Parallelization of Tree Algorithm in N-body Simulation[*]

Huai-Hsuan Chiu[†]

Jan 22, 2020

1 Introduction

The dynamic of a self-gravitation system can be simulated in two different ways. Naively one would think of direct N-body integration, which involves computation of $N(N-1)/2$ forces to be calculated in each time step. The second way is to model the potential field by the density distribution in the region of interest (solving the Poisson equation) and calculate the force on each particle. Number of computations of this approach grows only $N \log N$. 1986, Josh Barnes and Piet Hut published an algorithm that directly calculated the force interactions between each particles, or 'clusters', and the cost growth with only $N \log(N)$. This is known as the Tree Algorithm of N-body simulation.

The main idea of tree algorithm is that if we want to calculate force of a N-body system that acts on a single particle A, we directly sum the forces of the neighboring particles and view the particles far from A as a single particle, presented by its center of mass. Details of the algorithms and the parallelization is described in the following sections.

2 Method

2.1 Algorithms

The main algorithm of the tree algorithm can be presented in three parts : Tree building, Force computing and Particle evolving.

2.1.1 Build the Quad-Tree

This report mainly discuss about the 2D N-body problem. For 3-dimension cases, one should build Oct tree. First, begin with an empty large cell that can contain all of the particles. Next, load the particle into the box one by one. As long as there are two particles exists in a single cell, divide the cell into four sub-cells. At the end, each cell (no matter how large or small) will contain only one single particle.

As for the tree structure, each node of the tree contains the information of the corresponding cell and its sub-cells. The nodes records the number and the center of mass of particles inside. as

[*] The code can be found on github.

[†] Student Number : B07202025

long as the side length. If the particles distribute uniformly, the height of the tree has an upper bound of $\log_4 N$ the cost of building the tree is $N \log_4 N = O(N \log N)$.

2.1.2 Compute the Force on each particles

Assuming we want to calculate force of node n on the single particle k. If there is only one particle in node n, calculates the force between it and k directly. If not, calculate the distance of particle k and the center of mass of the node. Let the distance be r and the side length of the node n to be D. Set a fixed accuracy parameter θ. If $\frac{D}{r} > \theta$, the distance between the particle k and the node n is larger than the side length of the cell. We can view the particles in the cell as a cluster and use only their total mass and center of mass rather than calculating particles inside it respectively. To calculate the force of a single particle takes at most $O(logN)$ since the depth of the tree is $\log_4 N$ while the step takes total $N \log_4 N = O(N \log N)$

2.1.3 Evolve Particles

Last but not least, evolve the velocity and position of the particles in the system with the force already calculated. Also, calculate the total to estimate the error of each time step.

2.1.4 Boundary condition and Softening Length

In this project, I applied open boundary condition. That is to say that once the particle exceed the given boundary, remove the particle from our data. Hence, there is possibility that the total particle number is decreasing in some initial conditions.

Moreover, while calculating forces and potential of each particles, I applied softening length ϵ. When the distance between two particles is less than ϵ, I calculate the force by $\vec{f_i} = \frac{m_i m_j \vec{r_{ij}}}{(r_{ij}^2 + \epsilon^2)^{3/2}}$ to smooth out the force under this length scale.

2.2 Parallization

Procedure of applying tree algorithm into multiprocessors can be divided into four main parts. In this report I'll mainly discuss how I parallelize the algorithm with 1 GPU. In this section I briefly describe the idea of parallelization. More details about the code can be found in the data structure section.

2.2.1 Domain decomposition and mapping

Divide the domain into r sub-regions. The number of sub-regions (r) is suggested to be larger than or equal to the number of processors(p), which is the number of total threads in GPU. I use the *split* kernel to run over all the particles and label which region they belong to.

2.2.2 Tree Construction

Then, each processors calculate the tree within each sub-regions, and merge the local tree into global trees. I use the kernel *merge_bottom*, *merge_bottom2* and the *atomicAdd* function to calculate the center of mass and the total mass in each sub-regions. These are the only information to create the global tree. Then, I merge these information to create a global tree in kernel *merge_top1* and *merge_top2*.

