香 港 中 文 大 學 (深 圳)
The Chinese University of Hong Kong, Shenzhen

# N-Body Simulation

Assignment III

CSC4005: Parallel Programming

Name: *Derong Jin*

Student ID: *120090562*

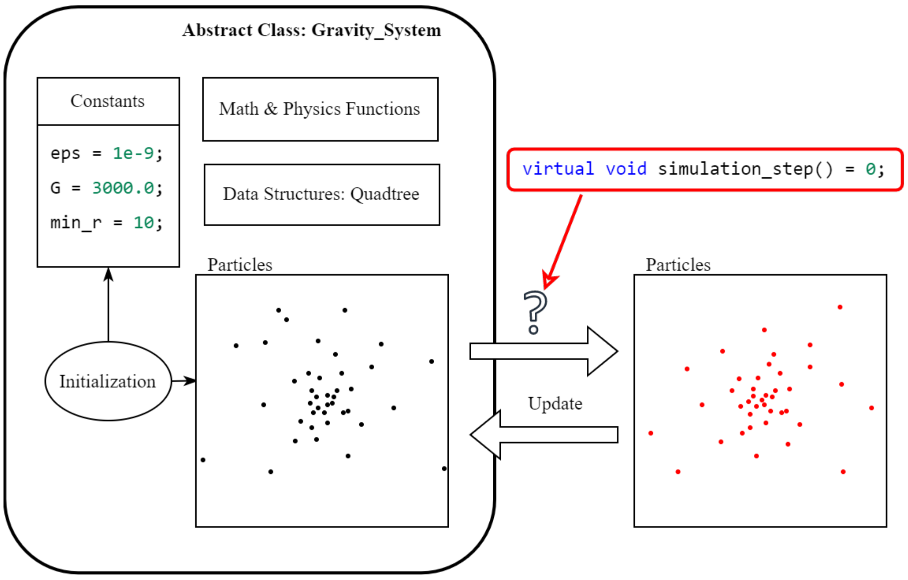
Date: November 17, 2022

## 1 Introduction

The N-body problem, which started out as a physics problem, consists of $6N$ differential equations. Since this system of equations, mathematically, has no analytic solution, many times we use iterative methods to solve to solve it. Many numerical methods have also been invented to solve this type of numerical integrals to obtain a more accurate numerical solution. In this project, I will use leapfrog integration method to solve this numerical integral, and I have implemented four different parallel methods to speed up the solution process. (CUDA, MPI, Pthread, OpenMP and hybrid method using MPI with OpenMP) And compare their advantages and disadvantages

## 2 Method

### 2.1 Basic Framework



```
eps = 1e-9;
G = 3000.0;
min_r = 10;
```

```
virtual void simulation_step() = 0;
```

Figure 1: Framework

The mathematical and physical functions used in both serial and parallel programs are the same, because in this simulation only the acceleration of each particle needs to be recalculated after each state as well as checking for collisions. So based on this, we can create an abstract class. Inside it we first encapsulate the relevant constants and formulas.

Among the encapsulated functions, the most important formulas are gravity formula:

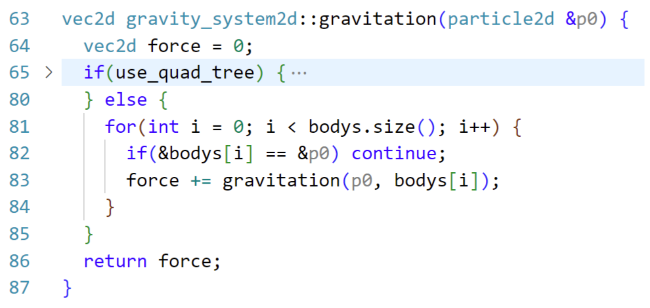
$$F_{i,j} = \frac{Gm_1m_2}{\|\mathbf{x}_1 - \mathbf{x}_2\|^2} \tag{1}$$

and formulas to solve elastic collision:

