CSCE 435 Group project

0. Group number: 2

1. Group members:

1. First: Aditya Biradar (Merge Sort)
2. Second: Eyad Nazir
3. Third: Eduardo Alvarez Hernandez
4. Fourth: Juan Carrasco

1a. Method of Communication:

- We will use normal phone messaging as our method of coomunication for this project

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

- For this project, we will be implementing various sorting algorithms for parallel computing and do a comparitive analysis of the algorithms to identify pros and cons of each algorithm.

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

- Architecture: For all the algorithms below we will be implementing them with an MPI architecture

- Bitonic Sort: This algorithm requires the size of the input to be a power of 2, it creates a bitonic sequence from the array before making comparisons and returning a sorted array. The runtime of this algorithm is N/P log^2(N/P), where P is the number of processors. This algorithm will be implemented by Juan Carrasco

- Sample Sort (Eyad Nazir): A parallel sorting algorithm that divides the input array into smaller subarrays, sorts them independently, and merges them back together.

- Merge Sort (Aditya Biradar): At its base the merge sort is considered a recursive algorithm, and usually has a runtime of n log(n). In this assignment we will be paralleizing this algorithm.

- Radix Sort (Eduardo Alvarez Hernandez): Radix Sort is a non-comparative sorting algorithm that processes individual digits of the numbers in a given list, sorting them based on place value. The runtime for Radix Sort is typically O(d*(n + k)), where d is the number of digits in the largest number, n is the number of elements, and k is the range of digits.

2b. Pseudocode for each parallel algorithm

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

- Bitonic Sort

```
```

```
// Pseudocode for Parallel Bitonic Sort using MPI

// Initialize MPI environment
MPI_Init()
rank = MPI_Comm_rank(MPI_COMM_WORLD)
size = MPI_Comm_size(MPI_COMM_WORLD)

// N is the total number of elements to sort
// P is the number of processors (P = size)
N_local = N / P   // Divide data evenly among processors

// Step 1: Local Sorting
// Generate or receive local data for each process
local_data = GetLocalData(rank, N_local)

// Perform local bitonic sort on each processor
// Bitonic sort on local data
BitonicSort(local_data, N_local)

// Step 2: Bitonic Merge Across Processors
// Start the parallel merging using the bitonic merge network
// Phase 1: Up-sweep phase
for stage in 1 to log2(P):
    for step in 0 to log2(N_local):
        partner = rank XOR 2^(stage - 1)  // Determine partner processor

        // Communicate with the partner process
        MPI_Sendrecv(local_data, N_local, partner)

        // Perform local comparison and merge
        if (rank < partner):
            // Merge for increasing order
            BitonicMerge(local_data, received_data, "ASCENDING")
        else:
            // Merge for decreasing order
            BitonicMerge(local_data, received_data, "DESCENDING")

// Step 3: Global Communication and Final Sort
// Continue merging until the entire sequence is sorted
for stage in 1 to log2(N):
    step = 2^stage
    for i in 0 to N_local - 1:
        // Compare and merge elements within each local block
        if (i % step == 0):
            BitonicMerge(local_data[i:i+step], "ASCENDING")

// Step 4: Gather the results from all processors
sorted_data = MPI_Gather(local_data, N_local, MPI_COMM_WORLD)

// Finalize MPI
MPI_Finalize()

// BitonicSort function: Sorts a sequence using bitonic sorting network
```

```
function BitonicSort(data, N_local):
    for k in 2 to N_local step *= 2:
        for j = k/2 down to 1:
            for i in 0 to N_local - 1:
                if ((i & k) == 0):
                    CompareAndSwap(data[i], data[i+j], "ASCENDING")
                else:
                    CompareAndSwap(data[i], data[i+j], "DESCENDING")

// BitonicMerge function: Merges two sorted sequences
function BitonicMerge(data, received_data, order):
    for i = 0 to N_local - 1:
        if (order == "ASCENDING" and data[i] > received_data[i]):
            Swap(data[i], received_data[i])
        else if (order == "DESCENDING" and data[i] < received_data[i]):
            Swap(data[i], received_data[i])
```

- Sample Sort

```

function sampleSort():
    # Step 1: Initialize MPI
    MPI.Init()
    rank = MPI.Comm_rank()        # Get process rank
    size = MPI.Comm_size()        # Get number of processes

    # Step 2: Root generates data
    if rank == 0:
        data = generate_data()    # Root generates full data set

    # Step 3: Calculate send counts and displacements (only root)
    if rank == 0:
        base_count = len(data) // size
        remainder = len(data) % size
        send_counts = [base_count + (1 if i < remainder else 0) for i in
range(size)]
        send_displs = [sum(send_counts[:i]) for i in range(size)]

    # Step 4: Broadcast send counts
    send_counts = MPI.Bcast(send_counts)

    # Step 5: Allocate space for local data
    local_data = allocate_array(send_counts[rank])  # Each process gets its
portion

    # Step 6: Scatter data to all processes
    MPI.Scatterv(data, local_data)

    # Step 7: Sort local data
    local_data.sort()

    # Step 8: Select local samples
    sample_gap = len(local_data) // (size - 1)
    local_samples = [local_data[i * sample_gap] for i in range(1, size)]

    # Step 9: Gather samples at root
    all_samples = MPI.Gather(local_samples)

    # Step 10: Root chooses splitters
    if rank == 0:
        all_samples.sort()
        splitters = [all_samples[i * (size - 1)] for i in range(1, size)]

    # Step 11: Broadcast splitters
    splitters = MPI.Bcast(splitters)

    # Step 12: Partition local data
    partitions = [[] for _ in range(size)]
    for elem in local_data:
        dest = find_partition(splitters, elem)
        partitions[dest].append(elem)

    # Step 13: Exchange partitions
    received_data = MPI.Alltoall(partitions)

    # Step 14: Sort received data
    received_data.sort()

    # Step 15: Gather sorted data at root
    sorted_data = MPI.Gatherv(received_data)

    # Step 16: Print final sorted data (root)
    if rank == 0:
        print(sorted_data)

    # Step 17: Finalize MPI
    MPI.Finalize()
```

