Report.md

CSCE 435 Group project

0. Group number: 2 (Honors)

1. Group members:

1. Kimberly Chen
2. Spencer Le
3. Suhu Lavu
4. Andrew Mao
5. Jeff Ooi

We will communicate via a messaging group chat.

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)

We will be comparing the following algorithms:

- Bitonic Sort - Kimberly Chen
- Sample Sort - Spencer Le
- Merge Sort - Suhu Lavu
- Radix Sort - Andrew Mao
- Column Sort - Jeff Ooi

We will use the Grace cluster on the TAMU HPRC.

2b. Pseudocode for each parallel algorithm

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

```
func bitonic_sort(matrix, lowIndex, count, direction)
    Initialize MPI
    rank <- MPI rank
    num_procs <- MPI size

    if count > 1
        k = count / 2  // Divide the array into two halves

        // Sort the first half in ascending order
        bitonicSort(matrix, lowIndex, k, 1)

        // Sort the second half in descending order
        bitonicSort(matrix, lowIndex + k, k, 0)

        // Merge the two halves according to the 'direction'
        bitonicMerge(arr[], lowIndex, count, direction)

        for step = 1 to log2(num_procs)
            partner <- rank XOR step

            // Exchange sorted halves with partner process and merge
            if direction == 1 // this means it is ascending
                if rank < partner
                    MPI send local_array to partner
                    MPI receive partner_array from partner
                    matrix <- bitonic_merge_ascending(matrix, partner_array)
                else
                    MPI send local_array to partner
                    MPI receive partner_array from partner
                    matrix <- bitonic_merge_descending(matrix, partner_array)
            else // this means it is descending
                if rank < partner
                    MPI send local_array to partner
                    MPI receive partner_array from partner
                    matrix <- bitonic_merge_descending(matrix, partner_array)
                else
                    MPI send local_array to partner
                    MPI receive partner_array from partner
                    matrix <- bitonic_merge_ascending(matrix, partner_array)
            end if
        end for
    end if

    // Gather all sorted data at master
```

```
        if rank is master
            for i: 1 -> num_procs
                MPI receive sorted segments from all processes
            end for
            output "Final sorted array"
        end if

        Finalize MPI
    end func
```

- Sample Sort Pseudocode

```
    func parallel_sample_sort(local_data):
        // Initialize MPI environment
        MPI_Init()

        // Get total number of processes and rank of each process
        int num_procs, rank
        MPI_Comm_size(MPI_COMM_WORLD, &num_procs)
        MPI_Comm_rank(MPI_COMM_WORLD, &rank)

        // Step 1: Sort local data locally
        local_data = sort(local_data)

        // Step 2: Choose 'p' samples from the sorted local data
        samples = select_samples(local_data, num_procs)

        // Step 3: Gather all samples to the root process
        all_samples = []
        MPI_Gather(samples, num_procs, MPI_INT, all_samples, num_procs, MPI_INT, roc

        // Step 4: Root process sorts the gathered samples and selects splitters
        splitters = []
        if rank == 0:
            all_samples_sorted = sort(all_samples)
            splitters = select_splitters(all_samples_sorted, num_procs)

        // Step 5: Broadcast splitters to all processes
        MPI_Bcast(splitters, num_procs, MPI_INT, root=0, MPI_COMM_WORLD)

        // Step 6: Each process redistributes its local data based on the splitters
        buckets = distribute_data(local_data, splitters, num_procs)

        // Step 7: Each process sends its buckets to the appropriate processes
        for i = 0 to num_procs - 1:
            // Send and receive buckets to/from each process
            send_bucket_to_process(buckets[i], i)
            receive_bucket_from_process(i)
```

```
    // Step 8: Merge the received buckets
    local_data = merge(received_buckets)

    // Step 9: Perform local sort on the merged data
    local_data = sort(local_data)

    // Step 10: Gather sorted data on the root process
    sorted_data = []
    MPI_Gather(local_data, len(local_data), MPI_INT, sorted_data, len(local_data

    // Finalize MPI environment
    MPI_Finalize()

    return sorted_data
end func
```

- Merge Sort Pseudocode

```
func parallel_merge_sort(array, n):
    Initialize MPI
    rank <- MPI rank
    size <- MPI size

    # calculate size of worker array and create local copy
    worker_size = n / size
    worker_arr = new array[worker_size]

    # send real array to all processes
    MPI_Scatter(array, worker_size, MPI_INT, worker_arr, worker_size, MPI_INT, r

    # sort local copy
    sorted_arr = merge_sort(worker_arr)

    # merge sorted pieces back together
    step = 1
    while step < size:
        if rank % (2 * step) == 0:
            if rank + step < size:
                # get sorted array from another process
                received_size = worker_size * step
                received_array = new array[received_size]
                MPI_Recv(received_array, received_size, MPI_INT, source=rank + s

                # merge received array with local array
                sorted_arr = merge(sorted_arr, received_array)
                worker_size += received_size
```

```
        else:
            # send sorted array to another process
            target = rank - step
            MPI_Send(sorted_arr, worker_size, MPI_INT, dest=target, tag=0, MPI_C
            break out of loop
        step *= 2

    # master process gathers the final sorted array
    if rank == 0:
        MPI_Gather(sorted_arr, worker_size, MPI_INT, array, worker_size, MPI_INT

    Finalize MPI
end func

func merge(arr1, arr2):
    size_1 = size(arr1)
    size_2 = size(arr2)
    final = new array[size_1 + size_2]
    i, j, k = 0

    while i < size_1 and j < size_2:
        if arr1[i] < arr2[j]:
            final[k] = arr1[i]
            i += 1
        else:
            final[k] = arr2[j]
            j += 1
        k += 1

    # copy leftover elements
    while i < size_1:
        final[k] = arr1[i]
        i += 1
        k += 1

    while j < size_2:
        final[k] = arr2[j]
        j += 1
        k += 1

    return final
end func

func merge_sort(arr):
    # base case
    if size(arr) <= 1:
        return arr

    # find midpoint
```

```
    mid = size(arr) / 2

    # sort left array
    left_sorted = merge_sort(arr[0:mid])

    # sort right array
    right_sorted = merge_sort(arr[mid + 1:])

    # merge them together
    return merge(left_sorted, right_sorted)
end func
```

