CSCE 435 Group project

0. Group number: 8

1. Group members:

1. Kevin Dai
2. An Duong
3. Steven Mao
4. Matthew Wang

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

```
Running different matrix multiplication algorithms on CPU and GPU (CUDA).
```

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

```
Our main method of communication will be through Discord, text messages, and in-person me

We will be running 2 different matrix multiplication algorithms using parallel computing:
and NVIDIA CUDA GPU. After implementing these algorithms, we will compare the runtimes fo
well as the algorithm's runtimes on CPU VS GPU. Finally, we'll be using the cuBLAS librar
what CUDA should theoretically achieve.
```

2b. Pseudocode for each parallel algorithm

- Algorithm 1 (Normal Matrix Multiplication)

  -------Pseudo-------

  Matrix-Multiply(A, B)

```
n = A.rows
let C be a new (n x n) matrix

for i = 1 to n
    for j = 1 to n
        C(ij) = 0
        for k = 1 to n
            C(ij) = C(ij) + a(ik) * b(kj)
return C
```

  - MPI
    - MPI_Scatter to scatter differetn rows of A to different processes and MPI_Bcast to broadcast B to all processes
    - MPI_Gather to gather the results from all processes
  - CUDA
    - Use '2D' blocks (really just long 1D) and grids to parallelize the matrix multiplication
    - Data transfer between host and device using cudaMemcpy is done in the main function

  ---

- Algorithm 2 (Normal Matrix Multiplication Recursion)

  -------Pseudo-------

  Square-Matrix-Multiply-Recursive(A, B)

```
n = A.rows
let C be a new (n x n) matrix

if n == 1
    c(11) = a(11) * b(11)
else partition A, B, and C
    C(11) = Square-Matrix-Multiply-Recursive(A(11), B(11))
            + Square-Matrix-Multiply-Recursive(A(12), B(21))

    C(12) = Square-Matrix-Multiply-Recursive(A(11), B(12))
            + Square-Matrix-Multiply-Recursive(A(12), B(22))

    C(21) = Square-Matrix-Multiply-Recursive(A(21), B(11))
            + Square-Matrix-Multiply-Recursive(A(22), B(21))

    C(22) = Square-Matrix-Multiply-Recursive(A(21), B(12))
            + Square-Matrix-Multiply-Recursive(A(22), B(22))

return C
```

  - MPI
    - MPI_Send and MPI_Recv to send and receive data between processes
  - Cuda
    - N/A TODO

- Algorithm 3 (Strassen's Algorithm)

  -------Pseudo-------

  Square-Matrix-Multiply-Recursive(A, B)

```
n = A.rows
let C be a new (n x n) matrix

if n == 1
    c(11) = a(11) * b(11)
else make S and P matrices and partition C

    S(1) = B(12) - B(22)
    S(2) = A(11) + A(12)
    S(3) = A(21) + A(22)
    S(4) = B(21) - B(11)
    S(5) = A(11) + A(22)
    S(6) = B(11) + B(22)
    S(7) = A(12) - A(22)
    S(8) = B(21) + B(22)
    S(9) = A(11) - A(21)
    S(10) = B(11) + B(12)

    P(1) = Square-Matrix-Multiply-Recursive(A(11), S(1))
    P(2) = Square-Matrix-Multiply-Recursive(S(2), B(22))
    P(3) = Square-Matrix-Multiply-Recursive(S(3), B(11))
    P(4) = Square-Matrix-Multiply-Recursive(A(22), S(4))
    P(5) = Square-Matrix-Multiply-Recursive(S(5), S(6))
    P(6) = Square-Matrix-Multiply-Recursive(S(7), S(8))
    P(7) = Square-Matrix-Multiply-Recursive(S(9), S(10))

    C(11) = P(5) + P(4) - P(2) + P(6)
    C(12) = P(1) + P(2)
    C(21) = P(3) + P(4)
    C(22) = P(5) + P(1) - P(3) - P(7)

return C
```

- MPI
    - MPI_Bcast to broadcast A, B, and matrix_size to all processes
- CUDA
    - matrix multiplication is done in the kernel function if below a certain size
    - add/subtract matrices, split a matrix, and combine matrices are done in kernel functions
    - Data transfer between host and device using cudaMemcpy is done in the strassen function

---

- Algorithm 3 (cuBLAS)
    - use CUDA to call cuBLAS library functions to perform matrix multiplication

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

```
For each measurement, we will be using a variety of nxn matrix sizes as well as number of
mulitple data points taken where the matrix size stays the same while the number of proce
different matrix sizes (strong scaling). That way we can compare the runtime for the same
of processers/threads, and also compare the runtime for different matrix sizes with the s

matrix sizes:
    128x128
    256x256
    512x512
    1024x1024
    //2048x2048 MAYBE

number of processers/threads:
    1 (sequential/linear)
    4
    16
    64
    128
```

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 sorting the splitters in Sample sort
|    |_ comp_large  // When you perform the computation on all of the data the process
has, such as sorting all local elements
|_ 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., "MergeSort", "BitonicSort")
adiak::value("ProgrammingModel", programmingModel); // e.g., "MPI", "CUDA",
"MPIwithCUDA"
adiak::value("Datatype", datatype); // The datatype of input elements (e.g., double,
int, float) -> int/float
adiak::value("SizeOfDatatype", sizeOfDatatype); // sizeof(datatype) of input elements
in bytes (e.g., 1, 2, 4) -> 4
adiak::value("InputSize", inputSize); // The number of elements in input dataset
(1000)
adiak::value("InputType", inputType); // For sorting, this would be "Sorted",
"ReverseSorted", "Random", "1%perturbed"
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) -> 8
adiak::value("implementation_source", implementation_source) // Where you got the
source code of your algorithm; choices: ("Online", "AI", "Handwritten"). -> Online/AI
```

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.