CSCE 435 Group project

0. Group number:

1. Group members:

1. Evan Burriola
2. Min Zhang
3. Cole McAnelly
4. Saddy Khakimova

All communication between team members will be coordinated through Discord.

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

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

For this project we will be comparing the following sorting algortithms:

- Merge sort
- Quick sort
- Bitonic sort
- Odd even sort

Each sorting algorithm will be implemented two ways: MPI and CUDA

2b. Pseudocode for each parallel algorithm

- For MPI programs, include MPI calls you will use to coordinate between processes
- For CUDA programs, indicate which computation will be performed in a CUDA kernel, and where you will transfer data to/from GPU

MPI Merge Sort

CUDA Merge Sort

MPI Quick Sort

CUDA Quick Sort

MPI Bitonic Sort

CUDA Bitonic Sort

MPI Odd Even Sort

CUDA Odd Even Sort

```
int main()
    create unsorted array

    cudaMalloc to allocate memory for array

    cudaMemcpy to transfer array to GPU

    for (int i = 0; i < NUM_VALS; i++) {
        odd_even_sort<<<BLOCKS, THREADS>>>(dev_values, NUM_VALS);
    }

    check if the array is sorted
```

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

- Input sizes, Input types
- Strong scaling (same problem size, increase number of processors/nodes)
- Weak scaling (increase problem size, increase number of processors)
- Number of threads in a block on the GPU

3. Project implementation

Implement your proposed algorithms, and test them starting on a small scale.
Instrument your code, and turn in at least one Caliper file per algorithm;
if you have implemented an MPI and a CUDA version of your algorithm,
turn in a Caliper file for each.

3a. Caliper instrumentation

Please use the caliper build `/scratch/group/csce435-f23/Caliper/caliper/share/cmake/caliper` (same as lab1 build.sh) to collect caliper files for each experiment you run.

Your Caliper regions should resemble the following calltree
(use `Thicket.tree()` to see the calltree collected on your runs):

```
main
|_ data_init
|_ comm
|    |_ MPI_Barrier
|    |_ comm_small  // When you broadcast just a few elements, such as splitters in Sample sort
|    |   |_ MPI_Bcast
|    |   |_ MPI_Send
|    |   |_ cudaMemcpy
|    |_ comm_large  // When you send all of the data the process has
|        |_ MPI_Send
|        |_ MPI_Bcast
|        |_ cudaMemcpy
|_ comp
|    |_ comp_small  // When you perform the computation on a small number of elements, such as s
|    |_ comp_large  // When you perform the computation on all of the data the process has, such
|_ correctness_check
```

Required code regions:

- `main` - top-level main function.
  - `data_init` - the function where input data is generated or read in from 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.

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 - all computation code regions should be nested in the "comp" parent code region as following:

```
CALI_MARK_BEGIN("comp");
CALI_MARK_BEGIN("comp_large");
mergesort();
CALI_MARK_END("comp_large");
CALI_MARK_END("comp");
```

Looped GPU kernels - to time GPU kernels in a loop:

```
### Bitonic sort example.
int count = 1;
CALI_MARK_BEGIN("comp");
CALI_MARK_BEGIN("comp_large");
int j, k;
/* Major step */
for (k = 2; k <= NUM_VALS; k <<= 1) {
    /* Minor step */
    for (j=k>>1; j>0; j=j>>1) {
        bitonic_sort_step<<<blocks, threads>>>(dev_values, j, k);
        count++;
    }
}
CALI_MARK_END("comp_large");
CALI_MARK_END("comp");
```

Calltree Examples:

```
# Bitonic sort tree - CUDA looped kernel
1.000 main
├─ 1.000 comm
│  └─ 1.000 comm_large
│     └─ 1.000 cudaMemcpy
├─ 1.000 comp
│  └─ 1.000 comp_large
└─ 1.000 data_init


# Matrix multiplication example - MPI
1.000 main
├─ 1.000 comm
│  ├─ 1.000 MPI_Barrier
│  ├─ 1.000 comm_large
│  │  ├─ 1.000 MPI_Recv
│  │  └─ 1.000 MPI_Send
│  └─ 1.000 comm_small
│     ├─ 1.000 MPI_Recv
│     └─ 1.000 MPI_Send
├─ 1.000 comp
│  └─ 1.000 comp_large
└─ 1.000 data_init


# Mergesort - MPI
1.000 main
├─ 1.000 comm
│  ├─ 1.000 MPI_Barrier
│  └─ 1.000 comm_large
│     ├─ 1.000 MPI_Gather
│     └─ 1.000 MPI_Scatter
├─ 1.000 comp
│  └─ 1.000 comp_large
└─ 1.000 data_init
```

3b. Collect Metadata

Have the following `adiak` 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., "MergeSo
adiak::value("ProgrammingModel", programmingModel); // e.g., "MPI", "CUDA", "MPIwithCUDA"
adiak::value("Datatype", datatype); // The datatype of input elements (e.g., double, int, float)
adiak::value("SizeOfDatatype", sizeOfDatatype); // sizeof(datatype) of input elements in bytes (
adiak::value("InputSize", inputSize); // The number of elements in input dataset (1000)
adiak::value("InputType", inputType); // For sorting, this would be "Sorted", "ReverseSorted", '
adiak::value("num_procs", num_procs); // The number of processors (MPI ranks)
adiak::value("num_threads", num_threads); // The number of CUDA or OpenMP threads
adiak::value("num_blocks", num_blocks); // The number of CUDA blocks
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
```

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 above metrics for CUDA, MPI, or OpenMP programs. They will show up in the `Thicket.dataframe` when the Caliper file is read into Thicket.