2.2.3 Force Evaluation

In general, to calculate the force on a single particle k, there are three situations that should be taken into consideration.

- k-to-all broadcast : Starting from the root of the tree, particles far from the target particle k is considered as a single particle and the information can be taken from nodes near the roots. Since these node will be frequently accessed, we let them to be accessible to every threads.

- Force evaluation : For particles near k, the information is in the local node and one can calculate it in the each processor at the same time.

- All-to-all broadcast : There are still some particles that is close to particle k but in the different sub-regions. We pass the information of k (three float number) to the corresponding processor and calculates forces at that processor. Passing the force (also three float number) to the origin processor and sum them up.

In this project, I handle only the k-to-all broadcast and force evaluation. That is to say that for every particles that is not in the same region with particle k, their effects can only be considered by the center of mass of the region they belonged to. Also, I do the force evaluation by direct N-body instead of tree algorithm within a single region. I do load balance when calculating the direct N-body in each sub-regions (See Data structure section).

2.2.4 Moving particles

Known the force acting on each particles, a simple parallelization on particles can evolve the positions and velocities of particles.

2.3 Data Structure

2.3.1 Tree Node of CPU code : linked list

In the CPU code of tree algorithm, I use a self-defined structure 'NODE' to store the information in each node. Each node contains the center of the cell, center of mass, total mass and the number of particles withing the node. The pointers NODE* next[4] will be linked to the pointer of sub-nodes. If there are no sub-nodes, the pointer will point to NULL.

2.3.2 Tree Node in GPU code : quaternion tree in array

With the center of mass and total mass known in each sub-regions, we use a full-size quad-tree to merge these information. If we split the whole box into $n_x \times n_y$ sub-regions, the global tree will have $n_x \times n_y$ leafs. For simplicity, let $n_x = n_y$. We need $n_x \times n_x + \frac{n_x}{2} \times \frac{n_x}{2} + ... = 1 + \frac{4}{3}(n_x^2 - 1)$ elements to create an array *root*. For every elements $A[i]$ the *root* array, its four children are $A[4i+1], A[4i+2], A[4i+3]$ and $A[4i+4]$, respectively. We fill in the late n_x^2 element in the *root* array and fill in the successive $\frac{n_x^2}{4}$ elements until we fill in $A[0]$.

2.3.3 Force calculation in GPU code : stack and while-loop instead of recursion

To calculate forces with the *root* array (global tree), we start from the root of the whole tree. If the root node doesn't satisfy the condition mentioned in section 2.1.2, we take the children of the node as root and explore them recursively. However, recursive function is not supported in GPU device. Hence, I use stack and while-loop to replace the recurrence in CPU code. First ask for a int-array as the queue should be explored. Initialize the queue by $Q[0] = 0$ as the index of the root and the length=1. Take one element from the end of the queue and decrease the length of queue by 1 once. If the node satisfy the θ-condition, calculate the forces. If not, put the index of all its children in the queue and increase the length of queue by 4. The while-loop will stop until there are no elements in the queue.

2.3.4 Load balance : Min-heap

Assuming there are n_r particles in sub-region r, the calculation amount of the region is proportional to n_r^2. We first assign one sub-region to every thread. If the number of thread is more then the total number of sub-regions, there will be a waste when doing N-body calculation. Then, we create a min-heap such that the root of the heap contains the least calculation amount within all threads. We assigned one region to the thread at the root of the min-heap and do Max-heapify the maintain the min-heap. I applied the simple greedy algorithm that might not find the optimal solution but will give a OK running time in this project, since the optimal solution of this kind of assignment problem is NP-complete.

3 Results

We randomly initialize the position and mass of the particles and let the velocity of every particle to be zero, with a given number of particles. For the output *.dat* file, I record the position of the particles. We set three main particles size to test the performance of the code, they are $10^5, 10^6, 10^7$ and 10^8. In the following sections and tables, region size=32 means we use 32^2 sub-regions. Block size=16(Grid size=16) means using 16×16 threads(blocks) per block (grid). Particle sizes refer to the total number of particles.

