UNIVERSITY OF GRONINGEN

PARALLEL COMPUTING

Homework 4: Parallel simulation of dynamical systems

Author:
Konstantinos MAZGALTZIDIS
(s5100453)

May 23, 2022

1 Introduction to N-Body

The N-body problem refers to finding out the subsequent motion of multiple bodies with known initial positions, velocities and masses.

2 Algorithm parallelization

Inspecting the provided code `nBody.c` we observe that a lot of time is spent on the function `startSimulation`, and we are going to focus our attention in that function, the other functions are used to read input and write output, so we can't parallelize that kind of work.

Without getting into much detail about the math behind the N-Body simulation:

```
1  void startSimulation(Particle *particles, int n, int timeSteps, int numThreads) {
2      // Accelerations
3      vec3 *acc = malloc(n * sizeof(vec3));
4      for (int t = 0; t < timeSteps; t++) {
5          memset(acc, 0, n * sizeof(vec3));
6          for (int i = 0; i < n; i++) { // Compute all-to-all forces and accelerations
7              for (int j = 0; j < n; j++) {
8                  if (i == j) {continue;} // Skip interaction with self
9                  float rx = particles[j].pos.x - particles[i].pos.x;
10                 float ry = particles[j].pos.y - particles[i].pos.y;
11                 float rz = particles[j].pos.z - particles[i].pos.z;
12                 float dd = rx * rx + ry * ry + rz * rz + EPS;
13                 float d = 1 / sqrtf(dd * dd * dd);
14                 float s = particles[j].mass * d;
15                 acc[i].x += rx * s;     acc[i].y += ry * s;     acc[i].z += rz * s;
16             }
17         }
18         for (int i = 0; i < n; i++) { // Update positions and velocities
19             particles[i].pos.x += particles[i].v.x;     particles[i].pos.y += particles[i].v.y;
20             particles[i].pos.z += particles[i].v.z;     particles[i].v.x += acc[i].x;
21             particles[i].v.y += acc[i].y;       particles[i].v.z += acc[i].z;
22         }
23     }
24     free(acc);
```

We can highlith some key observations regarding the parallelization of this code:

- **Partitioning:** We can easily split the `particles` array into (N/P) parts. (N: size of array `particles`, P: number of threads).

  In more detail we see there is a lot of independence in the second loop (line 6 to 17), that part only writes to the i position of the `acc` v3 array (line 15).

  With that in mind we can parallelize by taking the loop code (6-17) and move it to a threaded function.

- **Communication:** There is no need for threads to communicate to each other, because each thread will only write data to each own part of `acc` and `particles`.
- **Synchronization:** This is important because we observe in the last loop (18:22), the `particles` array is altered, but the array can be accessed from another thread that hasn't finished yet the previous loop (9:15), and cause the calculations to be wrong.

  We could use a synchronization barrier[1] but because when testing it didn't make the program faster, I decided to not use it, and just put the "Update positions and velocities" part after all the threads have finished.

3 Execution time

3.1 Methodology

I run the code in Peregrine multiple times to count the execution time. But because at each run the time is slightly different I chose to run the same test two times and get the average time. Also, because the cluster is used by other users, they might be a little noise to the results.

To run the same test multiple time, I used the scripts:

```
# ./a.out is the original serial code
for cmd in 1 1; do ./a.out 10000 $cmd performanceTest1.in > /dev/null; done
for cmd in 1 1; do ./a.out 3 $cmd performanceTest2.in > /dev/null; done
# ./a.out is my threaded code
for cmd in 1 1 2 2 4 4 8 8 16 16; do ./a.out 10000 $cmd performanceTest1.in > /dev/null; done
for cmd in 1 1 2 2 4 4 8 8 16 16; do ./a.out 3 $cmd performanceTest2.in > /dev/null; done
```

3.2 Table

Input file	Timesteps	Metric	Serial	1	2	4	8	16
performanceTest1.in	10 000	Execution time	81.06	108.97	55.86	30.35	14.30	9.48
		Speed-up	1.34	1.00	1.95	3.59	7.62	11.49
		Efficiency	134%	100%	98%	90%	95%	72%
performanceTest2.in	3	Execution time	167.48	159.34	82.21	42.63	21.19	10.61
		Speed-up	0.95	1.00	1.94	3.74	7.52	15.02
		Efficiency	95%	100%	97%	93%	94%	94%

I consider the difference between the serial and the 1 threaded version to be load differences between each run (noise).

The raw data I used for the calculations can be found in the spreadsheet : https://docs.google.com/spreadsheets/d/1nVi68oISUVzHmLsSRiDWntMQOLV8m2oOW2mk0dqJu4o

[1] Manual:https://docs.oracle.com/cd/E19253-01/816-5137/gfwek/index.html