Contributors: Yujia Lin, Dong Liang

Introduction

This is the project of 422 fall 2017. The problem of this project is to simulate n-Bodies and collisions in a universe.

In the report, the program used to simulate the problem will be discussed. Also, the report will give the solution of how to test the program (how do we know the output is true). Tables will be used to record the result of running time.

We used "Oxford" to test and run our program.

There is a bash script used to run for timing tests. And the timing tests are focused on different number of workers and different number of bodies.

Analysis and conclusions will be in the "Conclusion" section.

Programs

The entire program is written in Java. And there are there packages: controller, model, view. A text view will be generated as running the program. One of the arguments was designed as an option of GUI. So, if there is no arguments or the arguments is a string "false", the GUI will not show up.

There are basically two versions of the program. One is sequential and the other is a parallel solution. Both versions are running together in one main method to present the different performances as the other situations keep the same (same number of bodies, radius of a body, number of steps).

For the parallel solution, we used barrier to synchronize workers.

There are basically four command arguments for both sequential and parallel versions. They are:

Number of workers (2 to 32), Number of bodies, Size of each body(radius), Number of time steps.

Additional arguments in command line includes: GUI option, value of seed and debug option.

Note: The number of workers will be ignored by the sequential version.

Verification

So, how do we know that our program is working correctly? We break this problem into 2 parts: 1. Does the entire program working fine? (Can deal with the arguments and having the input and output correctly)

2. Are those n-bodies generated by our program moving and colliding correctly?

For the 1st part: we run a list of commands to test it. The lists of commands are recorded in “README.txt”.

For the 2nd part: we use smaller number of bodies (like 2 or 10) and run the program with GUI. Then, we look and make sure the moving and collisions of the bodies are correct.

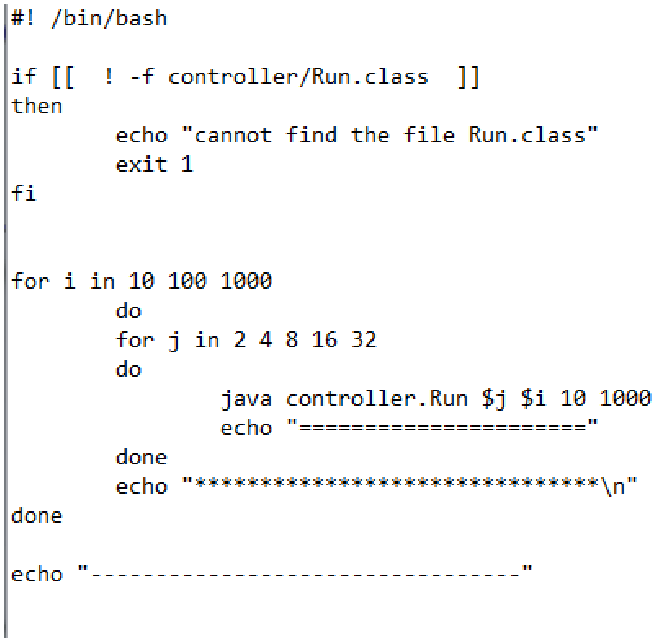
The more accurate way to test this is using the Junit test. For example, we could write some test like this: To test the moving is working correctly we could create 2 or more points with their positions. Then calculate values of their positions after n seconds by using the formula. Then run the program and get those points’ position values by using getters. Then compare the position values with the ones calculated. If they are the same, it means we had them working correctly and the green bar will show up.

The unit test would be more accurate, but we just didn’t have the time to do it.

Timing Experiments

For this section, a bash script was used by us to run the commands. We keep track of the computation time and the number of collisions as the number of workers and number of bodies changed and the other arguments keep the same.

This is the bash script we used to run those commands:



```bash
#! /bin/bash

if [[  ! -f controller/Run.class  ]]
then
        echo "cannot find the file Run.class"
        exit 1
fi


for i in 10 100 1000
        do
        for j in 2 4 8 16 32
        do
                java controller.Run $j $i 10 1000
                echo "======================"
        done
        echo "*******************************\n"
done

echo "---------------------------------"
```

Data and charts

Time section

We keep track of the computation time in this section.

Table 1: Sequential version unit: seconds

number of body\number of workers	2	4	8	16	32
10	0	0	0	0	0
100	0	0	0	0	0
1000	22	21	21	21	21

Table 2: Parallel version unit: seconds

number of body\number of workers	2	4	8	16	32
10	0	0	0	0	1
100	0	0	0	0	1
1000	11	14	10	10	10

Collision section

We keep track of the number of collisions detected in this section.

Table 3: Sequential version

number of body\number of workers	2	4	8	16	32
10	3000	3000	3000	3000	3000
100	139982	139982	139982	139982	139982
1000	14507001	14507001	14507001	14507001	14507001

Table 4: Parallel version

number of body\number of workers	2	4	8	16	32
10	3000	3000	2003	3000	3000
100	140076	142358	143863	142886	147437
1000	13850309	13258186	12030268	12140004	12281254

Analysis

For the time, based on table 1 and 2, the parallel version is faster than the sequential version as the number of body increased until a large value (1000 in this case).

