Particle System Simulation Optimization

Xiaoxu Na and Shun-Chiao Chang

**Abstract** – Using binning method to optimize particle system simulation.

**Index Term** – serial, openMP, MPI, optimization

**1 Introduction**

In this project, we worked on parallelizing a particle system on a shared memory system and a distributed memory system.

In our particle system, particles interact by repelling each other. The interaction is by nearby force, and there is a relatively short cutoff distance. If the distance between two particles is larger than the cutoff distance they do not interact. Otherwise there is a certain replelling force between them specified in the function apply\_force() in common.cpp. The particles move according to Newton’s Laws.

**2 Algorithms and Code Structure**

**2.1 The Naïve Algorithm**

The naïve algorithm is straightforward that computes the forces on the particles by iteration through every pair of particles, thus it is expected the asymptotic complexity of the simulation is O(n2).

**2.2 The Optimized Serial Algorithm and Structure**

Considering the serial algorithm optimization, first, we came out with the method of binning. We made an array of bins and each bin contain typically 2 to 3 particles and this section is called binning. In order to keep the flexibility, instead of using a simple array, we chose std::vector which is a pre-constructed container in c++ and it has several convenient functions. Besides, since there will be 2 to 3 particles pushed into each bin, std::list is a good option, especially for its easy removal and auto resizable ability.

The bin size is defined as a square with the dimension of 2 \* cutoff due to experiments on 1 \* cutoff, 2 \* cutoff and 3 \* cutoff. 2 \* cutoff worked the best since it is not bigger than cutoff \* 1 which comes out with more bins, on the other hand, cutoff \* 3 comes out that each bin contains 6-8 particles, which does not contribute enough to reduce the complexity.

Instead of going though all the bins as of naïve algorithm, the particles of the optimized algorithm only needs to go through all 8 neighbor bins and itself, which is a small amount multiplies with n, thus the complexity goes down from O(n2) to nearly O(n). This portion helped a lot on computing the forces.

Then comes to the moving part. Together with movement, we should take care of those particles that moved out of its original bin before going to the next iteration. We moved each particle and compared every moved particle’s position to check if it was still in the same bin. If it went out of the bin, remove it from the current bin and just simply push it into a temp list. After they were all moved, we had a collection of temp list that contains all the particles that needs re-binning. We went though this list to push them into the new bin and clear the list at the end.

**2.3 The Optimized OpenMP Algorithm and Structure**

OpenMP is an easy way of parallelization. There are significantly 4 sections in the serial algorithm, binning, computing forces, moving and re-binning. We made the master thread did the first and last section, leaving computing forces and moving for paralleling threads processing. Concerning the section of computing forces, we need an omp reduction of navg and davg at the same time with all particles shared. While coming to the moving part, each thread need a separate list, thus we made padding the list to a vector with the index of the thread number.

**2.4 The Optimized MPI Algorithm and Structure**

MPI is another way of parallelization, however it is more complicated and sufficient comparing with OpenMP. Unlike OpenMP, processes of MPI do not have shared memory, thus we need to distribute data from the root process to all others and gather the results back to the root at the end.

For convenient function utilization, we made the binning and applyForceToBin inline functions so that each process can use the function dynamically. First, the root process initiated all the particles and broadcast to all other processes, so each process had an entire copy of all particles. Although they all did the binning to all particles, each process then was in charge of only several rows of the bins that were distributed as number of the rows of the bin divided by number of the process. Then the move were divided into two groups, one was the local move list which contained the moved particles within the same process, and the other was the remote move list which contained those particles went out of the area of the original process management. The only communication would be the remote move group. While we put remote move into lists, it was necessary to allocate them into arrays due to the sending and receiving buffers of MPI are array structure on the bottom. After the remove move particles were gathered at the root process, it was time to scatter the particles to the corresponding processes that would manage the scattered particles and re-bin them to the new bins.

