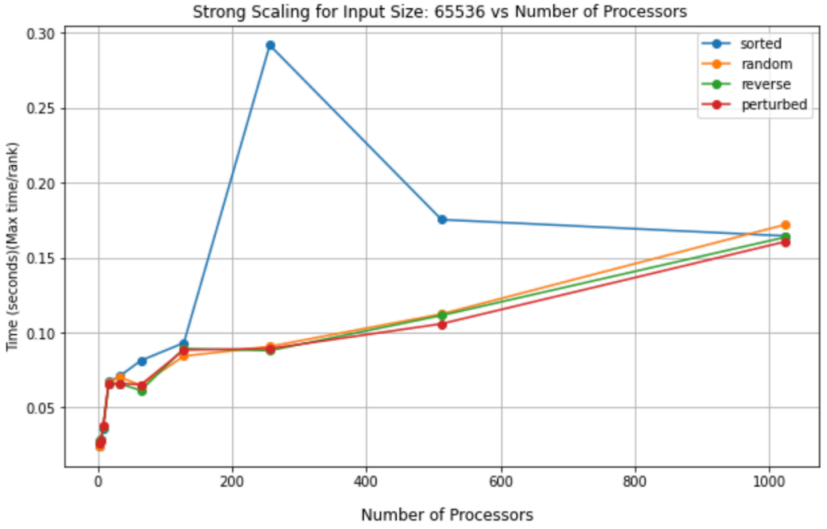


Similar to the above, in this plot, strong scaling seems effective up to a certain number of processes, after which the overhead of coordination reduces the speedup, particularly for smaller input sizes. Larger input sizes maintain better scalability, but still diminished speedup as they continue to benefit from additional processors.

Overall, random seems to benefit the most when increasing processors up to a certain amount, and has a higher processor count allowed before the speedup starts to weaken and decrease. Sorted has a steep dropoff in speedup very early on in processor count.

- Merge Sort:

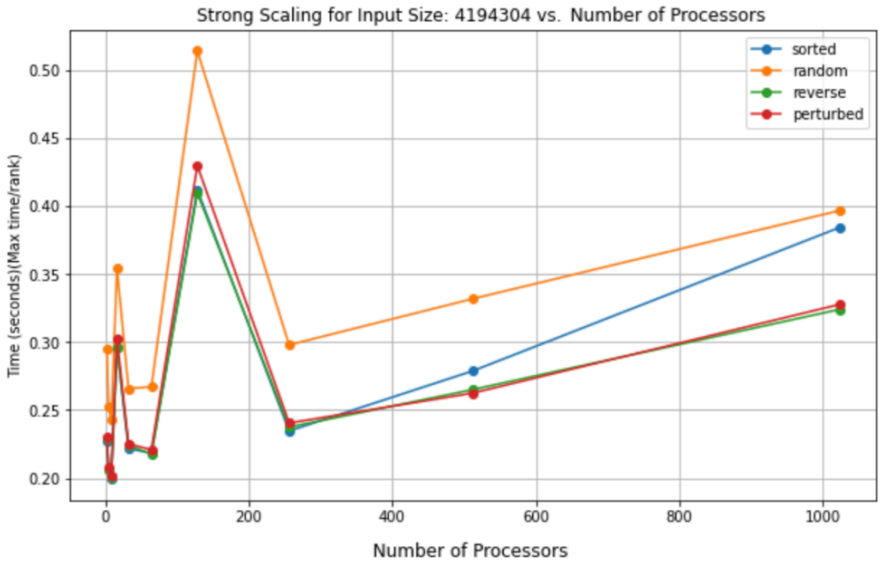
Strong Scaling Plot for 2^16:



Generally, looking at the smallest input size of 2^16 for strong scaling, the performance seems to decrease in strong scaling efficiency as the number of processors increase. The performance seems to spike at around 400

processors specifically for sorted input which could be due to some inefficiency or anomalies in the communication or workload distribution. After 400 processors, the performance does improve with a downward trend before leveling out. As the problem size is small, dividing it across a large number of processors (after 400) could mean that each process is handling a small amount of work. Thus, creating a situation where communciation and synchronization overhead will dominate. For the other input types, the random, reverse, and perturbed seem to have a more gradual scaling but also starts to degrade as the number of processors increase, specifically after around 200 processors as the time starts to rise quite steadily. Especially for small input sizes, the amount of work per process is relatively limited which dominates the synchronization costs. There could be a high synchronization overhead due to many processors working on the small problem. Also in the merging phase, the processors could need to wait for one another which could create larger times due to the smaller input size.

