CSCE 435 Group project

0. Group number: 9

1. Group members:

1. Rahul Singh
2. Kevin Thomas
3. Anthony Ciardelli
4. Brandon Thomas

Team Communication:

We will be using iMessage as our primary method of communication. We will share documents and information via Google Docs.

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

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

- Bitonic Sort: A bitonic sort implementation parallelizes the butterfly-pattern comparisons of bitonic sequences across processors, where each stage doubles the size of sorted subsequences by having processors exchange data with partners at varying "distances" (rank offsets), performing compare-and-swap operations locally, until the entire sequence is sorted in ascending or descending order across all processors.

- Sample Sort: The algorithm begins by distributing data evenly across processes. Each process then sorts its local data and selects regular samples, which are gathered and used to choose splitters that partition the data range. The splitters are used to redistribute the data using all-to-all communication so each process receives elements in its designated range, followed by a final local sort on each process.

- Merge Sort: This algorithm is a parallelized implementation of merge sort. The global array is divided into a number of segments that match the number of processes. Then, merge sort is run on each of the processes independently. It recursively divides the subarrays until there is one element in each subarray. Then, it procedurally merges and sorts each subarray until every element has been sorted. After each segment is sorted, a final merge is performed to create the fully sorted array.

- Radix Sort: This implementation performs parallel radix sort by first distributing data across processors based on the most significant bits (determined by the number of processors). Each processor then performs a local radix sort using counting sort on 8-bit chunks, with the final result remaining distributed across the processors in sorted order.

2b. Pseudocode for each parallel algorithm

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

Sample Sort:

```
function sampleSort(local_data, world_rank, world_size):
    // Start Caliper measurement for entire Sample Sort

    // 1. Local sorting phase
    // Start Caliper measurement for local sort
    sort(local_data)
    // End Caliper measurement for local sort

    // 2. Sample selection phase
    // Start Caliper measurement for sample selection
    samples = select_samples(local_data, world_size)
    // End Caliper measurement for sample selection

    // 3. Global splitter selection phase
    // Start Caliper measurement for global splitter selection
    splitters = gather_and_select_global_splitters(samples, world_rank,
world_size)
    // End Caliper measurement for global splitter selection

    // 4. Data partitioning phase
    // Start Caliper measurement for data partitioning
    partitioned_data = partition_data(local_data, splitters)
    // End Caliper measurement for data partitioning

    // 5. All-to-all exchange phase
    // Start Caliper measurement for all-to-all exchange
    received_data = all_to_all_exchange(partitioned_data, world_rank, world_size)
    // End Caliper measurement for all-to-all exchange

    // 6. Final local sorting phase
    // Start Caliper measurement for final local sort
    sort(received_data)
    // End Caliper measurement for final local sort

    // End Caliper measurement for entire Sample Sort
    return received_data

function main():
    // Initialize MPI
    // Get world_rank and world_size
    // Read or generate input data

    sorted_data = sampleSort(local_data, world_rank, world_size)

    // Gather results to rank 0 or write to file
    // Finalize MPI
```

Bitonic Sort

```
// Function to compare and exchange two elements
function compareExchange(data, i, j, ascending):
    // Compare elements at indices i and j
    // Swap if they are in the wrong order based on 'ascending' flag

// Function to merge a bitonic sequence
function bitonicMerge(data, low, count, ascending):
    // If count > 1:
    //     Find the greatest power of 2 less than count
    //     Compare-exchange pairs of elements
    //     Recursively merge the two halves

// Function to generate a bitonic sequence
function bitonicSort(data, low, count, ascending):
    // If count > 1:
    //     Recursively sort first half in ascending order
    //     Recursively sort second half in descending order
    //     Merge the resulting bitonic sequence

// Main MPI Bitonic Sort function
function mpiBitonicSort(local_data, world_rank, world_size):
    total_size = local_data.size() * world_size
    local_size = local_data.size()

    // Bitonic sort stages
    for k = 2 to total_size:
        for j = k/2 to 1 (dividing by 2 each iteration):
            // Start Caliper measurement for local compare-exchange
            for each element i in local_data:
                // Calculate the index of the element to compare with
                // If the partner element is in the same process:
                //     Perform local compare-exchange
                // Else:
                //     Start Caliper measurement for MPI exchange
                //     Use MPI_Sendrecv to exchange and compare elements with
partner process
                //     End Caliper measurement for MPI exchange
            // End Caliper measurement for local compare-exchange

// Main function
function main():
    // Initialize MPI
    // Get world_rank and world_size
    // Read or generate input data

    // Start Caliper measurement for entire MPI Bitonic Sort
    mpiBitonicSort(local_data, world_rank, world_size)
    // End Caliper measurement for entire MPI Bitonic Sort

    // Gather results to rank 0 or write to file
    // Finalize MPI
```