$$\begin{cases} \mathbf{v}_1' &= \mathbf{v}_1 - \frac{m_2}{m_1+m_2}\frac{\langle \mathbf{v}_1-\mathbf{v}_2, \mathbf{x}_1-\mathbf{x}_2\rangle}{\|\mathbf{x}_1-\mathbf{x}_2\|^2}(\mathbf{x}_1-\mathbf{x}_2) \\ \mathbf{v}_2' &= \mathbf{v}_2 - \frac{m_1}{m_1+m_2}\frac{\langle \mathbf{v}_2-\mathbf{v}_1, \mathbf{x}_2-\mathbf{x}_1\rangle}{\|\mathbf{x}_2-\mathbf{x}_1\|^2}(\mathbf{x}_2-\mathbf{x}_1) \end{cases} \tag{2}$$

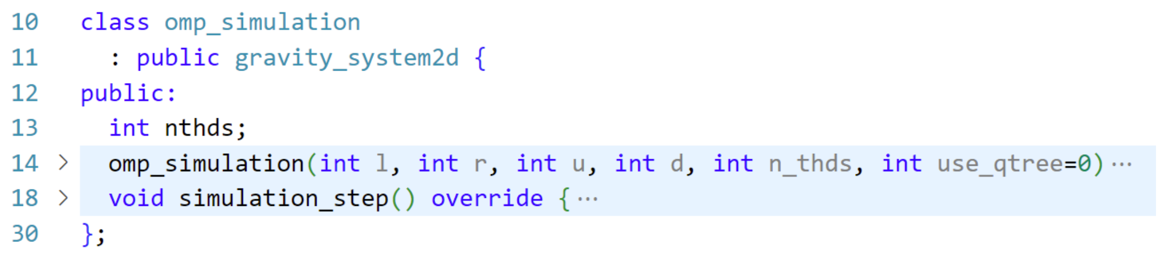
And leapfrog integration, which is used to iterate to the next step:

$$\begin{aligned} x_{i+1} &= x_i + v_i\Delta t + \frac{1}{2}a_i\Delta t^2 \\ v_{i+1} &= v_i + \frac{1}{2}(a_i + a_{i+1})\Delta t \end{aligned} \tag{3}$$



```
vec2d gravity_system2d::gravitation(particle2d &p0) {
  vec2d force = 0;
  if(use_quad_tree) {⋯
  } else {
    for(int i = 0; i < bodys.size(); i++) {
      if(&bodys[i] == &p0) continue;
      force += gravitation(p0, bodys[i]);
    }
  }
  return force;
}
```

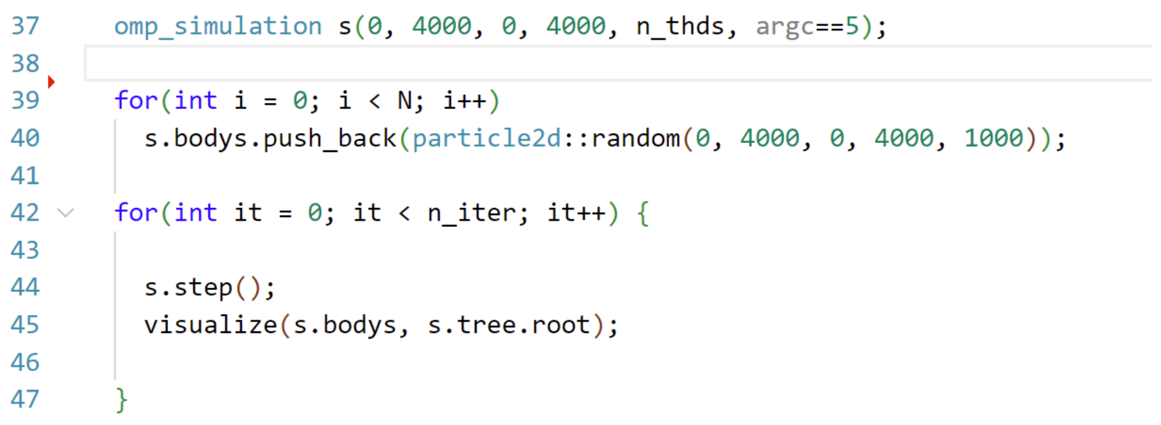
In the process of implementing a parallel program, we only need to use a subclass to inherit this abstract class and implement the `simulation_step` method to complete the acceleration update, because only this part is what we need to innovate and optimize in parallel, all other parts can be done serially in $O(N)$ time, where $N$ is the number of moving particles



```
class omp_simulation
  : public gravity_system2d {
public:
  int nthds;
  omp_simulation(int l, int r, int u, int d, int n_thds, int use_qtree=0)⋯
  void simulation_step() override {⋯
};
```

