INF560 Report

Towards Paralleled N-Body Interaction

Zhengqing Liu Jiongyan Zhang

March 2022

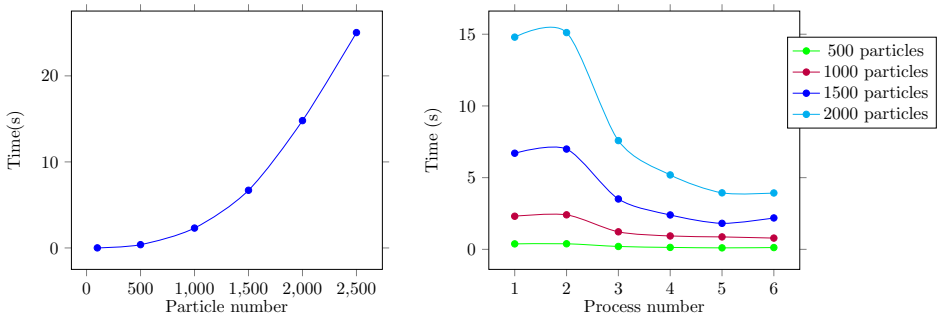
Abstract The N-body problem has been widely studied and applied in various fields. Since the traditional Brute Force algorithm is computationally expensive and over time-consuming, the Barnes Hut quad-tree algorithm is proposed to simulate the particle force, which greatly improves the efficiency of the algorithm. Nowadays, the advent of parallel computing has given new born to these legacy algorithms. With MPI, OpenMP, and CUDA, we enforce parallelization at various degrees to experiment with the performance improvement upon Brute Force and Barnes Hut algorithms. The performance evaluation by exploiting different parallel paradigms as per the input scope are given in details.

1 Introduction

N-body simulation consists of the evolution of a dynamic system of N bodies considering their mass, positions and initial speed, which is extensively used in astrophysics for studying how galaxies composed of millions of bodies behave [7]. Computers have been used to simulate N-body problems since the 1960s. Its main principle is based on Newtonian mechanics, simulating the force between each particle to calculate the direction and position at the next movement [1]. As the computing power increases, multiple algorithms (i.e., Barnes Hut, etc.) are used to replace this basic algorithm [5]. However, the development of computing power did not stop there. The availability of multicore processors has opened a new pan for high performance computing. Parallelization unleashes the multi-thread or multi-process capabilities of CPUs and GPUs, along with distributed storage or shared storage, to divide and conquer so that the algorithm can leverage the computing power to solve the problem.
In the rest of this paper, we target at optimizing two classic algorithms (Brute Force and Barnes Hut) with several parallel paradigms, and evaluate the performance of various modes.

2 Brute Force

Brute force is a naive algorithm that consists, for each time step, of computing the gravitational force applied to each particle, and then computing its new position respectively. The force applied to particle p_1 is computed by summing the force that all the other particles apply to p_1, which takes the complexity of $O(n^2)$. Hence, the loops of force computation and position movement are the two essential parts to unleash the capability of parallelism.



(a) Original (b) MPI

Figure 1: Orignal and MPI performance of Brute Force

2.1 MPI

MPI (Message-Passing Interface)[1] is a message-passing library interface specification. It addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process.

To utilize the distributed-memory paradigm, we define a *master* process and multiple *slave* processes according to the given processes. While slaves take force computation tasks equally, master finishes the modulo task and moves all the particles. Collective communication, such as broadcast and gather, facilitates the synchronization and collection. However, since the algorithm encompasses several arguments (i.e., force, velocity, position), synchronizing all of them would require broadcast with a couple of steps. To this end, a new MPI datatype, *particle_mpi_t*, is declared in our implementation, thus the whole particle structures can be transmitted at once. In case that the number of modulus task of master is not same as those of slave, gatherv is used to gather sets of particles in disparate lengths. In addition, the global variables, *max_acc* and *max_speed*, should also be updated among all the processes.

