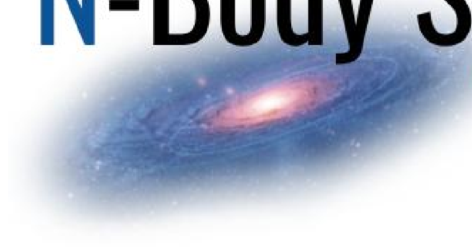
N-Body Simulation



Carnegie Mellon University

Assigned:	Wed, Mar. 1, 2023	
Due:	Wed, Mar. 22, 2023	11:59 PM
Last Day to Handin:	Sat, Mar. 25, 2023	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/cmu15418s23/asst4.git
```

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.

1.2 Machines

Feel free to use the machines in the Gates cluster for your initial development and testing. However, you need to collect your final performance numbers on the PSC Bridges-2 RM (Regular Memory) machines. Please note that the computation resource on the PSC machine is limited, so we strongly advise against doing developing or debugging on it, and you should never do things like a grid search on PSC to find a set of best parameters.

For more information on these machines, please see the `tutorials/machines.pdf` document. You may also find it helpful to refer to the Bridges-2 User Guide. To use OpenMPI on the PSC machines, you must first run module load mpi to load the OpenMPI module. To run a program with MPI, use the `mpirun` command. An example invocation of this command is as follows:

```
mpirun -np <numprocs> nbody-release-v1 -n 50000 -i 5 -in src/benchmark-files/random-50000-init.txt -s 500.0
-o logs/random-50000.txt > logs/random-50000.log
```

The handout also provides an example MPI program `sqrt3` that approximates sqrt(3).

1.3 MPI Resources

If you do not have an experience with MPI, we strongly recommend going through the MPI tutorial at https://hpc-tutorials.llnl.gov/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://hpc-tutorials.llnl.gov/mpi/exercise$_1$/.

2 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.

There are two source files that you need to modify: `src/mpi-simulator-v1.cpp` and `src/mpi-simulator-v2.cpp`. They are used for different sub-tasks, and they will compile to two different binaries: `nbody-[v1/v2]-[release/debug]`.

The starter code is greatly simplified compared to Assignment 3. We removed most debugging and visualizing utilities. If you are really interested, or you think you need them for debugging you can copy them back from Assignment 3.

2.1 Task 1: (15 Points) Naïve MPI Implementation

Please implement a naïve MPI version in `src/mpi-simulator-v1.cpp` (you will also need to copy your sequential quad-tree building from Assignment 3 to `quad-tree.h`). An iteration in this approach can be briefed as:

- Each node is assigned a subrange of particles, but also receives the **full** and **latest** particle list.
- Each node builds the quad-tree independently. Note that the trees built by all nodes should be identical and should contain all particles.
- Each node performs a one-step simulation on its subrange of particles with its tree.

For Task 1 through Task 3, 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. Your program should measure and print the total simulation time, which can basically be defined as the time of the loop of the simulation iterations. We will only use the total simulation time as the performance metric, but you will also need to measure specific parts in the program for the write-up and analysis section. For general time measurements in MPI, you may find `MPI_Barrier` useful.

We provide a new checker script `checker.py` to run your program. Hopefully Python will be more readable than Perl, so you can take a look at its content yourself. It takes two command line arguments: the first picks the program version to run, and the second determines whether to use load balancing (see Task 3). So for this task, you should run `./checker.py v1 0 -ghc/-psc`. It will run your program on 16 and 128 workers, collect the total simulation times, and compare them with our reference solution to calculate the performance score. Each data point contribute evenly to the score.

2.2 Task 2: (25 Points) Reduce Communication

After Task 1, you may see that the naïve MPI version doesn't scale so well, especially for the `sparse` test cases (which can also be seen from reference solution performance). You should implement a slightly more complicated algorithm in `src/mpi-simulator-v2.cpp`:

Table 1: Reference solution performance for Task 1

Scene	16 workers	128 workers
random-50000	0.588422	0.161911
corner-50000	1.148824	0.241397
repeat-10000	0.183667	0.136549
sparse-50000	0.354411	0.784285
sparse-200000	1.582532	3.821544

- Divide the space into grids. Each node is assigned the particles within a grid. It is okay that you divide the grid evenly in space (not considering particle numbers), and that your implementation only supports the case where the number of workers is perfect square number.
- Two nodes communicate the grid boundaries of each other to decide whether the two sets of particles can possibly be within the predefined distance threshold of considering particle interactions. If so, they need to do further communication to really send the particles.
- Each node still builds the quad-tree independently, but the trees will no longer be identical or contain all particles.
- Each node perform an one-step simulation on its particles with its tree.

Note that you may not want to rebuild the grid for every iteration, or to use the initial division throughout the iterations (think about why). Instead, the grid should be updated *regularly*.

Also note that the problem definition remains the same as Assignment 3: we don't add any more approximation. This means that we still need to exactly consider all the particle pairs within the predefined distance threshold. That being said, the computation order may not remain the same. We know that floating point operations are generally not commutative or associative, so a different order may produce small numerical errors. We also know that the N-body problem is numerical unstable, which means that these small numerical errors may be accumulated and amplified. To address this, we increased the tolerable error bounds compared to Assignment 3. This may be enough to let most correct implementations to pass, but we can't prove it theoretically. If you are very confident that your implementation is correct, but `checker.py` thinks you are wrong, please don't hesitate to ask us.

Run `./checker.py v2 0 -ghc/-psc` to evaluate this task. Our reference performance (on PSC machines) is:

Table 2: Reference solution performance for Task 2

Scene	16 workers	121 workers
random-50000	0.819216	0.176659
corner-50000	3.568420	0.754285
repeat-10000	0.297109	0.084589
sparse-50000	0.335568	0.100940
sparse-200000	1.532440	0.417115

You are welcome to completely ignore the instructions in this task and design your own algorithm. If your algorithm can match or exceed our reference solution, we will give you up to 10 points of extra credit.

(Hint: using asynchronous communication can make programming easier and may improve performance.)

2.3 Task 3: (5 Points + 15 Points Extra Credit) Load Balancing

Required Part (5 Points): In Assignment 3 you could use different scheduling strategies 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 Task 1 implementation, and describe how you might implement at least one reasonable strategy in a message passing framework such as MPI.

Extra Credit (15 Points): Implement your load balancing strategy in your `src/mpi-simulator-v1.cpp` program. Your program should be able to enable and disable load balancing, so it shouldn't affect the result of your Task 1. Note that you can run `./checker.py v1 1 -ghc/-psc` to help evaluate this task, but we we don't grade strictly based upon the script output. For the `corner-50000` scene, you should expect your solution to achieve 20-40% higher speedup over your solution to Task 1. It is possible that the overhead of your load-balancing solution leads to slowdown for cases where there is no load imbalance. As long as your implementation shows *reasonable* speedup over Task 1, you are likely to receive the bulk of these extra credit points.

2.4 Analysis and Write-up (55 Points)

While we do have some performance requirements, we care a lot more about your thought process in parallelizing the algorithm and your analysis of the results.

In your writeup, please include the following:

1. A detailed discussion of the design and rationale behind your approach to parallelizing the algorithm. Specifically try to address the following questions:
   - What approaches have you taken to parallelize the algorithm? What efforts did you make to profile your code and identify bottlenecks, and how did this data affect your designs?
   - Did you observe any load imbalance? Discuss the reason and prove it with concrete performance data.
   - Describe your strategy for load balancing in Task 3. Can it be easily adapted to your Task 2 implementation? Briefly state your reason. Please note that implementing it is an extra credit task, but proposing a strategy is mandatory.
   - What information is communicated between MPI processes, and how does this affect performance? How does communication scale with the number of processes? You may want to draw some plots to demonstrate the scaling.
   - At high process counts, do you observe a drop-off in performance? If so, why do you think this might be the case?
   - Why do you think your code is unable to achieve perfect speedup? (Is it workload imbalance?ccommunication? synchronization? data movement? etc?)
2. A plot of the speedup of the total simulation time vs. the number of processors sampled at 1, 4, 16, 64, and 128 processors. Speedup is calculated as $\frac{T_1}{T_P}$, where T_1 is the time for one processor, and T_P is the time for P processors. There should be three plots in total, each corresponding to a task. Each plot should include all the test cases. Discuss the results that you expected for all the plots and explain the reasons for any non-ideal behavior that you observe.
3. 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.
4. 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?
5. In Assignment 3 and 4 we disallow you to modify or delete existing fields in our class definitions and related functions. Suppose it is allowed, can you think about some possible optimization for the MPI version? (Hint:

think about how you can modify the `Particle` struct to reduce communication.) Bonus (5 extra credit points): do what you describe and compare the performance; the modified version should be your final submission.

6. Include the final output of `./checker.py v1 0 -psc` and `./checker.py v2 0 -psc` in your write-up. If you do the bonus question to implement load balancing, also include the output of `./checker.py v1 1 -psc` and/or `./checker.py v2 1 -psc`.

3 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.py`. 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`. If you find it absolutely necessary, you may modify the `Makefile` to include other files you have added.
   (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.