Merge Sort

```
// Function to merge two sorted arrays
function merge(left_array, right_array):
    // Merge the two arrays and return the result

// Function to perform local merge sort
function localMergeSort(array):
    // If array size > 1:
    //     Divide array into two halves
    //     Recursively sort both halves
    //     Merge the sorted halves

// Main MPI Merge Sort function
function mpiMergeSort(local_data, world_rank, world_size):
    // Start Caliper measurement for local sort

    // Perform local merge sort on this process's data
    localMergeSort(local_data)

    // End Caliper measurement for local sort

    // Parallel merge phase
    for step = 1 to log2(world_size):
        if this process should receive data:
            // Start Caliper measurement for MPI receive and merge

            // Receive data from partner process
            // Merge received data with local data

            // End Caliper measurement for MPI receive and merge
        else if this process should send data:
            // Start Caliper measurement for MPI send

            // Send local data to partner process

            // End Caliper measurement for MPI send

            // Break out of the loop as this process is done

    return local_data

// Main function
function main():
    // Initialize MPI

    // Get world_rank and world_size

    // Read or generate input data

    // Start Caliper measurement for entire MPI Merge Sort
    sorted_data = mpiMergeSort(local_data, world_rank, world_size)

    // End Caliper measurement for entire MPI Merge Sort

    // Gather results to rank 0 or write to file

    // Finalize MPI
```

Radix Sort

```
// Function to perform counting sort for a specific digit
function countingSort(array, exp):
    // Start Caliper measurement for counting
    // Count occurrences of each digit in the given place value
    // End Caliper measurement for counting

    // Calculate cumulative count
    // Build the output array

    // Start Caliper measurement for copying output
    // Copy the output array to the original array
    // End Caliper measurement for copying output

// Main MPI Radix Sort function
function mpiRadixSort(local_data, world_rank, world_size):
    // Find local maximum element

    // Start Caliper measurement for MPI reduce
    // Use MPI_Allreduce to find global maximum across all processes
    // End Caliper measurement for MPI reduce

    // Perform Radix Sort
    for exp = 1 to max_element:
        // Start Caliper measurement for local counting sort
        countingSort(local_data, exp)
        // End Caliper measurement for local counting sort

        // Start Caliper measurement for MPI all-to-all
        // Redistribute data across processes based on current digit
        // Use MPI_Alltoallv for flexible data redistribution
        // End Caliper measurement for MPI all-to-all

        // Start Caliper measurement for merging sorted chunks
        // Merge the received sorted chunks
        // End Caliper measurement for merging sorted chunks

// Main function
function main():
    // Initialize MPI
    // Get world_rank and world_size
    // Read or generate input data
    // Start Caliper measurement for entire MPI Radix Sort
    mpiRadixSort(local_data, world_rank, world_size)
    // End Caliper measurement for entire MPI Radix Sort

    // Gather results to rank 0 or write to file
    // Finalize MPI
```

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

- Input sizes, Input types
- 2^{16}, 2^{18}, 2^{20}, 2^{22}, 2^{24}, 2^{26}, 2^{28}; Sorted, Random, Reverse sorted, 1% perturbed
- Strong scaling (same problem size, increase number of processors/nodes)
- 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 processors
- Weak scaling (increase problem size, increase number of processors)
- 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 processors

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.