Figure 2: Example of a newly implemented class by inheritate the abstract framefork class

Once this has been implemented, we can complete the simulation by simply calling the `step` function:



```
omp_simulation s(0, 4000, 0, 4000, n_thds, argc==5);

for(int i = 0; i < N; i++)
  s.bodys.push_back(particle2d::random(0, 4000, 0, 4000, 1000));

for(int it = 0; it < n_iter; it++) {

  s.step();
  visualize(s.bodys, s.tree.root);

}
```

Figure 3: Example of simulation

## 2.2 Sequential Implementaion

The implementation of the sequential algorithm is very trivial. For each particle, the acceleration of the particle in its current state is calculated by simply computing the combined force of all the other particles. When the acceleration of all $N$ particles is calculated, the velocity and position are updated.

## 2.3 Parallization With MPI

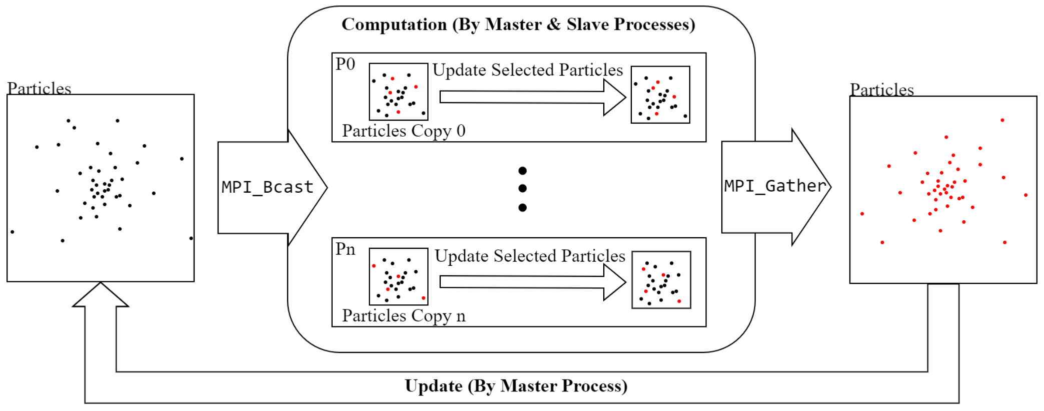

Computation (By Master & Slave Processes)
Particles
P0
Update Selected Particles
Particles Copy 0
MPI_Bcast
MPI_Gather
Pn
Update Selected Particles
Particles Copy n
Particles
Update (By Master Process)

Figure 4: Flowchart of MPI multi-processes implementation

### Design

From the above observation we find that the order of different particle calculations does not affect the final result of the calculation when updating the acceleration at one time (aka embarrassing parallization). So we use `MPI_Bcast` to send the state information of all the particles to all the other processors first. Each processor gets all the data and then processes the part that it needs to compute (usually an interval), and when each processor has finished its task, it calls `MPI_Gather` to send all the answers to the master processor. The main processor aggregates the states of all the particles and then updates the states once, using acceleration to update their positions, with a time complexity of $O(n)$, where $N$ is the number of particles.

