CSCE 435 Group project

0. Group number: 3

1. Group members:

1. Anjali Hole
2. Yahya Syed
3. Kyle Bundick
4. Peter Schlenker
5. Harsh Gangaramani

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

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

- **Bitonic Sort (Peter)**: A divide-and-conquer algorithm implemented using MPI that sorts data into many bitonic sequences (the first half only increasing, the second half only decreasing). It then creates alternating increasing and decreasing sequences out of the bitonic sequences to create half as many bitonic sequences, but twice the size. It keeps repeating this process until there is one large bitonic sequence left, at which point it creates one final increasing sequence. For the parallel version I'm implementing, instead of one value each process will keep a sorted list, and when two processes compare lists the smaller sequence will hold a sorted list where all the elements are smaller than the elements in the bigger sequence.
- **Sample Sort (Kyle)**: A divide-and-conquer algorithm implemented in MPI that splits the data into buckets based on data samples, sorts the buckets, and then recombines the data.
- **Merge Sort (Anjali)**: A parallel divide-and-conquer algorithm implemented using MPI for efficient data distribution and merging where each process independently sorts a portion of the data, and MPI coordinates the merging of subarrays across multiple processors on the Grace cluster.
- **Radix Sort (Yahya)**: A divide-and-conquer algorithm implemented with MPI that sorts an array of integers digit by digit, using counting and prefix sums instead of direct comparisons to determine sorted order. Data distribution is determined by digit values, with each process responsible for certain digits.
- **Column Sort (Harsh)**: A multi-step matrix manipulation algorithm implemented using MPI that sorts a matrix by its columns, redistributes it through a series of transpositions, and applies strategic global row shifts

Team Communication

- Team will communicate via Discord (for conferencing/meeting)
- Team will use the GitHub repo for reports, and Google Drive to share generated graphs/ report details

What versions do you plan to compare:

Communication strategies:

```
a. Point-to-point communication (as shown in the pseudocode)
b. Collective communication (using MPI_Allgather or MPI_Alltoall)
```

Parallelization strategies:

```
a. SPMD (Single Program, Multiple Data) as shown in the pseudocode
b. Master/Worker model
```

2b. Pseudocode for each parallel algorithm

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

Bitonic Sort

```
// Assumes the total list size is a power of 2, and that comm_size is a power or 2
less than or equal to the list size, and that local_data size times comm_size is
the total list size
function bitonic_sort(local_data, comm_size, rank):
    local_data = sequential_sort(local_data)

    for level = 0 to log2(comm_size) - 1:
        is_increasing = !rank.bit(level + 1)

        for current_bit = level to 0:
            other_rank = rank.flip_bit(current_bit)

            // While the data lives on two processes, only one needs to do the
comparison.
            // For now the lower rank process will always do the comparison,
though it might speed up the algorithm if we try to balance who does the
comparison more evenly.
            is_doing_comparison = rank < other_rank
            if (is_doing_comparison):
                other_data = MPI_Recv(other_rank)

                (smaller_half, larger_half) = merge(local_data, other_data)

                if (is_increasing):
                    local_data = smaller_half
                    MPI_Send(larger_half, other_rank)
                else:
                    local_data = larger_half
                    MPI_Send(smaller_half, other_rank)
            else:
                MPI_Send(local_data, other_rank)
                local_data = MPI_Recv(other_rank)
```

```
    return local_data

// Assumes data1 and data2 are the same size
function merge(data1, data2):
    array_size = sizeof(data1)

    lower_half = array size of array_size
    upper_half = array size of array_size

    index1 = index2 = 0

    while (index1 < array_size && index2 < array_size):
        output_index = index1 + index2
        choose_data1 = data1[index1] < data2[index2]

        if (choose_data1):
            value = data1[index1]
            index1++
        else:
            value = data2[index2]
            index2++

        if (output_index < array_size):
            lower_half[output_index] = value
        else:
            upper_half[output_index] = value

    // by this point we are guaranteed to be filling upper_half, since we have
completely gone through one of the input arrays
    while (index1 < array_size):
        output_index = index1 + index2
        upper_half[ouput_index] = data1[index1]
        index1++

    while (index2 < array_size):
        output_index = index1 + index2
        upper_half[ouput_index] = data2[index2]
        index2++

    return (lower_half, upper_half)

function main():
    // Initialize MPI
    MPI_Init()
    comm_size = MPI_Comm_size(MPI_COMM_WORLD)
    rank = MPI_Comm_rank(MPI_COMM_WORLD)

    // Get local data
    local_data = read_or_generate_data(rank, comm_size)

    // Sort
    local_data = bitonic_sort(local_data, comm_size, rank)

    // Verify
```

```
    verify_sorted(local_data, comm_size, rank)

    // End program
    MPI_Finalize()
```

MPI calls to be used:

```
MPI_Init()
MPI_Comm_size()
MPI_Comm_rank()
MPI_Send()
MPI_Recv()
MPI_Finalize()
```