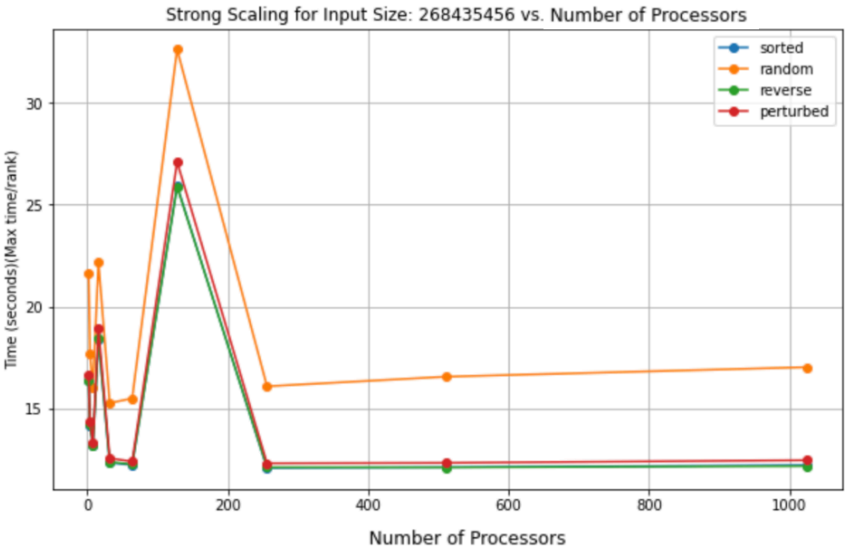
Strong Scaling Plot for 2^22:



This graph contains the strong scaling for the input size of 2^22. Here, the degradation is more subtle than the 2^16 graph with the performance being stabilized after 200 processors. There are initial spikes before 200 processors. However, even after the spikes, there is still a general increase as more processors are added which suggests that there could be more communication overhead. Synchronization could also be a issue when

introducing more processors because the synchronization between the processors will add an layer of overhead. With more processors, the merging phase requires the processors to exchange their results and ensures that the merges are done in a specific order which will increase communication and synchronization costs. Specifically for random input in processors before 200, there is instability being shown. The spikes should suggest that communication or data partitioning was inefficient and even after flattening out, the system does not gain much more speedup as the increasing amount of processors may add more overhead than what is saved in computation time. Moreover, the random input type data could be higher due to the load imbalances that are caused by the unpredictable nature of random data which could also make it harder to distribute the workload more evenly.

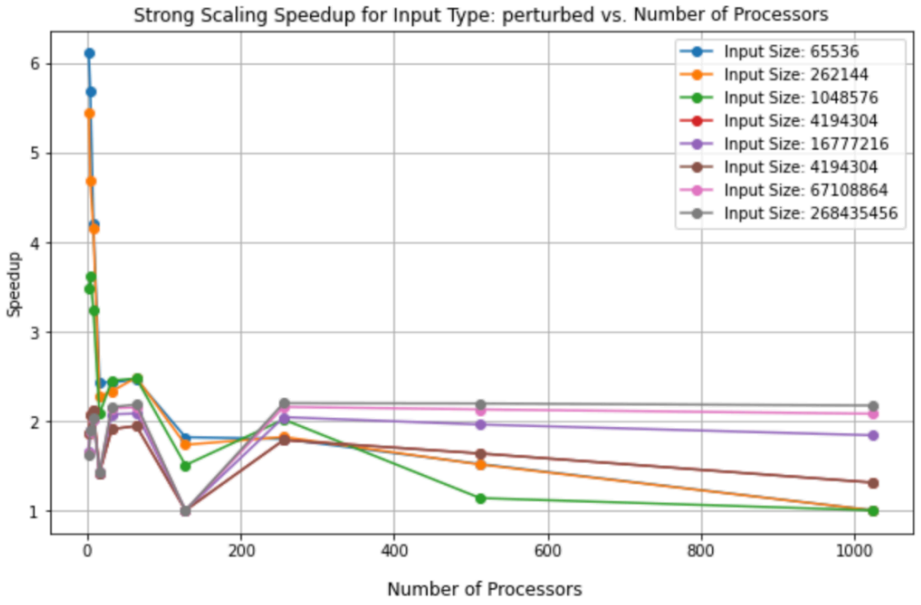
Strong Scaling Plot for 2^28:



