Assigned:	Sat, Oct 12, 2019 12:01 AM
Due:	Fri, Oct 25 2019 11:59 PM
Last Day to Handin:	Mon, Oct 28 2019 11:59 PM

1 Overview

Before you begin, please take the time to review the course policy on academic integrity:

Academic Integrity Policy

Download the Assignment 4 starter code from the course Github using:

```
$ git clone https://github.com/cmu15418/assignment4-nbody
```

1.1 Assignment Objectives

In the previous assignment, you implemented a parallel *n-body simulation* using shared-memory parallelism with the support of the OpenMP library. In this assignment, you will again implement a parallel n-body simulation using an alternative parallel programming model—message passing. As you already know from the lectures, unlike shared-memory parallelism where threads communicate by reading and writing shared memory, in the message passing model the processes do not have access to shared memory, and they can communicate only by sending messages to each other.

There are many languages that include message passing as their central feature, such as Erlang and Go. In this assignment, however, we will use Message Passing Interface (MPI), a portable message passing standard that defines syntax and semantics of a set of routines for parallel programming. There are multiple high-performance implementations of MPI, of which we will use Open MPI. Despite the similarity in their names, Open MPI and OpenMP are different libraries with different objectives. It is typical to combine OpenMP and OpenMPI within a single application for intra- and inter-machine parallelism, respectively.

You will be given a sequential implementation of the n-body simulation that you will need to parallelize using MPI primitives. The assignment will help you to understand the advantages and the limitations of the two approaches to parallelism—shared-memory and message passing—and will give you an idea where one approach is preferable to the other.

1.2 Machines

You will start by using GHC machines for this assignment. Please prepend `/usr/local/depot/openmpi/bin` to your `PATH` environment variable to access MPI tools. You will use `mpirun` command to run your MPI binary, which launch multiple instances of your program on the same host. For example, the following command launches 4 instances of the `echo` tool, with each process tied to a different core:

```
$ mpirun -n 4 echo hi
hi
hi
hi
hi
```

These processes do not share address space and can communicate with each other using message passing over the loopback network interface. Unlike `echo` tool, your program will use MPI calls to communicate with its instances.

Although MPI is usually used on a large cluster for massive computations, it is typical to develop an MPI application on a multicore machine and observe speedup there before trying it on a cluster. Early next week, we will post updates on Piazza with instructions on running on the `latedays` cluster.

1.3 Resources

If you do not have an experience with MPI, we strongly recommend going through the MPI tutorial at https://computing.llnl.gov/tutorials/mpi/ before you attempt doing the assignment. Learn how to compile and run MPI binaries on the same host and across the nodes of a cluster. Study, compile, and run at least a few of the exercises (Hello World, Array Decomposition, etc.) listed at https://computing.llnl.gov/tutorials/mpi/exercise.html.

2 Starter Code

We provide you with a sequential implementation of the nbody simulation. It is based on the code of the previous assignment, however, the code was rearranged to simplify your MPI implementation. More specifically, you will need to make changes to the file `src/mpi-simulator.cpp` to complete the normal portion of this assignment. If you choose to do the extra credit problem, you may need to modify `src/quad-tree.*` as well. You should not need to modify any other file.

Compile the provided code by typing `make`. You can determine the performance of the resulting binary, `nbody-release`, by running it using `./checker.pl`. You will notice that some lines are being printed multiple times. This is because the checker script runs the binary with `mpirun -n 8`—you should take some time to study the checker script—and each instance prints out details about its runtime. In your final MPI implementation only a single instance should produce such output.

Initially, you should observe similar times for the reference sequential implementation and the two MPI implementations, one of which is expected to avoid work imbalance. Once you complete the tasks described below, you should observe speedup with the MPI implementations.

3 Your Tasks

Since you are already familiar with the problem definition, we jump straight into the tasks for this assignment. Each item in the following list is a task that you need to complete, by answering a question in the write-up and/or writing code.

Task 1 (20 points) You need to provide an answer to this question in your write up. Although you are not expected to provide code, you may need to attempt an implementation to get to the answer.