- Merge Sort

```
    MergeSort():
        if((taskid == 0 )):
            //Master Process
            //Split the array into halves proportionate to the number of
processors
            split=len(array)//num of processors  **round down to nearest whole
number**

            //MPI_SCATTER scatter halves to processors(in)
            master_to_worker=MPI_SCATTER()

            //sort the local chunk in the master
            local_sort= splitter(local_array)
            //MPI Gather call to bring the sorted arrays back into one array
            worker_to_master=MPI_GATHER()

            //sort the final array from the gather
            final_array=MERGE_SORT(worker_to_master)


        if(taskid > 0 ):
            //Recieve the array from the scatter call
            MPI_SCATTER()

            //sort the array
            Splitter(recv array)

            // send result back to master
            MPI_GATHER()


        Splitter(array):
            If length == 0 || length == 1:
                return array
            split=len(array)//2 **round down to nearest whole number**
            lhs_array= array[:split]
            rhs_array= array[split:]


            lhs_sort=Splitter(lhs_array)
            rhs_sort=Splitter(rhs_array)

            return MERGE_SORT(lhs,rhs)

        MERGE_SORT(lhs,rhs):
            sort vector=[]

            while(length of lhs and rhs != 0):
                if lhs[0] < rhs[0]:
                    remove first element from lhs and append to sort
                else:
                    remove first element from rhs and append to sort

            if(len(lhs)!=0 and len(rhs)==0):
                append lhs array to sort

            if(len(rhs)!=0 and len(lhs)==0):
                append rhs array to sort

            return sort
```

- Radix Sort

```
    Function msd_radix_sort_mpi(data, digit_position, low, high, rank, size,
comm):
    If low >= high OR digit_position < 0:
        Return  # Base case: no more sorting needed

    # Step 1: Local histogram creation
    Initialize local_histogram[10] = {0}
    For i from low to high:
        local_histogram[get_digit(data[i], digit_position)] += 1

    # Step 2: Global histogram (MPI Reduce)
    Initialize global_histogram[10] = {0}
    MPI_Reduce(local_histogram, global_histogram, SUM, root = 0)

    # Step 3: Compute bucket positions (root process) and broadcast
    If rank == 0:
        cumulative_sum[0] = 0
        For i from 1 to 9:
            cumulative_sum[i] = cumulative_sum[i-1] + global_histogram[i-1]
    MPI_Bcast(cumulative_sum, root = 0)

    # Step 4: In-place bucket sorting
    i = low
    While i <= high:
        digit = get_digit(data[i], digit_position)
        correct_pos = cumulative_sum[digit]
        If i >= correct_pos AND i < correct_pos + global_histogram[digit]:
            i += 1
        Else:
            Swap data[i] with data[correct_pos]
            cumulative_sum[digit] += 1

    # Step 5: Recursively sort each bucket
    For digit from 0 to 9:
        next_start = cumulative_sum[digit] - global_histogram[digit]
        next_end = cumulative_sum[digit] - 1
        If next_start < next_end:
            msd_radix_sort_mpi(data, digit_position - 1, next_start, next_end,
rank, size, comm)


# Main function to initialize MPI and distribute the data
Function parallel_msd_radix_sort(data):
    MPI_Init()
    rank = MPI_Comm_rank(MPI_COMM_WORLD)
    size = MPI_Comm_size(MPI_COMM_WORLD)

    # Step 1: Scatter data across processors
    local_data_size = len(data) // size
    if rank == 0:
        local_data = np.array_split(data, size)
    else:
        local_data = None
    local_data = comm.scatter(local_data, root=0)

    # Step 2: Apply the MSD Radix Sort on the local data
    max_digit_position = get_max_digit_position(data)  # Find max digit length
    msd_radix_sort_mpi(local_data, max_digit_position, 0, len(local_data) - 1,
rank, size, comm)

    # Step 3: Gather sorted subarrays back to root
    sorted_data = comm.gather(local_data, root=0)

    if rank == 0:
        # Merge the results from all processors
        return np.concatenate(sorted_data)
```

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

- Input sizes, Input types:
  - For our input sizes, we will start from a small n size for our array and then progressively make it larger.
  - Input sizes: (2^{16}), (2^{18}), (2^{20}), (2^{22}), (2^{24}), (2^{26}), (2^{28})
  - Input types:
    - Sorted arrays
    - Reverse sorted arrays
    - Random arrays
    - 1% perturbed
- Strong scaling (same problem size, increase number of processors/nodes):
  - With this measurement, this could show diminishing returns as we try to find the optimized amount of processors for a given problem.
  - Number of processors/nodes: 2, 4, 8, 32, 64, 128, 256, 512, 1024
- Weak scaling (increase problem size, increase number of processors):
  - With this measurement, we could find the limit on each processor and how long it takes for something to compute among those algorithms.
  - Number of processors/nodes: 2, 4, 8, 32, 64, 128, 256, 512, 1024
- Run time():
  - With this measurement, we can compare the run times and although some algorithms inherently may be quicker than others, it's still good to compare to see how much extra time a certain algorithm could take to understand the costs associated with a given algorithm.