[1] https://www.mpi-forum.org/

Threads	1000 particles	2000 particles	3000 particles
1	2.587433	16.243086	43.273200
2	2.336531	14.792514	39.182745
3	2.255282	15.018447	42.185766
4	2.411418	15.249535	42.789421
5	2.452970	14.749308	43.151368

(a) 2 Processes

Threads	1000 particles	2000 particles	3000 particles
1	1.327676	8.315050	23.706969
2	1.894452	11.965878	33.056768
3	1.790665	10.809736	28.787725
4	1.654219	9.455634	24.729652
5	4.189442	9.502631	24.380480

(b) 3 Processes

Threads	1000 particles	2000 particles	3000 particles
1	0.899367	5.583197	15.528205
2	1.586002	9.062810	24.073398
3	1.414150	6.780683	22.774145
4	4.471012	9.574463	23.262464
5	5.686800	12.974246	26.736431

(c) 4 Processes

Table 1: MPI+OMP performance on Brute Force

2.2 MPI + OpenMP

The OpenMP (Open Multi-Processing)[2] supports multi-platform shared-memory parallel programming in C/C++ and Fortran. To leverage both MPI and OpenMP for N-body problem, a fine-grain mode of hybrid programming is implemented - distribution of tasks over MPI ranks and process of each subpart with OpenMP. Optimization of loop scheduling is a huge advantage of fine granurity. In our scenarios, each slave rank computes the forces added up to a set of particles, which are specified by the master. By creating omp parallel region for each slave, the for-loop of computing force can speed up. In this way, only few OpenMP constructs and clauses are involved and meanwhile relatively better performances are obtained.
In addition, setting more than 4 processes, the performance is degraded along with the increment of the number of processes. This phenomenon conforms to Amdahl's law. Not only cannot the sequential portion be diminished, but launching and cooperating threads or processes takes costs.

[2]https://www.openmp.org/

Processes	100 particles	500 particles	1000 particles
2 (No CUDA)	0.007055	0.385060	2.336531
2	0.631886	13.081082	78.902757
3	1.409775	22.233090	92.376449
4	1.477007	22.345278	109.797773

Table 2: Hybrid performance

2.3 MPI + OpenMP + CUDA

CUDA is a parallel computing platform and application programming interface that allows software to use certain types of graphics processing unit for general purpose processing. With the acceleration of dedicated hardware, parallelism can step further.
The unit task of computing all other forces upon one particle is transferred to CUDA. After allocating and copying memory to the device side, CUDA exploits its given blocks and threads to implement the force computation. Due to the computed forces are added up in the various threads, the atomicity should be guaranteed to eschew the asynchronization. Typically, the built-in *atomicAdd* is constrained to integer operations, thus rewriting the atomic adding for double type is necessary. Inspired by Marco Bertini's solution [3], *atomicAddDouble* is implemented to successfully execute double addition.
However, the performance is degraded with the involvement of CUDA. And we observe that with the increment of processes, the speed further slows down. Due to that OMP has trivial effects on the MPI version, we choose 2 omp threads and analyze the performance brought by CUDA. We choose *thr_per_block* as 10, 50, 100 for 100, 500, 1000 particles scenarios respectively.

3 Barnes Hut

The Barnes Hut algorithm exploits the sparsity of particles to estimate the force on each particle. Instead of calculating the interaction force among each pair of particles, the algorithm treats the distant group of particles as one large particle and performs mechanical calculations with it [6]. Via the approximate approach, one particle doesn't require to traverse all other particles.The core step in this algorithm is to establish a quad-tree, upon which one can gather space-neighbour particles [4].
The main idea of Barnes Hut parallelism is to make recursive structures loop. We mainly focus on the force computing part, as it is one of the most computing expensive steps and with the highest possibility to achieve parallelism [2]. The target is to separate the particles into several parts, which would be convenient for multiple threads or processes to handle independently. Meanwhile, we also try to use parallelism in the recursive function, aiming to finding a decent level to assign tasks to each thread. The original non-parallel code possesses the performance as below.

It shows that with development of the particle number, the code operation time increases with an approximate-linear model.

