Tutorial Project: MRA Matrix-Matrix Multiplication

Instructions : Complete the tutorial using either C.

To use C, edit only the file `C/main.cpp`.

See `C/mainTemplate.cpp` for skeleton code or use to restart if needed.

1 Initialization and Finalization

In C: Set up MPI by using the lines below to Initialize and Finalize MPI in the `main(...)` method.

```
MPI_Init(&argc, &argv);
MIP_Finalize();
```

2 Implement Blocked Matrix Matrix Multiply with MPI

In C: We already implemented this method in Homework 1. The only modification that needs to be made to convert `blocked_matmat()` method to be compatible with MPI.

```
void blocked_matmat (int n, double* A, double* B, double* C, int n_iter)
```

3 Implement Fox's and Cannon's Algorithm's

In C: Use your code from Homework 2. Review your code to check for correctness and make edit if needed. Use the function signatures below.

```
void fox_matmat (double* A, double* B, double* C, int n, int sq_num_procs, int rank_row, int rank_col)
void cannon_matmat (double* A, double* B,double* C,int n, int sq_num_procs, int rank_row, int rank_col)
```

4 Implement RMA Methods

In C: Implement a RMA version of Blocked Matrix-Matrix Multiplication and either a RMA version of Fox's Algorithm or a RMA version Cannon's Algorithms. Use the function signatures below.

```
void rma_blocked (int n, double* A, double* B, double* C, int n_iter)
void rma_fox (double* A, double* B, double* C, int n, int sq_num_procs, int rank_row, int rank_col)
void rma_cannon(double* A, double* B, double* C,int n, int sq_num_procs, int rank_row, int rank_col)
```

See Hints for some RMA tips.

5 Compile code

Compile your code with:

```
sh compile.sh
```

The script should compile and run all the code. with the following lines as tests.

```
//Smaller Tests
srun -n 1 ./test 2048
srun -n 4 ./test 2048
srun -n 16 ./test 2048
//Larger Tests
srun -n 4 ./test 4096
srun -n 16 ./test 4096
```

6 Check for Correctness

Make sure the code runs and gives an output, This is the output from my code for a small test.

```
Running: srun -n 1 ./test 2048
srun: job 134909 queued and waiting for resources
srun: job 134909 has been allocated resources
To request GPUs, add --gpus-per-node X or --gpus X, where X is the desired
    number of GPUs.
Job 134909 running on easley022
MPI Blocked      2.418642 sec
MPI Fox          45.085509 sec
MPI Cannon       36.801065 sec
RMA Blocked      2.446727 sec
RMA Cannon       2.772050 sec

Done
```

Explain the Runtime.

7 Some Hints:

Helpful Slides
Lecture Slides from U-Illinois.
Another Slide Deck from UI.
See "Class 20: One-Sided MPI" lecture slide on Canvas

Useful API's
APIs I used in my RMA implementations

```
int MPI_Win_create (void *base, MPI_Aint size, int disp_unit, MPI_Info
   info, MPI_Comm comm, MPI_Win *win)
```

Lock and Unlock
Use Lock and Unlock to use One-Sided Asynchronous Communication. Allows for Root process (R) to make a call to another process (P) without that process (P) sending back info, behaves like shared- memory models.

```
MPI_Win_lock(int locktype, int rank, int assert, MPI_Win win)
MPI_Win_unlock(int rank, MPI_Win win)
```

Get Put and Free
You only need to use one of the models, Get moves the data early. Put moves the data late. I used get in my implementation.
Get Moves the data: Target → Root.
Put Moves opposite: Root → Target.

Cont.

```
MPI_Get(void *origin_addr, int origin_count, MPI_Datatype origin_dtype,
    int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype
    target_dtype, MPI_Win win)
MPI_Put(void *origin_addr, int origin_count, MPI_Datatype origin_dtype,
    int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype
    target_dtype, MPI_Win win)
```

Use Free to unallocate the window object, allocation is done in 'create' so don't worry about allocation.

```
MPI_Win_free (MPI_Win win)
```

Other Tips

Only consider a square number of processes [2,4,8..], to make the MPI and RMA method better for testing. This allows for perfect partitioning for NxN matrices where N is a perfect square.

Use helper methods since the methods for Matrix Matrix Multiplication have similar implementations. I used `local_accum` and `coords_to_rank` as helpers.

Use `#inlcude <algorithm>` and `#include <vector>` header to use vectors and copy data. Helpful with data flow between buffers. See links below.
#include algorithm
#include vector

The End.