In this section, I've confirmed that the energy is conserved with a reasonable time step in the following test cases used. I focus on the code performance mainly in this report.

3.1 Performance between CPU & GPU code

I run the CPU and GPU code with different particle sizes, and record the time of tree creation, force calculation, position update and the energy calculation. For CPU code, I add only the time of Tree creation, Force calculation and Update particles as the total time. The detailed reason can be found at section 3.1.3. For GPU code, I calculate the potential energy when evaluating the force and do the kinetic energy component when updating positions of particles. In conclusion, the 'tree creation' time for GPU involves domain decomposition, merging each sub-regions and creating the global tree. The 'Force calculation' time includes the k-to-all broadcast and the direct N-body calculation in each sub-regions, as long as the estimation of potential energy. The 'Update particle' section is the time of moving particles and updating the velocities. Noted that the total time for GPU is estimated from the beginning to the end of the code and includes the time for memory allocation.

Table 1: Performance of the CPU and GPU code. The dimension of time is ms.

Particle size	10^5		10^6		10^7		10^8	
	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
Tree creation	24.16	15.96	408.29	25.406	5924.59	187.748	155311.21	1704.04
Force calculation	275.34	134.83	4239.91	232.49	61606.03	1789.49	958744.25	19264.80
Update particles	0.5	3.16	7.14	21.71	58.89	211.348	165856.78	14030.21
Estimate energy	219.83	-	3493.81	-	48478.91	-	764410.93	-
Total time	299.98	231.00	4655.32	407.00	67589.21	2212.00	1279912.24	29190.00
Speed up	1.29		11.43		30.55		43.84	

Table 2: Parameters of GPU used in table 1.

	10^5	10^6	10^7	10^8
Sub regions	512x512	512x512	1024x1024	2048x2048
Block size	16x16	16x16	16x16	16x16
Grid size	16x16	16x16	32x32	32x32

From table 1, we can see that GPU do speed up the code, and the speed-up factor increases as the number of particle grows. The bottleneck process that slow down the code is force calculation, for both the CPU and GPU code.

3.1.1 Tree creation time

The Tree creation time comparison seems to be unfair for CPU, because GPU code doesn't construct the tree in each sub-regions. However, we can do a fairer comparison in the 10^5 and 10^6 cases. We take 512×512 sub-regions in these cases, and there is only 0.4 and 3.4 particles in each sub-regions. The speed-up of tree construction are 1.51 and 16.07 respectively.

Discussion The main difference between CPU and GPU code in tree construction is that we use different data structure to store the information. In CPU, using linked list can avoid memory waste in the region without particles. In comparison, using array is more easy for GPU to do parallel reduction while merging the sub-trees into a single global tree. Noted that there might be a deviation according to our test problem, which spreads particles uniformly in space. The performance might be worse when the particles is unevenly distributed.

3.1.2 Force calculation

This is the dominant component of the running time. Given the constructed tree, the CPU code should go through the tree for every particles with a for-loop, while GPU code parallelize the for-loop of particle size.

Discussion I did load balance in the direct N-body calculation in each sub-region. We can see that there is no significant difference with or without load balance (See table 3). This might be the result of the uniform distribution of the initial condition. The advance might be significant if the particles distribute un-evenly. Noted that I use the same parameters as table 2 here.

Table 3: Performance comparison of load balance. Time unit in ms.

Particle size	with load balance		without load balance	
	force	total	tree	total
10^5	119.05	247	125.57	201
10^6	215.06	407	231.36	396
10^7	1833.53	2212	1791.38	2278
10^8	19264.8	29190	19334.38	28283

Table 4: Average particle number per sub-region with different size of sub-regions.

side number	total number	10^5	10^6	10^7	10^8
64	4096	24	244	2441	24414
128	16384	6	61	610	6104
256	65536	2	15	153	1526
512	262144	0	4	38	381
1024	1048576	0	1	10	95
2048	4194304	0	0	2	24

3.1.3 Update particles & Estimate energy

This is the simplest part of the parallelization of tree algorithm. However, the speed-up factor seems to be less than 1 in most cases. The reason is I reorganize the particles with the order of its region (i.e. [particles in region1, particles in region2,...]). In the update section, I should reorganize the order of velocity to be consist with the order of positions and forces. To execute a large particle size such as 10^8, I spend some running in allocating and releasing memories. This extra time is mush smaller than the force calculation time and considered acceptable.

