Parallelizing a Particle Simulation - OpenMP

Tongge Wu[1], Debbie Liang[2], Zhuoming Zhang[2]

[1] *Department of Mechanical Engineering, University of California, Berkeley*
[2] *Department of EECS, University of California, Berkeley*
February 20th, 2020

1 Introduction

Molecular dynamics simulation, as a well-studied problem in scientific computing, is widely used in biochemistry, mechanics, and astrophysics. The basic problem setting is as follows:

- Given a domain (i.e., square) and boundary conditions (i.e., fully elastic recover).
- Initialize positions and velocities of a group of particles with prescribed masses.
- The force between any pair of particles is a prescribed function of the distance between the two particles.
- All particles move within the domain according to Newton's second law in a finite time interval.

In this assignment, we optimized a naive molecular dynamic simulation on a serial machine and a shared-memory machine using OpenMP. Here is our contribution:

- Tongge Wu works on implementing serial and OpenMP, testing scalability, providing mathematical support to our algorithms and writing the report
- Debbie Liang works on implementing serial and OpenMP, testing scalability, searching for efficient algorithm, visualizing data, and writing the report
- Zhuoming Zhang works on implementing serial and OpenMP, testing correctness, and working on timing analysis using HPCToolkit, and writing the report

2 Serial Machine

2.1 Cell list and neighbor list

A naive particle simulation cost FLOPS of $\mathcal{O}(n^2)$, where n is the number of particles. Because for each particle, we need a loop to scan all other particles to get the total force. In general, *Cell Lists* and *Verlet Neighbor Lists* are two common approaches to optimized the naive particle simulation [1].

We proposed to use Cell and Neighbor Lists together. The outermost loop was all particles (like Neighbor List), for each particle, we located the cell it belonged to (like Cell List), where the cell size was the same as the cut-off value in the force function. In the inner loop, we swept nine cells (eight neighbor cells plus the center cell) around the center cell (like Neighbor List) to accumulate the force on this particle. Because this particle could potentially generate force with all particles in the center cell, as well as in the eight neighbor cells.

When all particles' forces, hence, accelerations were calculated, we would move the particle for a given time step using the Verlet ODE integrator.

To build such a framework, we kept a data structure 'vector<vector<list<int»>' to keep a map between cells and particles within the cell. This map was updated after each time iteration.

2.2 Symmetric force matrix

If we have a force matrix, where each row and column represents a particle, and each nonzero entry represents the magnitude of the force between the two particles. It is trivial to argue that the force matrix would be sparse and symmetric due to Newton's third law. Therefore, we can half our FLOPS using this fact that if we calculate f_{ij}, there is no need to calculate $f_{ji} = -f_{ij}$. In terms of performance, however, due to the overhead of additional `if` statements, the speedup is about 10%.

2.3 Unroll

To iterate through the 9 cells surrounding the current particle, we first created a vector that holds the 9 cells and used a for loop to iterate. In the process of optimizing our code, we discovered that unrolling the `for` loop to be 9 similar blocks of code improves the run time. The improvement may due to minimizing the branch prediction. In conclusion, unrolling sped up our code by up to a factor of 3.

2.4 Additional Optimization

Instead of iterating over the particles (the outermost loop), we iterated over each cell because it could eliminate unnecessary FLOPS to find the cell ID of a particle. This optimization did not work for all ranges of particle number. For example, performance decreased by 20% when particle number was 1K, but the performance increased 15% when the particle number was higher than 10K.

2.5 Performance

The results were checked with the naive particle simulation ($\mathcal{O}(n^2)$) solution. The raw performance data was in Table.1. A linear regression (Eq.1) suggested the linearity was high, and it indicated that our serial code achieved $\mathcal{O}(n)$ complexity. A log-log transformation plot also demonstrated the linearity between the simulation scale and the total computational time; see Fig.1.

