Lab 4: Parallel Programming

CS 33, UCLA

Spring 2021

1 Introduction

The n-body problem is a common computational scenario is physics and related fields, such as astronomy. One needs to predict the motions of objects as they gravitationally interact over a period of time. As with many real-world computational problems, the n-body problem is evaluated through simulation over a large number of infinitesimally small timestamps, in effect performing mathematical integration to approximate a solution.

An example of an n-body problem would be the motions of planets orbiting a star, which through the nature of gravity, every object exerts a force on every other object in the system, no matter how large or small. The issue with simulating such a problem is that each time step features a high computational complexity, since for *every object* we need to compute the gravitational effect of *every single other object* on it during *every single time step*. Effectively, this gives us a computational complexity of $O(n^2 * t)$, where n is the number of objects and t is the number of time steps.

Over the years, much effort has been spent optimizing these types of problems, which feature a high computational complexity, but also a problem structure that is ideal for being made parallel. The n-body problem in particular is ideal for this because the input to calculations made during one time step is the output of the previous time step, with intermediate values generated during the current time not being used in different objects. More specifically, each time step of the n-body simulation is usually performed in two separate "sub-steps": (1) computing the net displacement of every object based upon the net force from the distances and masses of the other objects in the system and (2) actually moving the objects. Since the net displacement is computed based upon the output of the *previous* phase, the computations for each object are independent during a given time step.

For more information about the structure and use of n-body simulations, please refer to `https://en.wikipedia.org/wiki/N-body_simulation`.

2 Assignment

We have written out a basic n-body simulation in C++, although we effectively only use C for the portions of the code that you will need to modify. However, our solution is poorly optimized and does not feature any parallelism. Your goal for this assignment is to reduce the program's runtime, **while maintaining the accuracy of the results**.

To accomplish this, you will need to make optimizations in a variety of locations throughout the assignment to decrease your programs runtime using the knowledge of computer organization you have gained this quarter. In particular, you will need to parallelize your code using the OpenMP library, as was covered in lecture. While this will be the major factor in reducing runtime, there are still other locations in the starter code that can be optimized in various ways for further improvements.

Note that you will need to use various features to match the runtime of our solution, including OpenMP parallelism and barriers.

3 Grading

As discussed above, grading will be based upon your code's average performance over a number of repeated trials as compared to our solution's runtime. Our solution is optimized, but *it is not as optimal as possible*. Your code will be graded on its (**correct**) runtime compared to our solution during our grading runs. Our solution's runtime will be treated as a 100% on this assignment.

However, as we mentioned previously, our solution is not 100% optimal. Therefore, if your code outperforms ours during our grading test runs, you can expect **extra credit above 100%**.

Note that in order to receive **ANY** credit for a test case, your submission's standard output (`std::cout`) must **EXACTLY** match the standard output of our solution. Feel free to output any debugging information to standard error (`std::cerr`), but keep in mind that writing any output takes up precious CPU cycles!

4 Building and Testing

NOTE: Please ONLY evaluate your code on the `cs33.seas.ucla.edu` machines, as these are the machines we will grade your code on. We cannot provide assistance with issues caused by developing and testing your code on any other machines.

We have provided a "Makefile" in order to simplify building your code. You can therefore build your code by running:

```
make
```

Then, to test your code against our test cases, run:

```
make test
```

This command will evaluate both your program's correctness compared to the test case, as well as its performance (i.e., runtime) compared to both the starter (unoptimized) implementation and our fairly optimized solution. Your goal is to reduce your program's runtime as much as possible compared to the starter implementation.

When you have finished optimizing your code and are ready to submit, please fill out the `README.md` file with the relevant information. Then, edit `Makefile` to change the "UID" variable to your UID (without any dashes, spaces, or other separating characters). Finally, run the following command to package your code:

```
make package
```

This will output a file named like `UIDUIDUID.tar.gz` (with "UIDUIDUID" replaced with your UID). Please submit this file via CCLE. Please do not package your project for submission using any other method. Note that this command will only include your `Makefile`, `README.md`, and any `*.cpp` and `*.hpp` files in the root directory of your project folder, so please make sure any additional source and header files that you may add are in the root directory of the project.

As always, please extract and test your generated archive file to a new directory before submission to make sure that it is able to be built and run from only the contents of the archive.

4.1 Interpreting Test Results

Our test package will evaluate your code for both runtime and correctness. The correctness with be indicated by a test case "passing". Below this, we will list the runtime of your program, comparing it against two implementations: the base, unoptimized implementation ("base") and our optimized solution.

Please note that the produced runtimes may vary greatly due to changing load on the evaluation systems. Please run the tests a few times and compute an average to obtain a valid picture of your implementation's comparative runtime.

5 Important Notes

- Please follow the OpenMP tutorial in the lecture slides. If you are still unclear about how to use OpenMP, please use CampusWire or go to office hours.
- To optimize your code, you **ARE** allowed to change anything in the source files in the root directory of the project, as well as the Makefile. Please do not edit the contents of the `tests` directory. Also please make sure your Makefile still generates an executable in the same directory named `nbody`.
- You are **ONLY** allowed to use the C and C++ standard libraries and OpenMP. Using any other library is prohibited.
- Try to think outside of the box when optimizing your code, particularly if you want to achieve extra credit on this assignment. This project is meant to test your knowledge of large portions of the material we covered this quarter, not just OpenMP.
- Please **ONLY** perform your testing and runtime comparisons on the `cs33.seas.ucla.edu` machines.
- Please **DO NOT** set the number of OpenMP threads in your program. Doing so may break evaluations and cause your code's runtime to be inaccurate, potentially negatively impacting your grade.
- If you are getting strange build issues, please ensure that your `$PATH` contains `/usr/local/cs/bin`. You can check this with:

  ```
  echo $PATH
  ```

  If the output of the above command does not contain this directory, please run this command every time you open a new connection to the servers:

  ```
  export PATH=/usr/local/cs/bin:$PATH
  ```

  This will ensure you are using a modern C++ compiler.