CSCE 435 Group project

0. Group number:

1. Group members:

1. Connor Nicholls
2. Noah Thompson
3. Cyril John
4. Tarun Arumugam

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

2. Primary mode of communication

Discord/GroupMe

2. *due 10/25* Project topic

We have chosen the example topic, which is: "Choose 3+ parallel sorting algorithms, implement in MPI and CUDA. Examine and compare performance in detail (computation time, communication time, how much data is sent) on a variety of inputs: sorted, random, reverse, sorted with 1% perturbed, etc. Strong scaling, weak scaling, GPU performance."

2. *due 10/25* Brief project description (what algorithms will you be comparing and on what architectures)

- Enumeration Sort (MPI + CUDA)
- Odd-Even Transposition Sort (MPI + CUDA)
- Parallel Merge Sort (MPI + CUDA)
- Samplesort (MPI + CUDA) All credit for pseudocode goes to https://www.tutorialspoint.com/parallel_algorithm/parallel_algorithm_sorting.htm ## Enumeration Sort Pseudocode: procedure ENUM_SORTING (n)

begin for each process P1,j do C[j] := 0;

for each process Pi, j do

```
if (A[i] < A[j]) or A[i] = A[j] and i < j) then
   C[j] := 1;
else
   C[j] := 0;
```

for each process P1, j do A[C[j]] := A[j];

end ENUM_SORTING

Odd-Even Transposition Sort

procedure ODD-EVEN_PAR (n)

begin id := process's label

for i := 1 to n do begin

```
if i is odd and id is odd then
   compare-exchange_min(id + 1);
else
   compare-exchange_max(id - 1);

if i is even and id is even then
   compare-exchange_min(id + 1);
else
   compare-exchange_max(id - 1);
```

end for

end ODD-EVEN_PAR

Parallel Merge Sort

procedureparallelmergesort(id, n, data, newdata)

begin data = sequentialmergesort(data)

```
for dim = 1 to n
   data = parallelmerge(id, dim, data)
endfor
```

newdata = data end ### 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

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.

Merge Sort Algorithm Explanation and Questions (Connor)

The algorithms are implemented similarly across CUDA and MPI, however there are some slight differences. For MPI, the data is distributed across all the worker threads, who use a quicksort to sort their individual chunks. After that has finished, half of the workers will send their data to their neighbor, and the other half will receive that data and merge the two received lists. This repeats until we have reassembled the list we started with sorted. For CUDA, the difference is that once we send the data to blocks, the blocks cannot access the memory of each other. Because of this, we will do the same procedure as above, where we pass out data to the threads inside the blocks which then sort their sections and merge back up to one whole set. Then, we must launch the kernel again with 1 block containing the whole data set, redefine our block boundaries, and perform the same merging process with the threads in this one last block until we have a fully sorted set the same as our original. Questions for the TA: are any of these methods inefficient enough to deserve reworking? While some of this was referenced from online, I came up with most of this by myself, so I have no idea how optimal it actually is, especially since we currently don't have access to thicket to check these unless I can figure out how to set it up on Terra. Also as a footnote, as my commit message says, issues have resulted in me being unable to obtain a .cali file for the CUDA implementation. However, as a complete anecdote, after running the CUDA code so many times, it does seem to perform in similar time right now just from observation on completion time, for what that is worth.

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 s
|    |   |_ 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
|    |_ comp_large  // When you perform the computation on all of the data the process has,
|_ 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");
```

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., "Mer
adiak::value("ProgrammingModel", programmingModel); // e.g., "MPI", "CUDA", "MPIwithCUDA"
adiak::value("Datatype", datatype); // The datatype of input elements (e.g., double, int, fl
adiak::value("SizeOfDatatype", sizeOfDatatype); // sizeof(datatype) of input elements in byt
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 cod
```

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.