This graph contains the strong scaling for the input size of 2^28. Here, there is less degradation as the larger input size allows for each process to handle more work so communication and computation ratio should be good. However, there seems to still be instability specifically at the spike before 200 processors especially for random input data which could be due to the unpredictable nature of random data. Even with improvement, after 200 processors the trend starts to become constant which could mean that the synchronization costs start to outweigh the computational benefits of adding more processors. Therefore, additional processors do

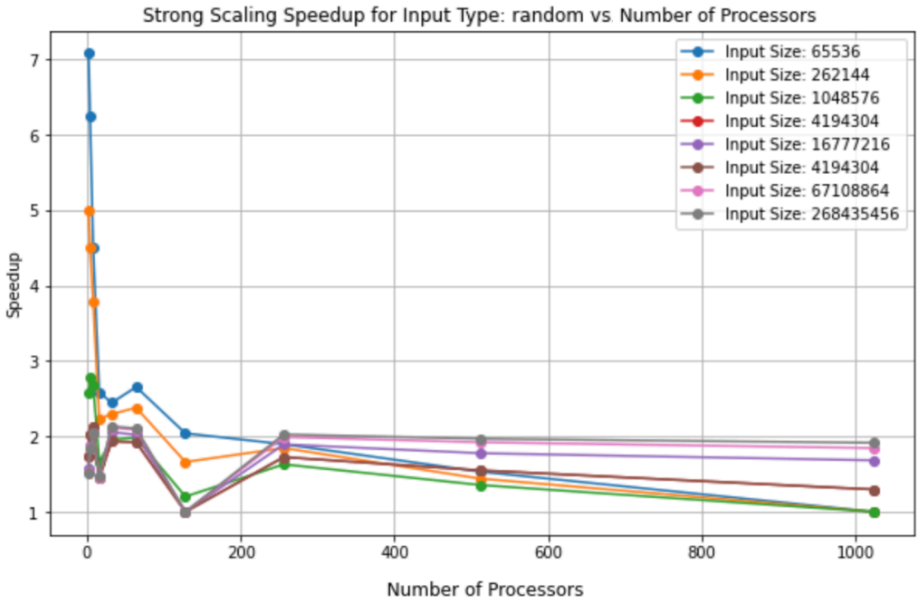
not significantly reduce overall time as they could be waiting on one another specifically in the merge phase.

Strong Scaling Speedup for 1%Perturbed:



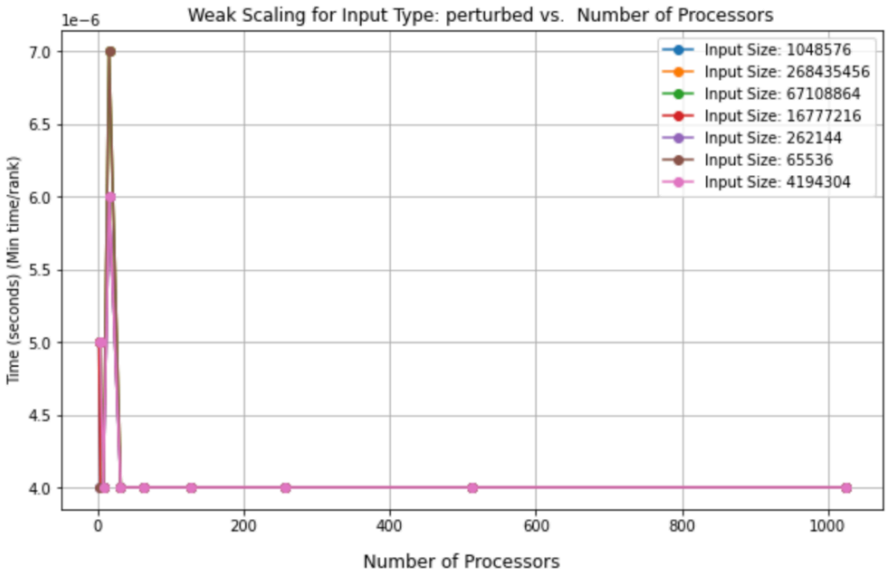
This graph is describing the strong scaling speedup for the 1% perturbed imput type. For the smaller processors, it seems that the speedup is a lot higher. The speedup seems to start off at 6 for the smallest input size which suggests that the parallelization should be working fine initially. Merge sort should be able to provide a substantial speedup early on when the input size is small and division of work is efficient. However after around 100 processors, the speedup starts to flatten out or degrade for all the input sizes, hovering at around 1.5 and 2 speedup for the larger input sizes. For the smaller input sizes, the speedup does drop quite sharply because the work per process becomes very small so synchronization/communication overhead can begin to dominate the runtime. In the larger input sizes, the speedup does remain quite constant at around 2 as the number of processors increase. This should suggest that there is pretty good speedup at the larger inputs. The flattening out of speedup could be because of the synchronization costs during the merging for merge sort. With the increase in input size, each merge should require more and more synchronization between processors, which could limit further speedup.

