Homework 3 Report, CS 5220, 2014 Spring

Lingnan Liu, Sheng Wang, Hang Chu, Jilong Wu, Mohammed sameed Shafi

March 28, 2014

1 $O(n)$ time

We tested our code with different h. The number of particles can be calculated from h as

$$n_{particle} = (\frac{1.3}{h})^3$$

The run time is shown in Table 1. We use 100 steps in each frame.

Table 1: Timing results (4000 time steps) with different mesh sizes and different processor numbers.

h	0.5	0.4	0.2	0.1	0.05	0.04	0.02
$n_{particle}$	8	27	216	2197	17576	32768	274625
$secperframe(serial)$	0.00042	0.00076	0.0097	0.081	0.43	1.1	36
$secperframe(parallel)$	0.0011	0.0016	0.010	0.036	0.21	0.55	27

The run time is also shown in Figure 1. From Figure 1, it can be seen that our serial code and parallel code run in $O(n)$ time. Also, it can be seen that the parallel code runs faster than the serial code unless the number of particles is extremely small, in that case the overhead would be larger than the computation time and thus the parallel code would be slower. It should be noticed that as the Z-Morton encoder we used has only 4 available digits, so for the last two columns in Table 1 when h is small the code overflows, particles that should belong to different bins are mapped to the same hash entry. This explains why the run time is a bit longer than $O(n)$.

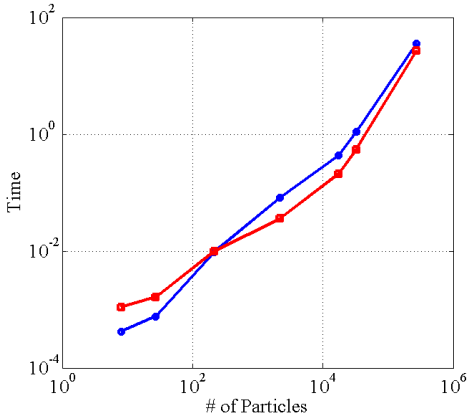


Figure 1: Run time of our serial code (blue curve) and parallel code (red curve) with different numbers of particles. Our code is approximately in $O(n)$ time.

2 Data structure

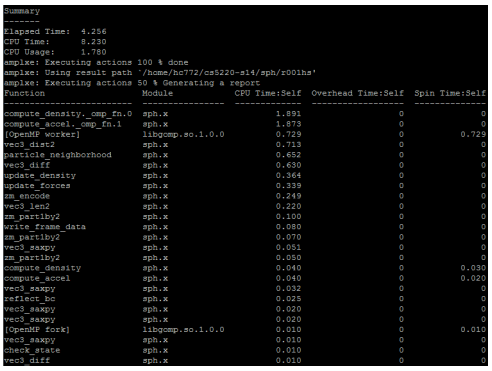
The original code is in $O(n^2)$ time, to improve this we use spatial binning and hashing.

- The function *particle_bucket* returns the Z-Morton code of the location of a particle.

- In the function *particle_neighborhood* we traverse the 27 neighboring bins of the current particle, we record the Z-Morton codes of legal neighboring bins and return the number of legal bins.
- In the function *hash_particles* we first clear the hash table, then we traverse all the particles and regenerate an updated hash table. Each entry of the hash table stores the first particle in the corresponding location, each particle also has a pointer to the next particle in the same bucket.
- We also added the function *particles_relocation*, where we relocate the particle storing order. We want particles in the same bucket stored in nearby memory regions, thus to maximize cache efficiency.