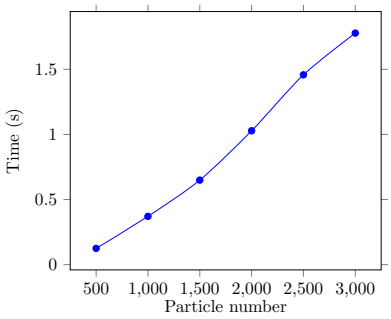


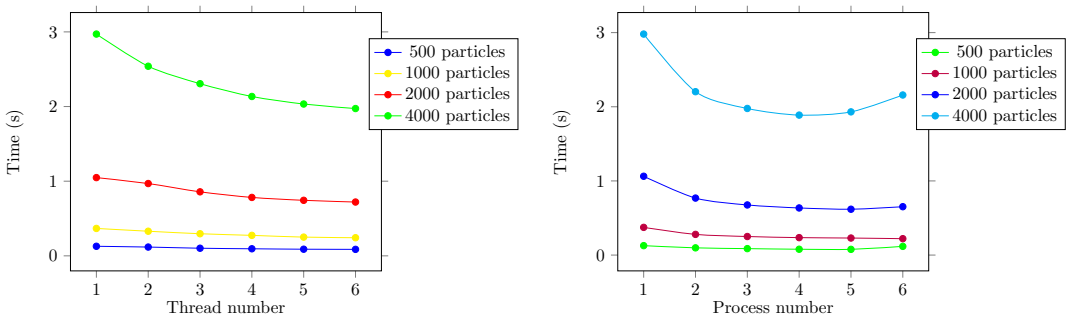
Figure 2: Original code performance of Barnes Hut

3.1 MPI

Due to the distributed memory system and the complexity of a quad-tree structure, we cannot transfer the tree directly. Thus, we use more space for less time. The basic idea is to establish the tree in each process simultaneously. Finally, the tree in each process is the same as the initial parameters are shared. Each process could obtain a certain quantity of tasks in correspondence to the process number. Based on 1000 particles, we add the required process gradually and the result is showed in Figure 3.

3.2 OpenMP

OpenMP provides a shared memory, thus the tree establishment in each thread is not required. We can just simply apply OpenMP for-loop with the dynamic mode. The parallelism mainly focuses on the force computing process. We try to increase the threads to examine the efficiency of OpenMP in this algorithm:



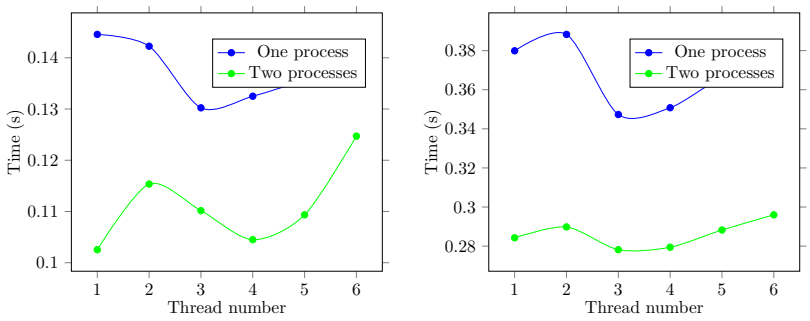
(a) MPI (b) OMP

Figure 3: MPI and OMP performance in Barnes Hut

The operating time decreases with the thread number increasing. However, it is clear that the speedup decreases, mainly because of the bus contention.

3.3 Hybrid of MPI and OpenMP

The hybrid mode of MPI and OpenMP could provide a further choice for parallelism. The combination of two tools comes from the previous two sections. The top level design provides an MPI infrastructure, which follows the each-tree-each-process principle in 3.1 to create an environment for labour division. Inside each process, the for-loop could be parallelized by OpenMP.



(a) 500 particles

(b) 1000 particles

(c) 2000 particles

(d) 4000 particles

Figure 4: Hybrid performance of Barnes Hut

We notice that there would be a time rising when applying the two threads. The possible reason is the additional thread cost.

We can then find that the increment of processes and threads contributes to the program's acceleration. However, the abnormal trend appears, i.e., there would be a strange rise with three-process. The context switching of threads probably counts a main reason as the threads in use may exceed the limitation of the device.

3.4 CUDA

Unlike MPI and OpenMP, the CUDA programming aims to utilizing numerous threads of GPU. The largest problem is how to transfer the tree structure from host to device. To simplify this problem, we try to establish the quad-tree on the device side directly. During the experiments, there are two problems: the stack space in GPU is relatively less, which could not support a normal establishment process; and, the computation of the

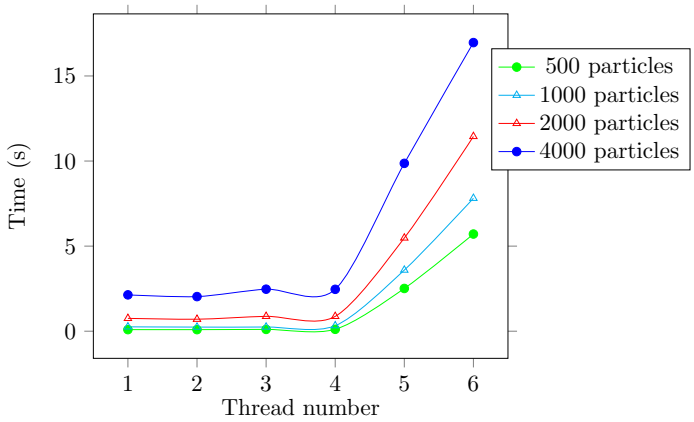


Figure 5: Abnormal trend in three processes

Thread number	1000 particles	2000 particles
20	16.026382	45.959570
1000	14.313604	40.887589
Demo (non-parallel)	0.398095	1.095418