Strong Scaling Speedup for Random:



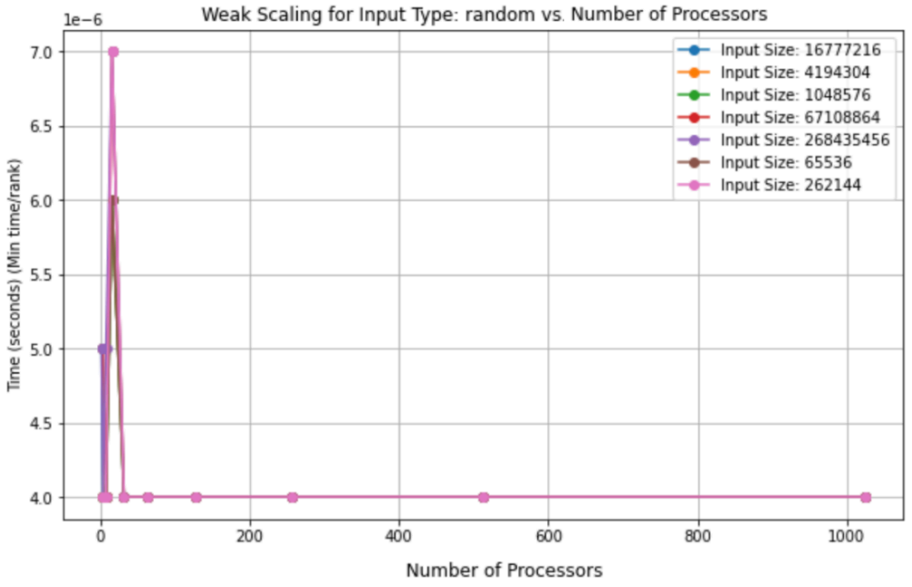
For this strong scaling speedup graph, the input type used is random and we see that it behaves quite similarly to the perturbed input graph, where the smaller input sizes are also able to show high speedup at around 7 with a small number of processors. Once again, the high initial speedup should be expected as merge sort should be able to easily parallelize the sorting phase when inputs are randomly distributed. However, when the number of processors start to grow, synchronization and communication costs increase which are shown in the trend in the graph. After around 100 processors, the speedup drops across all input sizes, with a particularly noticeable decrease for smaller inputs, where the speedup falls to around 1.5 or lower as the number of processors continues to increase. Quite similar to the perturbed input, the random input data also experiences significant load imbalance during the merging phase where some of the processors end up sitting idle, waiting for others to complete their merges. This could reduce the efficiency thus dropping the speedup. The larger input sizes do maintain better speedup than the smaller ones but still does not reflect the ideal condition of being linearly scaling with the increase in processors. This should show the communication bottlenecks which could be present in the merging phase which becomes more prominent as more processors are involved in merging random data.

Weak Scaling for 1%Perturbed:



This graph is looking at the weak scaling for the input type of 1% Perturbed. We see that there is a spike in time in the smaller number of processors. This could be due to initial overhead when only a few processors are used to sort and merge large chunks of data. This could also indicate that the overhead of both communication and task distribution is relatively significant when the problem size and number of processors are small. After the initial spike, the time per rank stabilitizes for all input sizes as the number of processors increase. This should indicate that once the system has enough processors to distribute the work efficiently, the execution time per process will start to stay constant even as input size and processors increase. This should suggest that there is pretty good weak scaling since the execution time does not really increase as the number of processors increase. This could be as this input type does not have much variability with weak scaling once the initial overhead is done. Both the sorting and the merging phases are distributed quite well across the processors.

Weak Scaling for Random:



This graph is quite similar to the perturbed one, the random input type also has a similar spike when there are few processors being used. This seems to be fairly reasonable as with fewer processors, the communication and synchronization overhead should be higher relative to the computational work which should cause that initial spike. After that spike, as the number of processors increase, it seems to be stabilized regardless of the input size. This should indicate good weak scaling performance, as the algorithm is able to effectively distribute its workload and handle all the communication overhead with the increase in processors. The random input type does not seem to cause an issue for weak scaling, once the number of processors start to increase beyond that initial spike, the time stabilizies which should indicate efficiency.

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