Number of particles	Wall-time(s)
1K	0.253702
5K	1.33267
10K	3.80737
50K	24.5359
100K	50.0934
500K	256.352
1M	528.592

Table 1: Serial machine: relation between simulation scale (K: $\times 10^3$; M: $\times 10^6$) and total computation time

$$\log(T) = 1.12\log(n) - 0.95 \quad \text{with} \quad R^2 = 0.997 \tag{1}$$

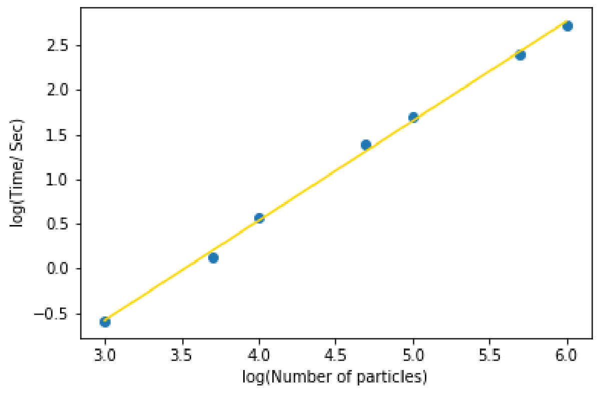

Figure 1: Serial machine: computational time as a function of simulation scale. Blue dots are numerical experiment results, while yellow line is the linear regression of these data points. All data are log transformed from Table.1

3 Shared-memory Machine

3.1 Spacial Locality

Initially, our parallel code was similar to the serial code (a hybrid of Cell List and Neighbor List). However, we realized that such a framework could make communication between threads very expensive. If we simply use multi-threads in the loop of accumulation force of each particle and the loop of moving each particle, there would be certain communication between threads when they are updating (read and write back) the position of each particle independently.

The cost of the above communication could reduce by changing the loop over particles to a loop over cells. The rationale behind the new design was particles were likely to stay in a group of cells that were adjacent to each other. If each thread took care of a group of cells, it is more likely that a particle position would be updated by the same thread for a few time steps. Hence, it reduces the communication between threads.

3.2 Synchronization

3.2.1 Atomicity

Derived from our serial framework, we first updated all particles' forces (loop of `apply_force()`), then separately updated all particles' position (loop of `move()`) within one time step. If we added `#pragma omp for` ahead of each loop, OpenMP guaranteed that the forces were accumulated, and particles were moved with no asynchronization. The speed-up from OpenMP was linear with respect to the number of threads ideally.

3.2.2 OpenMP Lock

Derived from our serial framework, we have a map between cell ID to particle ID. When multiple threads are trying to write the shared map, a data-racing may happen. The writing cases happen whenever a particle's position was updated, and the particle moves to another cell that is controlled by another thread.

To solve these issues, the first thing that came to our mind was to use `#pragma omp critical`. However, a critical region in OpenMP could result in a slow-down, and the algorithm approached the serial performance. After that, we used OpenMP lock, since we only need to lock a cell (or a particle) when any thread was trying to update that cell (or that particle). In conclusion, we had a 2D vector of `lock_t` to lock 2D vector of cells and a vector of `lock_t` to lock the vector of particles. Comparing to using `#pragma omp critical`, we increased our speed by a factor of 2.

3.3 Performance

The results were checked with the naive particle simulation ($\mathcal{O}(n^2)$) solution.

3.3.1 Complexity with simulation scale

When we fixed the number of cores to be 68, and progressively scaled up the simulation, the computation time was collected and listed in Table.2. A linear regression analysis showed that the complexity was $\mathcal{O}(n)$ (Eq.2). The data was also shown in Fig.2.

$$\log(T) = 1.12\log(n) - 3.93 \quad \text{with} \quad R^2 = 0.997 \tag{2}$$

Number of particles	Wall-time(s)
50K	0.866901
100K	1.82691
200K	3.70489
400K	7.30698
800K	14.474
1M	17.2298
2M	36.6943
4M	70.8856
8M	147.385
10M	180.926
20M	373.195
40M	759.171
80M	1643.93