Table 3: CUDA performance on Barnes Hut

normal-level amount of particles would be quite slower. Table 3 shows the efficiency of the algorithm based on 1000 and 2000 particles and that more threads lead to accelerate the program.

As the Barnes Hut is not independently computation-intensive, while requiring more space than the normal algorithm like Brute Force, GPU can not have an obvious speed-up on the program; instead, the cost greatly worsens the algorithm outcome.

3.5 Non-recursive algorithms

Non-recursive algorithms aims to solve the performance problem in CUDA. The main idea is to revise the recursive function to the non-recursive function. To fit the logic of Barnes Hut, Breadth-First Search is utilized to develop the non-recursive algorithm. A queue structure provides a container for the while-loop iteration search in the quad-tree. The problem still exists. First, the algorithm could not facilitate the performance in CUDA (with similar time consumption); second, the heap overflow problem occurs.

3.6 Other trials

(a) Linear quad-tree

The linear quad-tree is a strategy, which utilizes a 1D array where each index represents the Morton code of one 2D space cell. This structure is easy to be implemented and understood, which suits the particle separation and device or process transfer. However,

one great drawback is that this method requires a large space to generate redundant space for the linear structure.
After the experiment, the compiler error indicates that the heap space is used out. There are two main problems: first, the requested length number has already exceeded the limitation of the INT type; second, we can notice that based on 1000 particles, the depth is around 15 20, which requires about $4^{14} \sim 4^{19}$ cells for the array, which may be far more than the device burden limitation.

(b) OMP task in recursion

Though the recursion structure is not suitable for parallelism, OpenMP provides an *omp task* function, which may take convenience for recursion parallelism. However, in our experiment, the time consumption would slightly rise if combined with *omp task*. After granting each task operating condition (based on the node depth, forbidding deep omp tasks), we found in a more shadow level, less the time. We assume that the for-loop has already taken all the threads thus it might be a performance lose with the task.

4 Conclusion

In this work, we target at paralleling N-body simulation (Brute Force and Barnes Hut) with MPI, OpenMP and CUDA. In the Brute Force, MPI largely speeds up the computation, OpenMP provides slight enhancement in a fine-grain mode, while CUDA slows down the task at several magnitude of orders. In the Barnes Hut, the crucial point is to parallelize force computing as well. In face of a complex and huge tree structure, we can try to share the tree structure in the memory space or build the same tree structure, using space to acquire a more efficient parallel algorithm. Meanwhile, the control of thread overhead also greatly affects the performance of the program. Besides, the CUDA model is not suitable for such a non-computation-intensive task, which largely degrades performance. In both models, we observe the performance impairment when processes and threads are over some certain bounds. Launching paralleled regions, communication among processes, copying memory to dedicated hardware all contributes to such degradation.

References

[1] Sverre J Aarseth and Sverre Johannes Aarseth. *Gravitational N-body simulations: tools and algorithms*. Cambridge University Press, 2003.

[2] Maida Arnautović et al. "Parallelization of the ant colony optimization for the shortest path problem using OpenMP and CUDA". In: *2013 36th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE. 2013, pp. 1273–1277.

[3] Marco Bertini. *GPU Programming Basics*. 2017. URL: `https://www.micc.unifi.it/bertini/download/gpu-programming-basics/2017/gpu_cuda_5.pdf`.

[4] John Smith and S-F Chang. “Quad-tree segmentation for texture-based image query”. In: *Proceedings of the second ACM international conference on Multimedia*. 1994, pp. 279–286.

[5] Michele Trenti and Piet Hut. “N-body simulations (gravitational)”. In: *Scholarpedia* 3.5 (2008), p. 3930.

[6] Laurens Van Der Maaten. “Barnes-hut-sne”. In: *arXiv preprint arXiv:1301.3342* (2013).

[7] Wikipedia. *N-body Simulation*. [Online; accessed 13-March-2022]. 2022. URL: `https://en.wikipedia.org/wiki/N-body_simulation`.