One of the tasks in the previous assignment was to parallelize building a quad tree. Think about how you would parallelize the recursive `buildQuadTree` function using MPI. You will find that it is hard to do, but can you pinpoint what makes it hard compared to OpenMP? It may help you to know that the difficulty is not specific to `buildQuadTree` function, but to recursive functions in general, which is why recursive algorithm implementations in MPI are rare.

Task 2 (10 points) Even if you somehow manage to implement a recursive `buildQuadTree` function using MPI, you will find it not as efficient as the OpenMP implementation, especially given that the tree building is not compute-intensive. Can you guess why an MPI implementation will not be as efficient as an OpenMP implementation? Please provide your answer in your write-up.

Extra Credit (30 points) Thinking about the above questions will help you realize that parallelizing `buildQuadTree` using MPI is not a good idea—at least, not in its current form. It can, however, be parallelized more easily if converted to an iterative form. Provide a parallel implementation of `buildQuadTree` using MPI by modifying `src/quad-tree.*`. You are free to redesign the algorithm however you see fit. Briefly describe your algorithm in your write-up.

Since MPI has communication overhead and building the quad tree is not very compute-intensive, we are not going to judge your solution on speedup, but on whether you took a reasonable approach to parallelization.

Task 3 (50 points) In this task, you will parallelize the simulation step, which is a straightforward application of MPI primitives. If you choose to do the extra credit task above, please use your parallelized `buildQuadTree` implementation for this task. You should work in `src/mpi-simulator.cpp` for this task.

Task 4 (20 points) In the previous assignment you used dynamic scheduling in OpenMP to deal with the work imbalance in `corner` images. In MPI, you cannot avoid work imbalance with a single keyword as you did in OpenMP, however, there are other solutions. Think about how you can avoid work imbalance in your MPI implementation and briefly describe your strategy in your write-up and provide the corresponding implementation in `src/mpi-simulator.cpp`.

The checker script runs your binary with `-mpilb` option to invoke the load-balancing version of your code. Using this option sets the `options.simulatorType` to `SimulatorType::MPLIB`. In your code, you can condition on this to branch into the load-balancing version of your code.

4 Evaluation

You will be evaluated both on correctness and speed; an implementation that is not correct will not receive any points, so ensure that correctness does not fail. **After making your MPI changes to the code, make sure to adjust the timing code so that it includes the communication primitives. We will read your code to ensure you are timing correctly.** Your solution to task 3 should achieve 3-5× speedup over the reference sequential implementation across scenes. For `corner-50000` scene, your solution to task 4 should reliably achieve 20-40% higher speedup than your solution to task 3, over the reference sequential implementation.

For example, Listing 1 is a sample output from the reference implementation where `MPI-LB` shows the load balancing MPI solution. It is possible that the overhead of your load-balancing solution leads to slowdown for cases where there is no load imbalance, as you see here with `repeat-10000` scene. This will not result in point deduction.

Listing 1: Sample Reference solution results

```
---------------------------------------------------------
| Scene Name     | MPI Speedup    | MPI-LB Speedup |
---------------------------------------------------------
| random-10000   | 4.523137       | 4.415653       |
| random-50000   | 4.193441       | 4.186013       |
| corner-10000   | 3.296542       | 4.354282       |
| corner-50000   | 3.620516       | 4.325133       |
| repeat-10000   | 4.905386       | 4.160051       |
```

5 Hand-in Instructions

You will submit your code via Autolab and your report via Gradescope. For the code, you will be submitting all C++ header and source files in the src folder.

1. Submitting your code:
   (a) If you are working with a partner, form a group on Autolab. Do this before submitting your assignment. One submission per group is sufficient.
   (b) Make sure all of your code is compilable and runnable. We should be able to simply run make, then execute `./checker.pl`. Please remove excessive print statements, if they were added.
   (c) Run the command "make handin.tar." This will run "make clean" and then create an archive of any C++ source code in `src`.
   (d) Submit the file handin.tar to Autolab.
2. Submitting your writeup:
   (a) Please upload your report as file report.pdf to Gradescope, one submission per team, and select the appropriate pages for each part of the assignment. After submitting, you will be able to add your teammate using the add group members button on the top right of your submission.