Table 2: Shared-memory machine: relation between simulation scale (K: $\times 10^3$; M: $\times 10^6$) and computation time for 68 cores

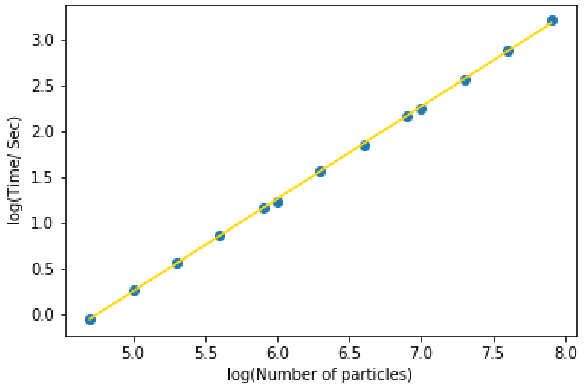

Figure 2: Shared-memory machine: computational time as a function of simulation scale. Blue dots are numerical experiment results, while yellow line is the linear regression of these data points. All data points are log transformed from Table.2

3.3.2 Weak scaling

To obtain the weak scaling characteristic, we assigned each core with 150K particles, then progressively increased the number of cores. The computation time was in Table.3.

If we define the Efficiency as

$$\text{Efficiency} = \frac{\text{Time(1 core)}}{\text{Time(p cores)}}$$

we can plot the efficiency as a function of number of core, see Fig.3. We observed that our efficiency decreased when the number of cores was less than 8, it indicated that within this range, increasing number core increased overhead of communication, and the overhead was more significant than the benefit of parallelism. We also observed that our efficiency went to a plateau when the number of cores was larger than 8 (about 43%), it indicated that the overhead spread into all threads calculation, the overhead effectively got hidden.

Number of cores	Number of particles	Wall-time(s)
1	150K	78.5878
2	300K	99.9693
4	600K	146.308
8	1.2M	169.719
16	2.4M	171.084
32	4.8M	182.736
64	9.6M	183.489
68	10.2M	184.355

Table 3: Weak scaling: relation between simulation scale and computation time for fixed number particle per core (150K)

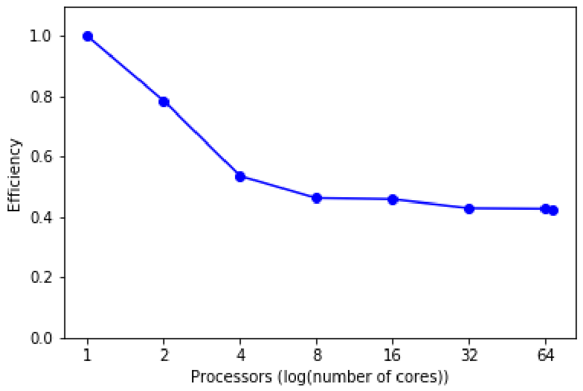

Figure 3: Weak scaling: efficiency as a function of number of cores.

3.3.3 Strong scaling

To obtain strong scaling characteristic, we fixed the simulation scale (10M particles) and progressively increased the number of cores. The computation time was in Table.4.

If we define the Speedup as

$$\text{Speedup} = \frac{\text{Time(p core)}}{\text{Time(1 cores)}}$$

we can plot the speedup against the number of cores, see Fig.4. The plot suggested we have a relatively good strong linear scale from the OpenMP code. However, we could not be able to reach the ideal case due to locks. When other threads locked a specific element (a cell or a particle), one thread had to wait until other threads until unlock happened. One possible way to minimize the communication between threads is to change the sub-cell's shape so that their total perimeter is minimized.