**3 Experiment Results**

**3.1 The Naïve vs. Optimized Serial Algorithm**

**serial - naive**

500 4.49939

1000 17.8947

2000 73.5546

4000 303.358

6000 644.423

Serial code is O(N^slope)

Slope estimates are : 1.991731 2.039283 2.044134 1.858216

Slope estimate for line fit is: 2.009265

**serial - optimized**

5000 6.7332

15000 20.5763

20000 27.4447

40000 55.2431

60000 87.8858

Serial code is O(N^slope)

Slope estimates are : 1.016819 1.001220 1.009267 1.145092

Slope estimate for line fit is: 1.027406

**3.2 The Naïve vs. Optimized openMP Algorithm**

**openMP - naive**

500 0.644094

500 1 4.60934

500 2 2.39668

500 4 1.26823

500 6 0.893189

500 12 0.533691

500 18 0.433656

500 24 0.441833

500 32 0.800103

1000 2 9.71025

2000 4 19.5053

3000 6 29.4135

6000 12 58.8271

9000 18 88.1273

12000 24 123.094

16000 32 268.39

Strong scaling estimates are :

0.14 0.27 0.51 0.72 1.21 1.49 1.46 0.81 (speedup)

0.14 0.13 0.13 0.12 0.10 0.08 0.06 0.03 (efficiency) for

1 2 4 6 12 18 24 32 threads/processors

Average strong scaling efficiency: 0.10

Weak scaling estimates are :

0.14 0.07 0.03 0.02 0.01 0.01 0.01 0.00 (efficiency) for

1 2 4 6 12 18 24 32 threads/processors

Average weak scaling efficiency: 0.04

**openMP – optimized**

5000 6.72051

5000 1 0.827446

5000 2 0.437314

5000 4 0.23603

5000 6 0.175107

5000 12 0.100155

5000 18 0.106487

5000 24 0.139663

5000 32 0.243005

10000 2 0.831386

20000 4 0.86903

30000 6 0.881964

60000 12 0.926128

90000 18 0.920795

120000 24 1.03447

160000 32 1.07086

Strong scaling estimates are :

8.12 15.37 28.47 38.38 67.10 63.11 48.12 27.66 (speedup)

8.12 7.68 7.12 6.40 5.59 3.51 2.00 0.86 (efficiency) for

1 2 4 6 12 18 24 32 threads/processors

Average strong scaling efficiency: 5.16

Weak scaling estimates are :

8.12 8.08 7.73 7.62 7.26 7.30 6.50 6.28 (efficiency) for

1 2 4 6 12 18 24 32 threads/processors

Average weak scaling efficiency: 7.36

**3.3 The Naïve vs. Optimized MPI Algorithm**

**MPI - naive**

500 0.647064

500 1 4.45359

500 2 2.27995

500 4 1.17055

500 6 0.80781

500 8 0.630182

1000 2 8.96745

2000 4 18.6793

3000 6 28.0924

6000 8 84.2137

Strong scaling estimates are :

0.15 0.28 0.55 0.80 1.03 (speedup)

0.15 0.14 0.14 0.13 0.13 (efficiency) for

1 2 4 6 8 threads/processors

Average strong scaling efficiency: 0.14

Weak scaling estimates are :

0.15 0.07 0.03 0.02 0.01 (efficiency) for

1 2 4 6 8 threads/processors

Average weak scaling efficiency: 0.06

**MPI - optimized**

5000 6.74093

5000 1 7.024

5000 2 3.37506

5000 4 1.65818

5000 6 1.13798

5000 8 1.0596

10000 2 6.91014

20000 4 6.91481

30000 6 7.27235

60000 8 11.1864

Strong scaling estimates are :

0.96 2.00 4.07 5.92 6.36 (speedup)

0.96 1.00 1.02 0.99 0.80 (efficiency) for

1 2 4 6 8 threads/processors

Average strong scaling efficiency: 0.95

Weak scaling estimates are :

0.96 0.98 0.97 0.93 0.60 (efficiency) for

1 2 4 6 8 threads/processors

Average weak scaling efficiency: 0.89

**4 Conclusions**

In the experiment, we ran the naïve parts based on a smaller amount of particles (500-6000), while on the optimized part, we tried a much larger number of particles (5000-160000).

For the optimized serial algorithm, we finally came to the time complexity of O(n) as expected. For the optimized openMP and MPI, they showed much better speedup and efficiency.

**Acknowledges**

The author would like to express the gratitude to Dr. Panruo Wu.