Other functions:

```
sequential_sort(data) - exact algorithm isn't relevant
integer.bit(n) - get the value of the nth bit of the integer as a bool
integer.flip_bit(n) - returns an integer with the same bits, except the nth bit is
flipped
read_or_generate_data(rank, comm_size) - data generation function used for each
sorting algorithm (to be implemented later)
verify_sorted(local_data, comm_size, rank) - function to verify local data is
sorted and that this sequence is smaller than the one stored in the next highest
rank (to be implemented later)
```

Sample Sort

```
// s: number of samples, m: number of buckets
function partition(full_data, s, m)
    // get samples
    for sample = 0 to s-1:
        samples.append(get_random_element(full_data))
    quicksort(samples)

    //select splitters
    oversample = s/m
    splitters = [-inf]
    for splitter = 1 to m-1:
        splitters.append(samples[floor(oversample*splitter)])
    splitters.append(inf)

    //Put data into buckets based on splitters
    for element in full_data:
        find j | splitters[j]<element<=splitters[j+1]
```

```
        buckets[j].append(element)

function main(data, samples):

    MPI_Init()
    rank = MPI_Comm_rank(MPI_COMM_WORLD)
    size = MPI_Comm_size(MPI_COMM_WORLD)

    if (rank == MASTER):
        buckets = partition(data, samples, size)

    MPI_Scatter(send = buckets, recv = local_data, root=0)
    local_data = quicksort(local_data)
    MPI_Gather(send = local_data, recv = sorted_data, root=0)

    MPI_Finalize()
```

MPI calls to be used:

```
MPI_Init()
MPI_Comm_size()
MPI_Comm_rank()
MPI_Scatter()
MPI_Gather()
MPI_Finalize()
```

Merge Sort

```
function parallel_merge_sort(local_data, comm_size, rank):

    // Sort local data using sequential merge sort
    local_data = sequential_merge_sort(local_data)

    // Parallel merge phase
    for step = 1 to log2(comm_size):
        partner = rank XOR (1 << (step - 1))  // Find the partner process
        if rank < partner:
            // Send local data to the partner and receive its data
            MPI_Send(local_data, partner)
            received_data = MPI_Recv(partner)
            // Merge local and received data
            local_data = merge(local_data, received_data)
        else:
            // Send local data to the partner and receive its data
            MPI_Send(local_data, partner)
            received_data = MPI_Recv(partner)
```

```
            // Merge received data first to maintain order
            local_data = merge(received_data, local_data)

    return local_data

function main():

    // Initialize MPI
    MPI_Init()
    comm_size = MPI_Comm_size(MPI_COMM_WORLD)  // Get number of processes
    rank = MPI_Comm_rank(MPI_COMM_WORLD)       // Get process rank

    // Read or generate local data (each process generates or receives its own
data)
    local_data = read_or_generate_data(rank, comm_size)

    // Perform parallel merge sort
    sorted_local_data = parallel_merge_sort(local_data, comm_size, rank)

    // Gather all sorted data at root process
    if rank == 0:
        global_sorted_data = MPI_Gather(sorted_local_data, root=0)
    else:
        MPI_Gather(sorted_local_data, root=0)

    // Finalize MPI
    MPI_Finalize()
```

MPI calls to be used:

```
MPI_Init()
MPI_Comm_size()
MPI_Comm_rank()
MPI_Send()
MPI_Recv()
MPI_Gather()
MPI_Finalize()
```

Radix Sort

```
function prefix_sum(local_counts, global_counts, array_size, communicator):
    // get data from processes
    temp = array with size array_size
    MPI_Allreduce(local_counts, temp, array_size, MPI_INT, MPI_SUM, communicator)

    // compute prefix sums
    global_counts[0] = temp[0]
    for i from 1 to size:
```

```
        global_counts[i] = global_counts[i - 1] + temp[i]

function radix_sort(array, array_size):
    rank = MPI_Comm_rank(MPI_COMM_WORLD)
    numtasks = MPI_Comm_size(MPI_COMM_WORLD)

    // determine how much data to send to each task
    chunk_size = array_size / numtasks
    local_chunk = array of size chunk_size

    // scatter data to processes
    MPI_Scatter(array, chunk_size, MPI_INT, local_chunk, chunk_size, MPI_INT, 0,
MPI_COMM_WORLD)

    // find global max
    local_max = max_element(local_chunk.begin(), local_chunk.end())
    global_max = MPI_Allreduce(local_max, global_max, 1, MPI_INT, MPI_MAX,
MPI_COMM_WORLD)

    while exp from 1, global_max / exp > 0, exp *= 10:

        // count the number of each digit in the local chunk
        local_count is an array of size 10
        for i from 0 to chunk_size, i++:
            local_count[(local_chunk[i] / exp) % 10]++

        // compute prefix sum so we can find global positions
        global_count is an array of size 10
        prefix_sum(local_count, global_count, 10, MPI_COMM_WORLD)

        // determine new locations for elements
        new_chunk is an array of size chunk_size
        position is an array of size 10
        for i from 0 to 10:
            position[i] = global_count[i] - local_count[i]

        // rearrange elements to new positions
        for i from chunk_size-1 to 0:
            index = (local_chunk[i] / exp) % 10
            new_chunk[--position[index]] = local_chunk[i]

        // redistribute globally sorted elements to tasks
        MPI_Alltoall(new_chunk, chunk_size, MPI_INT, local_chunk, chunk_size,
MPI_INT, MPI_COMM_WORLD)


        // gather sorted chunks in root task
        if rank is 0:
            sorted_array is an array of size array_size
            MPI_Gather(local_chunk, chunk_size, MPI_INT, sorted_array, chunk_size,
MPI_INT, 0, MPI_COMM_WORLD)

        else:
            MPI_Gather(local_chunk, chunk_size, MPI_INT, NULL, chunk_size,
```

```
MPI_INT, 0, MPI_COMM_WORLD)

function main():
    // initialize MPI
    MPI_Init()
    rank = MPI_Comm_rank()
    size = MPI_Comm_size()

    // provide input and sort
    input is array to sort
    radix_sort(input)

    // finalize MPI
    MPI_Finalize()
```