3.4 Runtime Breakdown

To obtain the timing analysis of our OpenMP code, we used HPCToolKit installed on Cori. The communication time is the summation of gomp_mutex_lock, gomp_mutex_unlock, and do_wait time. Basically, We fixed the simulation scale to be 1M particles and progressively increased the number of cores,

Number of cores	Wall-time(s)
1	7016.22
2	4684.34
4	2825.9
8	1461.57
16	737.41
32	373.18
64	190.213
68	179.729
136*	94.0628
272*	53.0355

Table 4: Strong scaling: relation between number of core and computation time for fixed number particle (10M), * hyper-threading

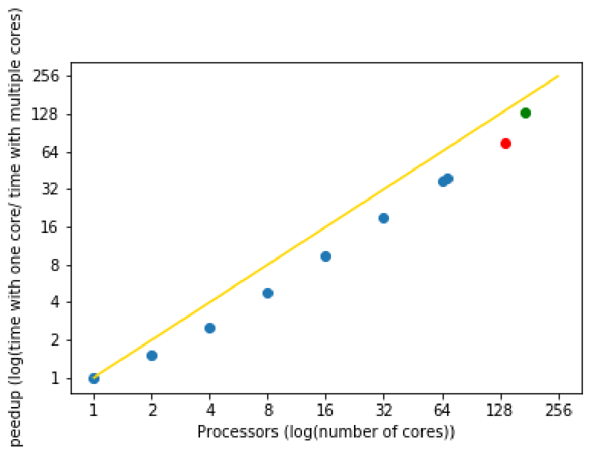

Figure 4: Strong scaling: speedup as function of number of core for a fixed simulation scale. The golden line indicates the ideal speedup with slope = 1. Blue dots are 1-68 cores data in Table.4, red dot is 136 threads (2 hyper-thread per core), green dot is 272 threas (4 hyper-thread per core)

see Fig.5. As shown in the figure, we observed that synchronization time fraction decreased initially and increased significantly when the number of core are over 64. To explain this behavior, we thought that when we requested 4 "cores" in the OpenMP, the hardware actually returned us 4 threads within one core, considering that on KNL node, each core has 4 hyper-threads. When the requested number of "core" is larger than 4, the hardware returned us more than one core; this was the case where the overhead between threads showed up.

4 Conclusion

In this HW, we optimized the serial molecular dynamic simulation, designed and optimized parallel molecular dynamic simulation using OpenMP. In the serial case, we achieved O(n) complexity, which was better than naive O(n^2) complexity. In the parallel case, we also achieved O(n) complexity. Moreover, the weak scaling test shows our code reached a plateau when the number of core was greater than 8, indi-

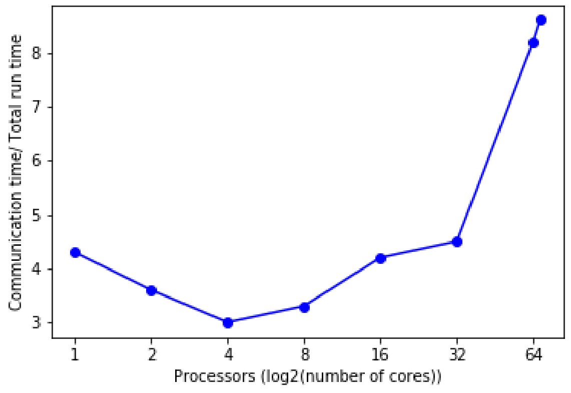

Figure 5: Runtime analysis: Communication time fraction as a function number of cores, given a fixed simulation scale (1M particle)

cating consistent performance within a computing node. On the other hand, the strong scaling test shows our code linearly scaled up to 272 threads (using hyper-threading), which is optimal on a computing node on Cori. One of the extreme cases that we tested was 100M particle on 68 cores using 272 threads, the computation time was 588 seconds (0.16 hr). If the same simulation was done on a single core, it would cost 14 hrs to finish. In conclusion, we achieved a reasonable parallel version of the basic molecular dynamic simulation.

References

[1] Victor Eijkhout with Robert van de Geijn and Edmond Chow. *Introduction to High Performance Scientific Computing*. `http://www.tacc.utexas.edu/~eijkhout/istc/istc.html`. lulu.com, 2011. ISBN: 978-1-257-99254-6.