Moreover, the energy estimation time in CPU code is almost the same as force calculation because it contains the calculation of potential energy. This calculation will let all particles to walk through the entire tree again, and can be entirely merged into force calculation. So I add only the first three component for the total time in all CPU cases to make a fair comparison.

3.2 Optimal block/grid-size of GPU code

To find the optimal block/grid size of GPU code, I considered the fixed sub-regions by table 4. The colored block is the total number of sub-regions chosen to find the optimal block/grid size in this section.

I calculate the average run time of two time steps in all of the test cases with different block/grid size. I labeled the block/grid size with the highest speed-up factor. From table 5-8, we can see that the optimal block/gird size usually contributes the total thread numbers around the number of sub-regions. Also, low number of threads per block is not a good choice. Every optimal block/grid size exist at the block size=8 or 16.

Next, when the total number of threads (n_{th}) is larger than the number of sub-regions(n_r), there will be a lost when calculating direct n-body force within each sub-region. This may lead to a poor load balance and slow down the code performance. Unexceptly, since we control the average particle number within each sub-region in this paragraph, the waste of threads doesn't cause large problems. On the other hand, in 10^6 and 10^7 cases, the optimal block/grid size contribute

Table 5: Run time test for $10^5 (128 \times 128)$

grid size	block size	2	4	8	16
2	time	1255	404	185	104
	speed-up	0.24	0.74	1.62	2.88
4	time	403	182	59	87
	speed-up	0.74	1.64	5.07	3.44
8	time	192	102	58	39
	speed-up	1.56	2.93	5.16	7.67
16	time	183	87	39	35
	speed-up	1.63	3.44	7.67	8.54
32	time	160	61	34	35
	speed-up	1.87	4.90	8.79	8.54
64	time	151	58	35	37
	speed-up	1.98	5.16	8.54	8.08
128	time	156	59	38	45
	speed-up	1.92	5.07	7.87	6.64
256	time	163	67	50	74
	speed-up	1.83	4.46	5.98	4.04
512	time	168	86	91	176
	speed-up	1.78	3.48	3.29	1.70

$n_{th} = (256 \times 8)^2 > n_r = 256^2$ and $n_{th} = (256 \times 8)^2 > n_r = 1024^2$. The reason is that I parallelize the number of particles mainly in the algorithm. In the force calculation section, which is the bottleneck process in the entire code, this parallelization will not waste too much threads.

Last, noted that the last three cases in table 8 crush because the block/grid size requires too many threads. Memory in each threads cannot afford the assigned amount of calculation. The final result of the optimal block/grid size is list in table 9.

3.3 Optimal size of the sub-region

In this section, we take the 10^7 test case as an example to see how the running time changes with the region size (See table 10). Naively, I think the total running time will increase as the region size grows since the size of direct n-body calculation increases. However, the result is completely opposite to the original guess.

First of all, the run time of tree creation drops first then rises. This is because I use *atomicAdd* function to create the global free. If the number of sub-regions is small, there might be many particles want to access the same region at the same time and slow down the code.

Next, the number of region is the amount of leafs of the global tree. Since I drop the all-to-all broadcast in this project, the force resolution of the particles outside the region depend on the number of regions. That is to say that, the actual amount of calculation is decreasing as the number of region decrease. This might be a main reason of the decreasing run time. The tendency of decreasing run time might be wrong after adding all-to-all broadcast. The other main reason is that maybe tree algorithm is not suitable for parallelization comparing to direct N-body in this amount of particle numbers.[1]

[1] I think this answer can be answered after adding all-to-all broadcast feature, which is the one I have no time