- Radix Sort Pseudocode

```
func radix_sort(matrix, lowIndex, count, direction)
    Initialize MPI
    rank <- MPI rank
    num_procs <- MPI size

        keysPerWorker <- numKeys / num_procs

        if rank is master:
            for i: 1 -> num_procs - 1:
                MPI send arr[i * keysPerWorker to (i + 1) * keysPerWorker - 1] t
            end for
        end if

        localbuffer <- [keysPerWorker]

        if rank is worker:
            MPI receive arr[i * keysPerWorker to (i + 1) * keysPerWorker - 1] ir
        end if

        # For each iteration, process g bits at a time (Radix Sorting)
        max_bits <- get bits max(arr)
        for i: 0 -> (max_bits / g) do:
            local_bucket_counts <- counting_sort(localbuffer, g, i)

            # Aggregate the result of counting sort, make global count of keys p
            global_bucket_counts <- MPI all_to_all(local_bucket_counts)
            prefix_sums <- get prefix sums of global_bucket_counts
            bucketed_keys <- distribute_keys(localbuffer, g, i, prefix_sums)

            # update these results to preserve order
            exchanged_keys <- MPI all_to_all_exchange(bucketed_keys)
            sort_by_g_bits(exchanged_keys, g, i)
            localbuffer <- exchanged_keys
```

```
        end for

        if rank is worker:
            MPI send localbuffer to master
        end if

        if rank is master:
            for i: 1 -> num_procs - 1 do:
                MPI receive sorted buffer into arr
            end for

        Finalize MPI
    end func
```

- Column Sort Pseudocode

```
Assumption: numRows >= 2 * (numCols - 1)^2
func column_sort(matrix, numRows, numCols)
    Initialize MPI
    rank <- MPI rank
    num_procs <- MPI size

    colsPerWorker = numCols / num_procs

    if rank is master
        for i: 1 -> num_procs
            MPI send matrix columns i * colsPerWorker to (i + 1) * colsPerWorker
        end for

    localbuffer <- buffer with enough size to fit matrix columns i * colsPerWork
    if rank is not master
        MPI receive matrix columns i * colsPerWorker to (i + 1) * colsPerWorker

    sort every column in localbuffer

    if rank is not master
        MPI send sorted localbuffer to master
    if rank is master
        for i: 1 -> num_procs
        MPI receive all localbuffers into matrix
        end for

        transpose matrix

        for i: 1 -> num_procs
            MPI send matrix columns i * colsPerWorker to (i + 1) * colsPerWorker
        end for
```

```
if rank is not master
    MPI receive matrix columns i * colsPerWorker to (i + 1) * colsPerWorker

sort every column in localbuffer

if rank is not master
    MPI send sorted localbuffer to master
if rank is master
    for i: 1 -> num_procs
        MPI receive all localbuffers into matrix
    end for

    untranspose matrix

    for i: 1 -> num_procs
        MPI send matrix columns i * colsPerWorker to (i + 1) * colsPerWorker
    end for

if rank is not master
    MPI receive matrix columns i * colsPerWorker to (i + 1) * colsPerWorker

sort every column in localbuffer

if rank is not master
    MPI send sorted localbuffer to master
if rank is master
    for i: 1 -> num_procs
        MPI receive all localbuffers into matrix
    end for

    shift matrix

    for i: 1 -> num_procs
        MPI send matrix columns i * colsPerWorker to (i + 1) * colsPerWorker
    end for

if rank is not master
    MPI receive matrix columns i * colsPerWorker to (i + 1) * colsPerWorker

sort every column in localbuffer

if rank is not master
    MPI send sorted localbuffer to master
if rank is master
    sort last column in matrix

    unshift matrix // matrix should now be sorted in Column-Major order
```

```
            for i: 1 -> num_procs
                MPI send matrix columns i * colsPerWorker to (i + 1) * colsPerWorker
            end for
        if rank is not master
            MPI receive matrix columns i * colsPerWorker to (i + 1) * colsPerWorker

        check if localbuffer is sorted
        sorted <- 1 if sorted, 0 if not
        if rank is master
            isSorted <- -1
            MPI reduce sorted to master with minimum sorted into isSorted
            if isSorted is 1
                output "matrix sorted"
            else
                output "matrix not sorted"

        Finalize MPI
    end func
```

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

We will measure and compare the sorting times of:

- Different input sizes and input types
  - Input sizes are 2^16, 2^18, 2^20, 2^22, 2^24, 2^26, and 2^28
  - Input types are sorted, random, reverse sorted, and 1% perturbed
- Strong scaling (same problem size, increase number of processors)
  - Number of processes are 2, 4, 8, 16, 32, 64, 128, 256, 512, and 1024
- Weak scaling (increase problem size, increase number of processors)
  - Number of processes are 2, 4, 8, 16, 32, 64, 128, 256, 512, and 1024

We will collect them using Caliper and compare them using Thicket.