3 Profiling and bottlenecks

Using the script from the lecture slides, we got the profiling shown in Figure 2.



```
Summary
-------
Elapsed Time:  4.256
CPU Time:      8.230
CPU Usage:     1.780
amplxe: Executing actions 100 % done
amplxe: Using result path  '/home/hc772/cs5220-s14/sph/r001hs'
amplxe: Executing actions 50 % Generating a report
Function                 Module                CPU Time:Self  Overhead Time:Self  Spin Time:Self
-----------------------  --------------------  -------------  ------------------  --------------
compute_density._omp_fn.0  sph.x                       1.891                   0               0
compute_accel._omp_fn.1  sph.x                       1.873                   0               0
[OpenMP worker]          libgomp.so.1.0.0            0.729                   0           0.729
vec3_dist2               sph.x                       0.713                   0               0
particle_neighborhood    sph.x                       0.652                   0               0
vec3_diff                sph.x                       0.630                   0               0
update_density           sph.x                       0.364                   0               0
update_forces            sph.x                       0.339                   0               0
zm_encode                sph.x                       0.249                   0               0
vec3_len2                sph.x                       0.220                   0               0
zm_part1by2              sph.x                       0.100                   0               0
write_frame_data         sph.x                       0.080                   0               0
zm_part1by2              sph.x                       0.070                   0               0
vec3_saxpy               sph.x                       0.051                   0               0
zm_part1by2              sph.x                       0.050                   0               0
compute_density          sph.x                       0.040                   0           0.030
compute_accel            sph.x                       0.040                   0           0.020
vec3_saxpy               sph.x                       0.032                   0               0
reflect_bc               sph.x                       0.025                   0               0
vec3_saxpy               sph.x                       0.020                   0               0
vec3_saxpy               sph.x                       0.020                   0               0
[OpenMP fork]            libgomp.so.1.0.0            0.010                   0           0.010
vec3_saxpy               sph.x                       0.010                   0               0
check_state              sph.x                       0.010                   0               0
vec3_diff                sph.x                       0.010                   0               0
```

Figure 2: Profiling of the code using the script from the lecture slides.

From the profiling it can be seen that the bottlenecks of our code is the nested double for loops in the functions *compute_density* and *compute_accel*. More specifically, most time is spent in the *while* loop of updating particle data and go to the next particle until the next pointer is *NULL*.

4 Design choices

We implemented two additional designs in terms of synchronization and locality issues.

- We added two *#pragma omp parallel for* to parallelize the two main loops in *compute_density* and *compute_accel*. To further improve the performance, we added *shared(p,hash)*. Variable *p* points to the particles for current state and variable *bins* is used for finding adjecent bins. This improves the run speed by 43.6%.
- To improve the locality, we added the function *particles_relocation* which we call every 10 steps. In this function we put particles belonging to same bins into nearby memory regions. This improves the run speed by 11.2%.

5 Speedup

We tested our code using numbers of cores from 1 to 8. Figure 3 shows the speedup plot.

From Figure 3 we can see that when the number of cores increases from 1 to 4, the speed of our code also increases. When the number of cores is 5, the speed is lower than 4 cores, this is because there are two cpus, each has 4 cores, the communication is more expensive between two cpus than within one cpu, cache sharing also effects this.

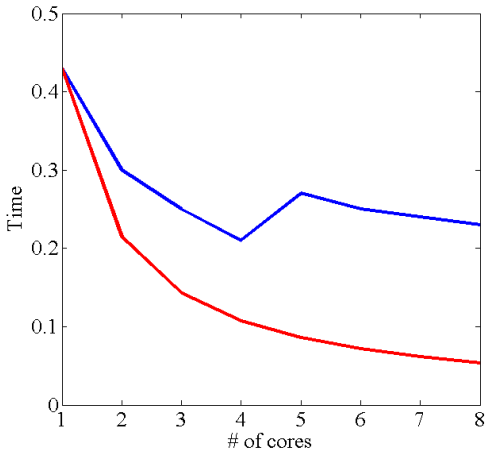


Figure 3: Speedup plot of actual result (blue curve) and ideal result (red curve).

6 Further discussions

Divide the time into computation time and synccomm time, using the result in Figure 3, the synccomm time for one step is $0.88s$ for two cores, $1.08s$ for within one cpu, and $1.79s$ for between two cpus. To handle more processors, a possible solution is to first divide all particles into several parts based on there location, and then assign them to different processors. To handle more particles, the Z-Morton encoder should be changed to using more bits, thus to prevent code overlapping.