The more workers we used, the faster speed we get. (table 2 last row)

For the collisions: Based on table 3 and 4, we observe that more number of bodies means more collisions.

Based on table 4, more workers used, fewer collisions we get.

Time test for large number of bodies (2000)

Bash script used for this timing test:

```
#! /bin/bash

if [[  ! -f controller/Run.class  ]]
then
        echo "cannot find the file Run.class"
        exit 1
fi



        for j in 2 4 8 16 32
        do
                java controller.Run $j 2000 10 1000
                echo "=====================\n"
        done
        echo "*******************************\n"


echo "---------------------------------"
```

Data

Sequential Unit: seconds

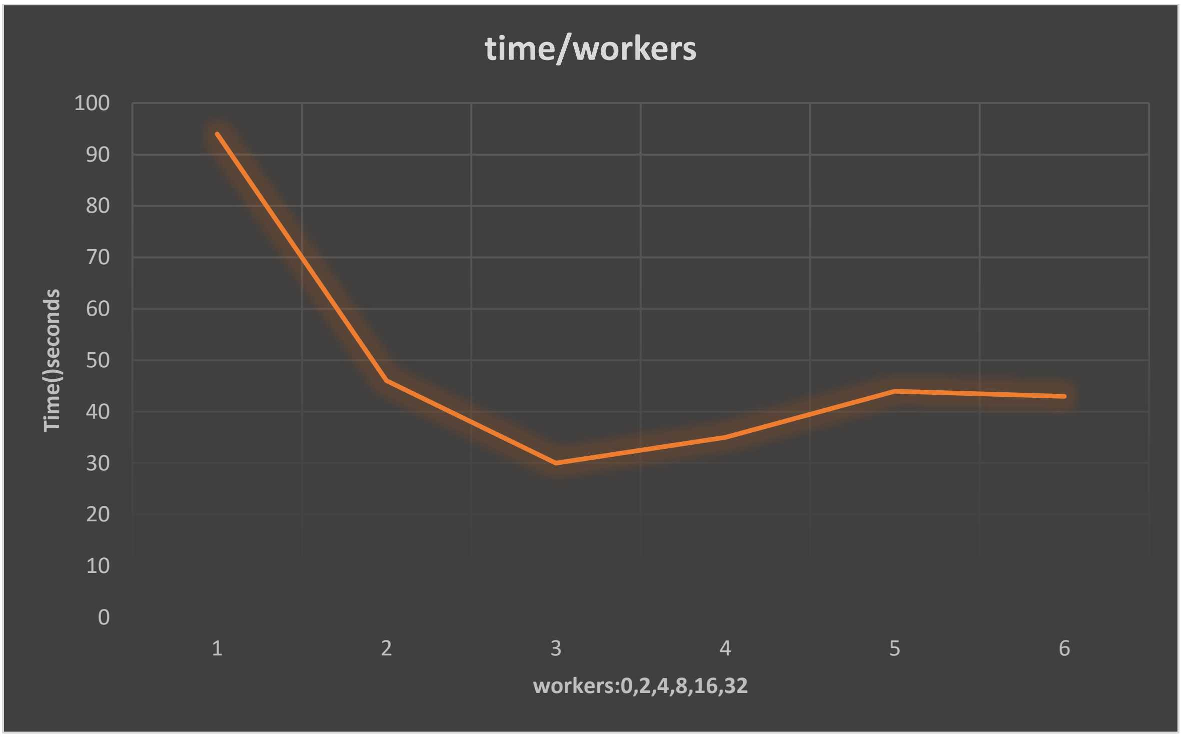
number of body\number of workers	Average	2	4	8	16	32
2000	94	93	94	95	94	94

Parallel Unit: seconds

number of body\number of workers	0	2	4	8	16	32
2000	94	46	30	35	44	43

Note: since the sequential version will ignore the number of workers, so we use the average number of upper table as the values of case that number of worker is 0.

Chart of time VS number of workers



Analysis:

Speed does increase as the use of workers increases, but 16 and 32 workers are not more helpful than 4 workers. Probably we need more running here. Unfortunately, there is no more time to do so.

Other Experiments

A very interesting feature of these bodies we observed is, when we ran the program with GUI, some of those particles may end up with sticking together after a while.

At first, we thought this is a bug of our code (especially the use of formula), then we realize this is possible to happen. In fact, the particles were not technically “sticking together”. Instead, they were still moving in a small range. After we increased the radius of the bodies, we observed this.

Conclusion

This report shows what is the problem and program, how do we test the program, the data collection and analysis of time and collisions.

The most important thing that we have learnt from this project is how greatly the parallel programming would help the performance and how large a real-world-related project would be like.

Moreover, the test of the correctness of moving and collisions gave us a headache. This might be better if we planned to write a unit test at the beginning of this project.