Sample Sort Calltree: (2^{23} elements, 8 processors, random input)

```
1.646 main
├─ 0.032 MPI_Comm_dup
├─ 0.000 MPI_Comm_free
├─ 0.000 MPI_Finalize
├─ 0.000 MPI_Finalized
├─ 0.000 MPI_Init
├─ 0.000 MPI_Initialized
├─ 0.276 comm
│  ├─ 0.199 comm_large
│  │  ├─ 0.004 MPI_Alltoallv
│  │  ├─ 0.036 MPI_Gatherv
│  │  └─ 0.159 MPI_Scatter
│  └─ 0.076 comm_small
│     ├─ 0.000 MPI_Alltoall
│     ├─ 0.002 MPI_Bcast
│     └─ 0.074 MPI_Gather
├─ 0.782 comp
│  ├─ 0.782 comp_large
│  └─ 0.000 comp_small
├─ 0.024 correctness_check
└─ 0.172 data_init_runtime
```

Radix Sort Calltree: (2^{16} elements, 4 processors, sorted input)

```
0.003 main
├─ 0.000 data_init_runtime
├─ 0.001 MPI_Barrier
├─ 0.001 comp
│  ├─ 0.000 comp_small
│  └─ 0.001 comp_large
├─ 0.000 comm
│  ├─ 0.000 comm_small
│  │  └─ 0.000 MPI_Scatter
│  └─ 0.000 comm_large
│     ├─ 0.000 MPI_Send
│     └─ 0.000 MPI_Recv
└─ 0.000 correctness_check
   ├─ 0.000 MPI_Allreduce
   ├─ 0.000 MPI_Send
   └─ 0.000 MPI_Recv
0.000 MPI_Finalize
0.000 MPI_Initialized
0.000 MPI_Finalized
0.001 MPI_Comm_dup
```

Merge Sort Calltree: (2^{22} elements, 8 processors, random input)

```
2.613 main
├─ 0.590 MPI_Comm_dup
├─ 0.000 MPI_Comm_free
├─ 0.000 MPI_Finalize
├─ 0.000 MPI_Finalized
├─ 0.000 MPI_Init
├─ 0.000 MPI_Initialized
├─ 0.115 comm
│  └─ 0.115 comm_large
│     ├─ 0.003 MPI_Barrier
│     ├─ 0.002 MPI_Gather
│     └─ 0.110 MPI_Scatter
├─ 1.408 comp
│  └─ 1.408 comp_large
├─ 0.028 correctness_check
└─ 0.123 data_init_runtime
```

Bitonic Sort Calltree: (2^{22} elements, 8 processors, random input)

```
9.100 main
├─ 0.000 MPI_Init
├─ 6.717 MPI_Reduce
├─ 0.107 comm
│  └─ 0.107 comm_large
│     ├─ 0.002 MPI_Gather
│     └─ 0.105 MPI_Scatter
├─ 0.843 comp
│  └─ 0.843 comp_large
├─ 0.031 correctness_check
```

```
└─ 0.097 data_init_runtime
0.027 MPI_Comm_dup
0.000 MPI_Finalize
0.000 MPI_Finalized
0.000 MPI_Initialized
```

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 the job
adiak::clustername();   // Name of the cluster
adiak::value("algorithm", algorithm); // The name of the algorithm you are using (e.g., "merge", "bitonic")
adiak::value("programming_model", programming_model); // e.g. "mpi"
adiak::value("data_type", data_type); // The datatype of input elements (e.g., double, int, float)
adiak::value("size_of_data_type", size_of_data_type); // sizeof(datatype) of input elements in bytes (e.g., 1, 2, 4)
adiak::value("input_size", input_size); // The number of elements in input dataset (1000)
adiak::value("input_type", input_type); // For sorting, this would be choices: ("Sorted", "ReverseSorted", "Random", "1_perc_perturbed")
adiak::value("num_procs", num_procs); // The number of processors (MPI ranks)
adiak::value("scalability", scalability); // The scalability of your algorithm. choices: ("strong", "weak")
adiak::value("group_num", group_number); // The number of your group (integer, e.g., 1, 10)
adiak::value("implementation_source", implementation_source); // Where you got the source code of your algorithm. choices: ("online", "ai", "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.