**Implementation** During the implementation, in order to reduce the message passing time during processing, we define a new `MPI_Datatype` to store the information about particles.

```
int stat_blocklen[5] = {1, 1, 1, 1, 1};
MPI_Datatype stat_types[5] = {MPI_DOUBLE, MPI_DOUBLE, MPI_DOUBLE, MPI_DOUBLE, MPI_DOUBLE};
MPI_Aint stat_offsets[5] = {
  offsetof(particle2d_stat_t, m),
  offsetof(particle2d_stat_t, dx), offsetof(particle2d_stat_t, dy),
  offsetof(particle2d_stat_t, vx), offsetof(particle2d_stat_t, vy) };
MPI_Type_create_struct(5, stat_blocklen, stat_offsets, stat_types, &MPI_PARTICLE2D_STAT);
MPI_Type_commit(&MPI_PARTICLE2D_STAT);
```

Figure 5: The detailed code to create MPI_Datatype

## 2.4 Parallization With Multithreads (Pthread & OpenMP)

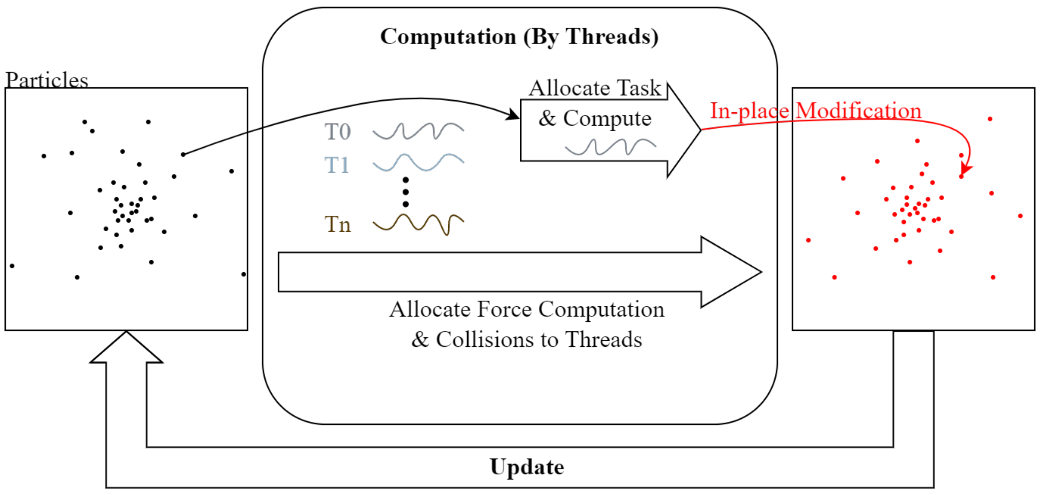


Figure 6: Flowchart of multi-threading implementation

The method of parallelizing the simulation using multi-threads is very similar to parallelizing with muti-processes, except that each computation uses a thread instead of a process, and a particle is assigned to an available thread, which then computes the acceleration of the particle. Once all the particle information has been computed, the state is updated. The whole process is very easy to implement, , and the code does not exceed 100 lines. For the use of multi-threads, I have implemented both OpenMP and Pthread versions.

## 2.5 Parallization With GPU (CUDA)

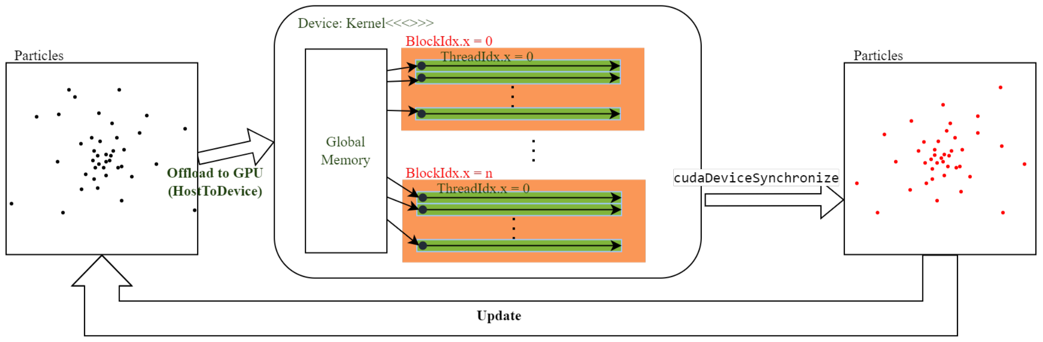


Figure 7: Flowchart of CUDA implementation

In each iteration, we need to transfer the state information of the particles to the gpu, implement the computation function on the gpu, call the computation, wait for the gpu to finish all the computations on the host, and then copy back the results of the computation from the gpu to update the current state.

