CSCE 435 Group project

0. Group number: 18

1. Group members:

1. Anna Hartman
2. Tate Moskal
3. Nicole Hernandez
4. Vasudev Agarwal

Our team will communicate primarily through Text message for real-time discussions and updates. We will also use GitHub to track code contributions and Trello for project management, ensuring transparency and accountability for task assignments.

2. Parallel Sorting Algorithms

2a. Brief project description

In this project, we will be comparing the efficiencies of different sorting algorithms running in parallel. The algorithms will be implemented and tested on parallel architectures, such as multi-core processors and possibly distributed systems. Each team member will focus on one of the following algorithms:

- Bitonic Sort: Anna Hartman
- Sample Sort: Vasudev Agarwal
- Merge Sort: Nicole Hernandez
- Radix Sort: Tate Moskal

The comparison will assess execution time, we will evaluate the efficiency on different input sizes (small, medium, large) to measure scalability, and analyzing how these algorithms leverage parallelism to improve sorting performance.

2b. Pseudocode for each parallel algorithm

2b.1 Bitonic Sort

In Bitonic sort, a bitonic sequence is built (a sequence that first increases and then decreases) and sorted by merging. Within multiple threads, chunks of the given sequence are sorted into bitonic order. The merging process is carried out in parallel, merging each piece into a large bitonic sequence. Threads should be synchronized. The entire algorithm is carried out recursively in order to build the final bitonic sequence.

1. Divide the given array to be sorted into parallel chunks with corresponding threads
2. For each thread ands its piece (in parallel): sort the piece into bitonic order by recursively splitting the piece into two halves and sorting the first half into ascending order and the second into descending order, and then merging the two haves into a bitonic sequence.
3. Synchronize threads by ensuring each thread has completed its sorting before continuing on.
4. Merge all of the pieces bitonically (in parallel), for each chunk and thread: compare and swap such that if the current chunk is in the lower half, merge it in ascending order, and if it is in the upper half,

merge it in descending order. Recursively merge the two halves of the sequence to ensure they are fully sorted in either ascending or descending order. This should be carried out log(array_size) times, as each chunk doubles in size after being recursively merged.
5. Output the result.

Actual Code Algorithm Description:

1. Initializes the MPI environment with MPI_Init, retrieving the rank of each process and the total number of processes using MPI_Comm_rank and MPI_Comm_size. The root process (rank 0) parses command-line arguments to determine the size of the array (as a power of 2) and the number of processes to be used.
2. Root process (rank 0) generates a random array of integers of size 2^exponent. This size is calculated from the exponent passed as an argument. The root process seeds the random number generator with the current time to ensure different data for each run.
3. Root process then divides the array into equal-sized chunks and distributes these chunks to all processes, including itself, using MPI_Scatter. Each process receives a local sub-array (of size total array size / number of processes) for sorting, ensuring that the sorting workload is spread evenly across all processes.
4. Once each process receives its portion of the array, it applies the Bitonic Sort algorithm locally. the Bitonic Sort works as follows: a. Recursively divides the sub-array into smaller parts. b. Sorts the smaller parts in ascending and descending order alternately. c. Merges these parts using the Bitonic Merge step, which compares and swaps elements in such a way that the sub-array becomes sorted.
5. After sorting their local sub-arrays, each process sends its sorted sub-array back to the root process using MPI_Gather. The root process collects all the sorted sub-arrays into the original array.
6. Once all sub-arrays are gathered, the root process performs a final Bitonic Sort on the entire array to merge the sorted sub-arrays into a fully sorted array. This final step is necessary because the sub-arrays are only partially sorted, and a global sort ensures that the entire array is in the correct order.
7. The root process then finally checks if the final array is correctly sorted by comparing each element with the next. If any element is out of order, it reports an error; otherwise, it confirms that the array is sorted.
8. Finally, the program shuts down the MPI environment with MPI_Finalize, cleaning up all resources used by MPI.

2b.2 Sample Sort

In sample sort a large dataset is divided into smaller paritions, after which each partition is sorted independently, and then these sorted paritions are merged to obtain the final sorted result.

1. Splitting the given dataset into equally divided smaller segments, where each segement is given to a processor
2. Run basic sorting algorithm on each of the segments, where each processor is handling its piece independently.
3. From these sorted segments, each processor selects few samples. then, MPI communication is used to collect samples from all processors
4. Sort the selected samples, which will help us establish a global order among the samples.
5. From the sorted samples, we pick few speical elements. which act as a pivot. which are shared with all processors. MPI_Bcast is used to broadcast the pivots to all processors.

6. Each processor takes its ordered segment and divides it into subsegments based on the choicen pivot.
7. Now, processors share their ordered segments globally with the corresponding processor based on the segment number. using MPI_Alltoall
8. Finally, each processor merges and sorts the recieved elements.

2b.3 Merge Sort

Parallel Merge Sort uses the divide and conquer technique, recursively dividing the dataset into smaller parts, sorting them, and merging the results where in parallel ver. it is distributed across multiple processors. Parallel Merge Sort will be implemented using MPI. Each processor will perform its own sorting idenpendently, merging sorted data using MPI communication.

1. Start MPI for Communication between processors.
2. Identify if the process is the master (rank 0) or worker (rank > 0)
3. The master splits the dataset into smaller parts
4. The master sends each part to a worker process
5. Each worker process sorts its part of the dataset independently
6. Workers send their sorted parts back to the master
7. The master merges all sorted parts into one sorted array
8. Close MPI after sorting is complete

2b.4 Radix Sort

Radix Sort is an algorithm that sorts by processing through individual digits, sorting along the way. The process can be sped up by allowing each processor to handle a portion of the total array. By sorting a subarray and keeping note of the order of subarray chunks being sorted in each processor, they can be placed accordingly back into the main array. While this example sorts via binary, the process can account for numbers of any base as long as the # of arrays corresponds correctly.

1. Initialize MPI for multiprocessor communication
2. Convert array digits into binary (helps with initial implementation).
3. Find the maximum element and its # of digits.
4. Begin iterating through digits starting at the least significant digit up to the maximum digit significance.
5. Split the array into subarrays depending on the # of processors used and send to workers, keeping track of the order in which each subarray gets sent where.
6. Each worker will sort its subarray into 2 arrays, the first with digits that are 0, the second with digits that are 1.
7. Worker returns arrays and master combines the 0 array in order of worker process, then repeats for the 1 array.
8. Repeat with the next digit until all digits places have been parsed.
9. End MPI once complete.

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

- Input sizes, Input types (Alter input array to be: random, sorted, reverse sorted,...)
- Strong scaling (same problem size, increase number of processors/nodes)

- Weak scaling (increase problem size, increase number of processors)
- Parallelization strategies (master/worker vs SPMD, calculating speedup and runtime differences)
- Communication strategies (collectives vs point-to-point, measure runtime differences between code for each communication strategy)

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
```

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.

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

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