Metadata we will be collection by sorting algorithm

Merge Sort Metadata

```
adiak::init(NULL);
adiak::launchdate();
adiak::libraries();
adiak::cmdline();
adiak::clustername();
```

```
adiak::value("algorithm", "Merge");
adiak::value("programming_model", "MPI");
adiak::value("data_type", "I");
adiak::value("size_of_data_type", sizeof(int));
adiak::value("input_size", total_size);
adiak::value("input_type", input_type);
adiak::value("num_procs", num_procs);
adiak::value("scalability", "strong");
adiak::value("group_num", 9);
adiak::value("implementation_source", "online");
```

Sample Sort Metadata

```
adiak::init(NULL);
adiak::launchdate();
adiak::libraries();
adiak::cmdline();
adiak::clustername();
adiak::value("algorithm", "Radix");
adiak::value("programming_model", "MPI");
adiak::value("data_type", "I");
adiak::value("size_of_data_type", sizeof(int));
adiak::value("input_size", total_size);
adiak::value("input_type", input_type);
adiak::value("num_procs", num_procs);
adiak::value("scalability", "strong");
adiak::value("group_num", 9);
adiak::value("implementation_source", "online");
```

Radix Sort Metadata

```
adiak::init(NULL);
adiak::launchdate();
adiak::libraries();
adiak::cmdline();
adiak::clustername();
adiak::value("algorithm", "Sample");
adiak::value("programming_model", "MPI");
adiak::value("data_type", "I");
adiak::value("size_of_data_type", sizeof(int));
adiak::value("input_size", total_size);
adiak::value("input_type", input_type);
adiak::value("num_procs", num_procs);
adiak::value("scalability", "strong");
adiak::value("group_num", 9);
adiak::value("implementation_source", "online");
```

Bitonic Sort Metadata

```
adiak::init(NULL);
adiak::launchdate();
adiak::libraries();
adiak::cmdline();
adiak::clustername();
adiak::value("algorithm", "Radix");
adiak::value("programming_model", "MPI");
adiak::value("data_type", "I");
adiak::value("size_of_data_type", sizeof(int));
adiak::value("input_size", total_size);
adiak::value("input_type", input_type);
adiak::value("num_procs", num_procs);
adiak::value("scalability", "strong");
adiak::value("group_num", 9);
adiak::value("implementation_source", "online");
```

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

Merge Sort Graphs and Explanations

For this implementation, on the smaller array sizes, the parallel overhead caused the time to increase. However, as the array size and number of processors increased, the time to sort the array decreased. When processor amounts began to get into the hundreds the parallel overhead started to take over again, and made the program slower. This can be seen in the graphs where the processor values increase the comm times per rank also increase. As the processors increased, the comp values also decreased, supporting that each processor will have less to compute as more get added.

I was unable to get the 512 and 1024 cali files due to issues with the Grace queue and hydra errors. However, all other cali files are finished.

Radix Sort Graphs and Explanations

I have not been able to generate the graphs due to the added complexity of another variable (3d vs 2d in previous labs) however I did notice a decrease in time of the alhorithm as the processor count increased. While there was an increase when moving from 1 node to many due to the network cost overall the time to completion still decreased. Overall the sorted and perturbed input types performed very simmilarly due to the data being roughly the same. The next slowest was random where roughly 1 / number of processors data stayed local to a processor where everything else got moved to another processor. Lastly the slowest was reverse sorted as every piece of data had to be transfered to another processor causing significant communication times.

I was unable to generate the files for reversed sorted as the time to completion was significantly longer than the other three input types due to the excessive ammount of communication that has to occur at the splitting stage of the algorithm.

all caliper files can be found at radix_sort/caliper_files

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