CUDA allows us to use some functions to call the gpu's resources. Specifically, we need to define a kernel function that performs a multi-threaded computation on the gpu based on the data in CUDA memory. Each thread calculates the acceleration and other state information of a particle, and when all particles are updated, the original data is copied back from the GPU to the CPU.

## 2.6 Bonus: Parallization With MPI+OpenMP

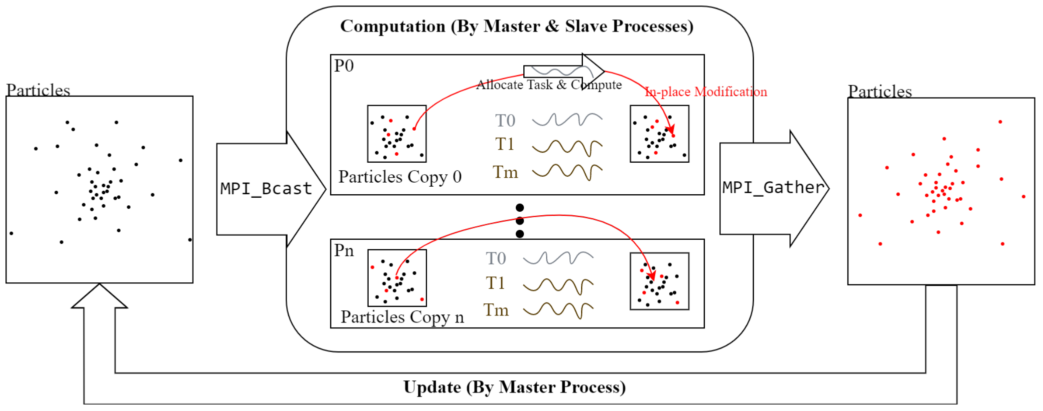


Figure 8: Flowchart of MPI+OpenMP Implementaion

In the MPI version, only multi-processes are used to compute the N-body problem in parallel. In fact, this algorithm still has room for optimization, because it is possible to use multi-threaded methods to increase concurrency in a single process to achieve a certain level of speedup.

So it is possible to use OpenMP on the basis of MPI to increase concurrency on a per-process basis using a multi-threaded approach. The specific design is shown in the figure above, where after `MPI_Bcast` copies the particle information, the API of openmp is called to update the particle state information in a multi-threaded manner.