MPI calls to be used:

```
MPI_Init()
MPI_Comm_rank()
MPI_Comm_size()
MPI_Finalize()
MPI_Allreduce()
MPI_Scatter()
MPI_Alltoall()
MPI_Gather()
```

Column Sort

```
// Function to extract a column from a 2D matrix
function get_column_data(local_data, col_index):
    // Initialize an empty list to store the column data
    column_data = []

    // Loop over each row in the local data
    for row in local_data:
        // Append the element at the column index to the column_data list
        column_data.append(row[col_index])

    return column_data

function parallel_column_sort(local_data, num_rows, num_cols, comm_size, rank):
    // Sort each column locally
    for col = 0 to num_cols - 1:
        local_column_data = get_column_data(local_data, col)
        sorted_column_data = sequential_sort(local_column_data) // can use any
efficient sequential search such as merge, bubble
        update_column(local_data, col, sorted_column_data)
```

```
    // Transpose the matrix
    local_data = transpose_local(local_data)

    // Perform all-to-all communication to redistribute columns as rows
    new_rows = MPI_Alltoallv(send_data=local_data,
send_counts=calculate_send_counts(rank, comm_size),
                             recv_data=new_matrix_space,
recv_counts=calculate_recv_counts(rank, comm_size))

    // Sort all new rows received
    for row = 0 to num_rows - 1:
        sorted_row = sequential_sort(new_rows[row])
        new_rows[row] = sorted_row

    // Transpose the matrix back
    local_data = transpose_local(new_rows)

    // Another all-to-all communication to redistribute original rows
    final_matrix = MPI_Alltoallv(send_data=local_data,
send_counts=calculate_send_counts(rank, comm_size),
                                 recv_data=final_matrix_space,
recv_counts=calculate_recv_counts(rank, comm_size))

    // Final local sort of each column again
    for col = 0 to num_cols - 1:
        local_column_data = get_column_data(final_matrix, col)
        sorted_column_data = sequential_sort(local_column_data)
        update_column(final_matrix, col, sorted_column_data)

    return final_matrix

function main():

    // Initialize MPI
    MPI_Init()
    comm_size = MPI_Comm_size(MPI_COMM_WORLD)  // Get number of processes
    rank = MPI_Comm_rank(MPI_COMM_WORLD)       // Get process rank

    // Setup matrix dimensions and generate local data
    num_rows, num_cols = determine_dimensions(comm_size)
    local_data = read_or_generate_data(num_rows, num_cols, rank)

    // Perform parallel column sort
    sorted_matrix = parallel_column_sort(local_data, num_rows, num_cols,
comm_size, rank)

    // Gather all sorted matrices at root process
    if rank == 0:
        global_sorted_matrix = MPI_Gather(sorted_matrix, root=0)
    else:
        MPI_Gather(sorted_matrix, root=0)

    // Finalize MPI
    MPI_Finalize()
```

MPI calls to be used

```
MPI_Init()
MPI_Comm_size()
MPI_Comm_rank()
MPI_Alltoallv()  // Used for transposing the matrix
MPI_Gather()
MPI_Finalize()
```

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

Input:

- Input Sizes
  - 2^{16}
  - 2^{18}
  - 2^{20}
  - 2^{22}
  - 2^{24}
  - 2^{26}
  - 2^{28}
- Input Types:
  - Sorted
  - Sorted with 1% perturbed
  - Random
  - Reverse sorted

Strong scaling (same problem size, increase number of processors/nodes)

- Fix problem size at 2^{24} elements
- Increase number of processors: 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024
- Measure and compare:
  - Total execution time
  - Speedup (T1 / Tn)
  - Parallel efficiency ((T1 / Tn) / n)

Weak scaling (increase problem size, increase number of processors)

- Start with 2^{16} elements per processor
- Increase both problem size and number of processors proportionally
  - (e.g., 2 processors: 2 x 2^{16}, 4 processors: 4 x 2^{16}, etc.)
- Measure and compare:

    - Execution time
    - Parallel efficiency

Performance Metrics (to be measured for all experiments):

- Total execution time
- Communication time
- Computation time
- Memory usage