Table 6: Run time test for $10^6 (256 \times 256)$

grid size	block size	2	4	8	16
8	time	1123	467	305	301
	speed-up	4.15	9.97	15.26	15.47
16	time	939	417	297	257
	speed-up	4.96	11.16	15.67	18.11
32	time	801	385	252	259
	speed-up	5.81	12.09	18.47	17.97
64	time	766	350	264	262
	speed-up	6.08	13.30	17.63	17.77
128	time	743	338	259	253
	speed-up	6.27	13.77	17.97	18.40
256	time	749	351	248	269
	speed-up	6.21	13.26	18.77	17.30
512	time	740	364	274	366
	speed-up	6.29	12.79	16.99	12.72

Table 7: Run time test for $10^7 (1024 \times 1024)$

grid size	block size	2	4	8	16
16	time	10661	3677	2675	2356
	speed-up	6.34	18.38	25.27	28.69
32	time	8694	3080	2353	2280
	speed-up	7.77	21.94	28.72	29.64
64	time	8406	2964	2276	2036
	speed-up	8.04	22.80	29.70	33.20
128	time	8316	2876	2034	1892
	speed-up	8.13	23.50	33.23	35.72
256	time	8199	2647	1895	1918
	speed-up	8.24	25.53	35.67	35.24
512	time	7964	2512	1906	2011
	speed-up	8.49	26.91	35.46	33.61
1024	time	7830	2539	2003	2371
	speed-up	8.63	26.62	33.74	28.51

Table 8: Run time test for $10^8 (2048 \times 2048)$

grid size	block size	2	4	8
16	time	40700	31535	28408
	speed-up	31.45	40.59	45.05
32	time	34600	28458	28225
	speed-up	36.99	44.98	45.35
64	time	33386	28262	33367
	speed-up	38.337	45.287	38.359
128	time	32883	27894	27042
	speed-up	38.92	45.88	47.33
256	time	32626	27002	26519
	speed-up	39.23	47.40	48.26
512	time	31731	26463	26579
	speed-up	40.34	48.37	48.16
1024	time	31169	26529	-
	speed-up	41.06	48.25	-
2048	time	31272	-	-
	speed-up	40.93	-	-

Table 9: The optimal block/grid size in each test cases.

Particle size	10^5	10^6	10^7	10^8
Block size	8	8	16	4
Grid size	32	256	128	512

Table 10: Run time of different region size.

Region size	Total time	Force calculation	Tree Creation
2048	7268	2448	300
1024	1886	1399	188
512	1484	1097	140
256	1224	854	59
128	1093	808	11
64	975	642	8
32	884	539	9
16	779	490	13
8	675	331	16
4	555	214	26
2	478	137	36

4 Conclusion

In this report, we implement the parallelization of tree algorithm by single GPU, and get the following conclusions:

- GPU can significantly speed-up the CPU version of tree algorithm. The speed-up factor can reach 43.84 when there are 10^8 particles.
- The bottleneck process that slow down the code is force calculation, which makes every particles to walk through the entire tree. I parallelize the for-loop on particles in the GPU version code, resulting in a great progress of the total running time.
- When deciding the optimal block/grid size, I found that small block size performs worse. Also, the total number of threads need not to be more than the number of sub-regions since I mainly parallelize the for-loop on particles.
- The optimal size of sub-regions can not be decided by the running time since the all-to-all feature is important to test cases with small sub-regions.

There are also some more future work that can be done in this project:

- Adding the all-to-all broadcast feature.
- Using different initial conditions to test the performance when the particles are un-evenly, distributed.
- Parallelize by multi-GPUs.

Reference

[1] A hierarchical O(NlogN) force-calculation algorithm, Josh Barnes & Piet Hu, 1986.

[2] Scalable Parallel Formulations of the Barnes-Hut Method for n-Body Simulations, Ananth Y. Grama, Vipin Kumar, and Ahmed Sameh, 1994.

[3] Cosmology Applications : N-Body Simulations, Lecture Slides of Dr. James Demmel and Dr. Kathy Yelick, University of California, Berkeley, 1999.

This project is inspired by the candidate final project ‘Tree Algorithm’ in the Computational Astrophysics Course, 2021, NTU.

to finish before the deadline.