## 2.7 Optimization Using Quad Tree (Barnes-Hut Simulation)

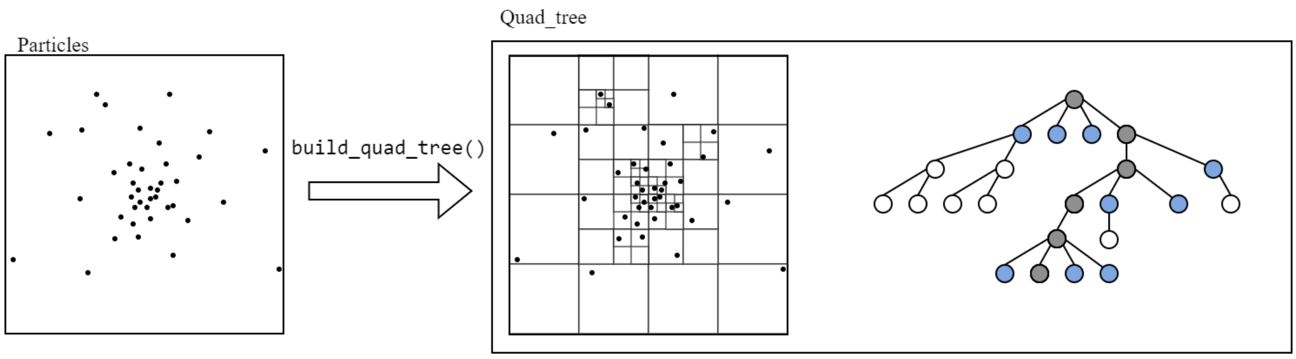


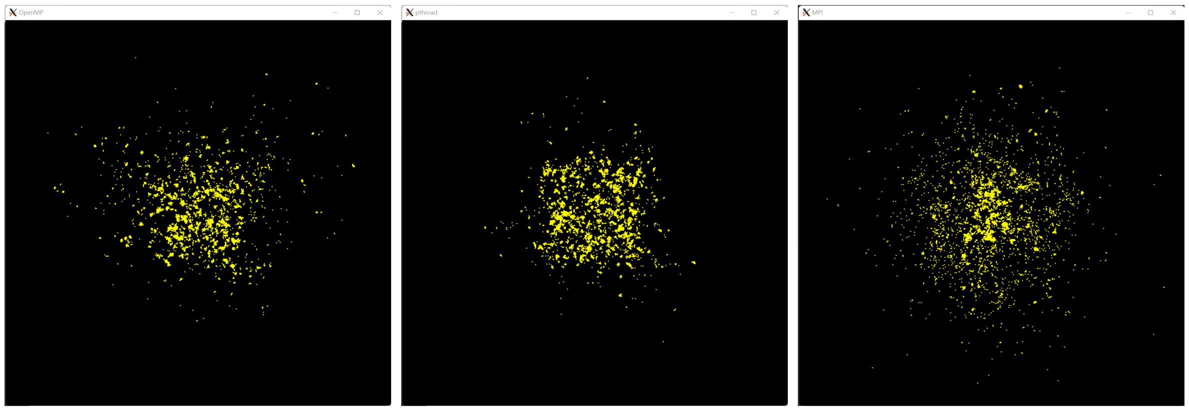
Figure 9: Quad-tree method in partitioning the particles by there spatial information

In previous implementations, we directly enumerate and compute the force between every pair of particles. Since there are $O(N^2)$ pairs of this relationship in a N-body system, the computation costs $O(N^2)$ time in one step. By adopting Barnes-Hut simulation, we use a quad-tree to divide space and store the information about particles, thus can recieve a $O(N\log(N))$ time complexity by trading off the accuracy. The main idea of this method is to use the attraction of center of mass to substitute the overall attractions by the particles in the subtree. When calculating the gravitational force, it is only necessary to check every neighbouring node in the path to the root. For example, in fig. 9, assume the queried particle is stored in the deepest gray node of the quadtree, then every node in blue color should be counted into the final answer.

Additionally, the task of collision checking costs also $O(\log(N))$ time. Since we only need to check the 8 neighbouring blocks to find if there is a collision.

## 3 Results & Analysis

### 3.1 GUI Results

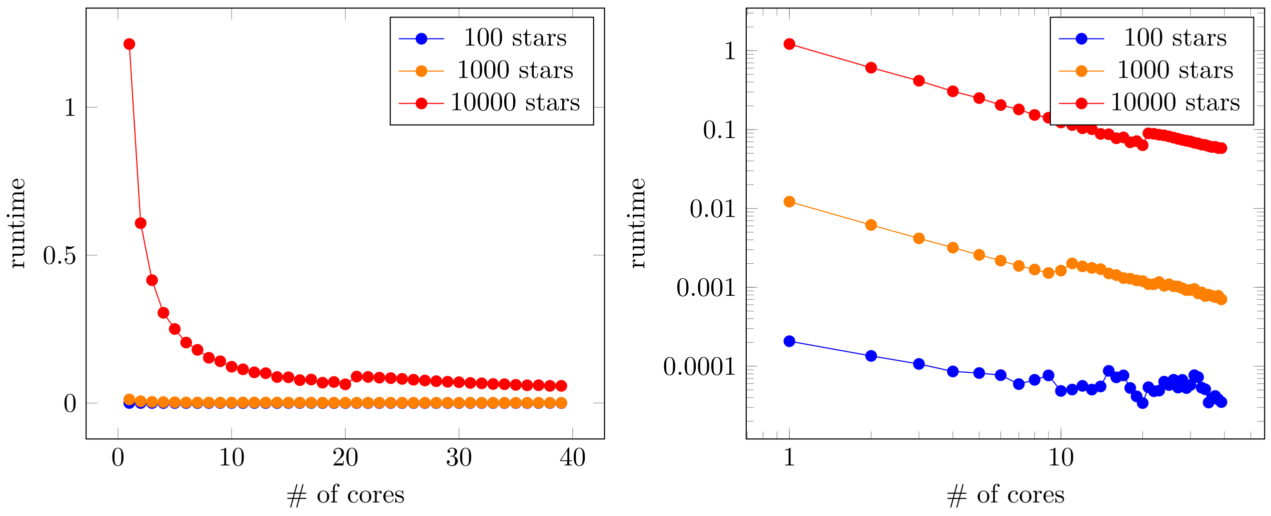


(a) (b) (c)

Figure 10: GUI Demo: yellow points are particles

### 3.2 Performance of MPI Implementation

#### Performance Under Different Number of Cores



(a) Normal (b) Logarithmic

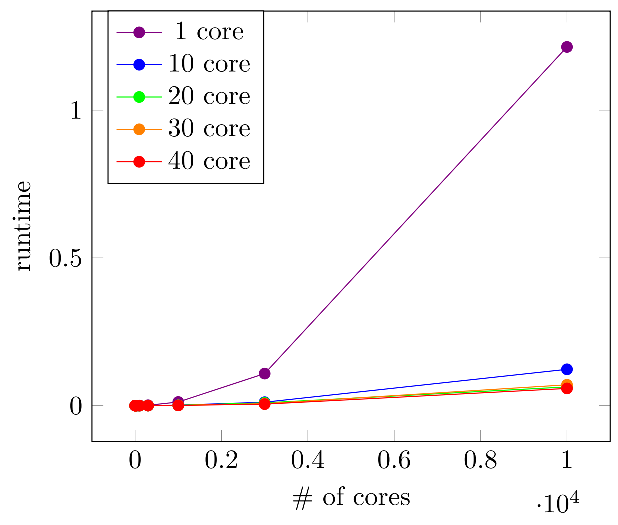
Figure 11: MPI program runtime v.s. # core

| # cores | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 100 | 0.00021 | 0.00013 | 0.00011 | 0.00009 | 0.00008 | 0.00008 | 0.00006 | 0.00007 | 0.00008 | 0.00005 |
| 1000 | 0.01218 | 0.00617 | 0.00417 | 0.00318 | 0.00258 | 0.00218 | 0.00187 | 0.00168 | 0.00152 | 0.00163 |
| 10000 | 1.21392 | 0.60824 | 0.41548 | 0.30522 | 0.25041 | 0.20455 | 0.17987 | 0.15309 | 0.14128 | 0.12284 |
| # cores | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 100 | 0.00005 | 0.00006 | 0.00005 | 0.00006 | 0.00009 | 0.00007 | 0.00008 | 0.00005 | 0.00004 | 0.00003 |
| 1000 | 0.00201 | 0.00184 | 0.00176 | 0.00170 | 0.00150 | 0.00143 | 0.00131 | 0.00129 | 0.00122 | 0.00120 |
| 10000 | 0.11412 | 0.10383 | 0.10096 | 0.08801 | 0.08713 | 0.07751 | 0.07951 | 0.06886 | 0.07129 | 0.06323 |
| # cores | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 100 | 0.00005 | 0.00005 | 0.00005 | 0.00006 | 0.00006 | 0.00007 | 0.00005 | 0.00007 | 0.00005 | 0.00006 |
| 1000 | 0.00109 | 0.00109 | 0.00116 | 0.00105 | 0.00109 | 0.00103 | 0.00102 | 0.00097 | 0.00092 | 0.00093 |
| 10000 | 0.08953 | 0.08828 | 0.08592 | 0.08438 | 0.08172 | 0.07894 | 0.07626 | 0.07390 | 0.07202 | 0.07044 |
| # cores | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 |
| 100 | 0.00008 | 0.00007 | 0.00005 | 0.00005 | 0.00003 | 0.00004 | 0.00004 | 0.00004 | 0.00004 | 0.00466 |
| 1000 | 0.00095 | 0.00084 | 0.00086 | 0.00078 | 0.00081 | 0.00078 | 0.00075 | 0.00078 | 0.00070 | 0.00131 |
| 10000 | 0.06799 | 0.06668 | 0.06446 | 0.06370 | 0.06162 | 0.06005 | 0.06027 | 0.05819 | 0.05816 | 0.05641 |

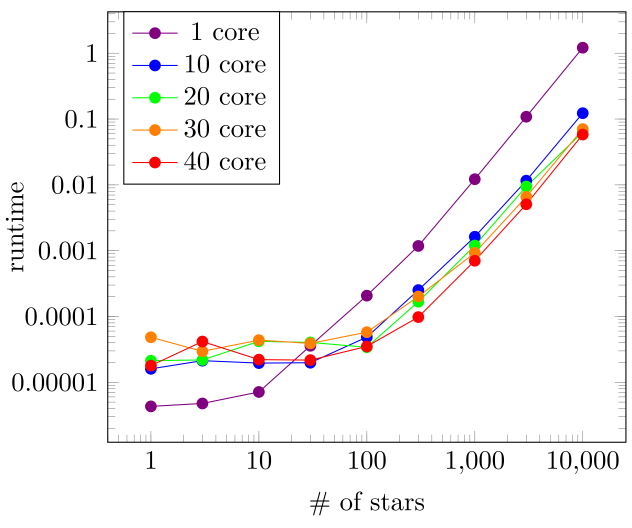
Table 1: runtime of MPI program (Part 1)

From the left image, we can clearly see that for the same problem size, when we increase the number of mpi processes, the time used will gradually decrease, and when the number of processes is not very large, it will roughly take the shape of an inverse proportional function, especially when the problem size is large, this law is particularly obvious. As we can see from the picture on the right, when we increase the number of processes, the whole image shows a straight line with a downward slope less than zero. This confirms our suspicion that when increasing the number of processes, the whole image is almost similar to an inverse proportional function. All of these illustrate that the MPI program does not obviously suffer from latency increased by additional processors.

**Performance Under Different Problem Sizes**



(a) Normal



(b) Logarithmic

Figure 12: MPI program runtime v.s. problem size

| # of stars | 1 | 3 | 10 | 30 | 100 | 300 | 1000 | 3000 | 10000 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.00000 | 0.00000 | 0.00001 | 0.00004 | 0.00021 | 0.00118 | 0.01218 | 0.10846 | 1.21392 |
| 10 | 0.00002 | 0.00002 | 0.00002 | 0.00002 | 0.00005 | 0.00025 | 0.00163 | 0.01158 | 0.12284 |
| 20 | 0.00002 | 0.00002 | 0.00004 | 0.00004 | 0.00003 | 0.00017 | 0.00120 | 0.00943 | 0.06323 |
| 30 | 0.00005 | 0.00003 | 0.00004 | 0.00004 | 0.00006 | 0.00020 | 0.00093 | 0.00658 | 0.07044 |
| 40 | 0.00002 | 0.00091 | 0.00002 | 0.00002 | 0.00466 | 0.00010 | 0.00131 | 0.00505 | 0.05641 |

Table 2: runtime of MPI program (Part 2)

As we can see from the graph on the left, the computation takes more and more time as the problem size increases, and using multiple cores in parallel at the same time can reduce the runtime significantly. However, we can still see from the analysis on the right that in the logarithmic coordinate system, regardless of the configuration of the mpi program, the image tends to be a sloped straight line with a fixed slope as the problem size increases, which shows that although the program runs shorter in parallel, we still do not reduce its time complexity.

## 3.3 Performance of Pthread Implementation

### Performance Under Different Number of Threads

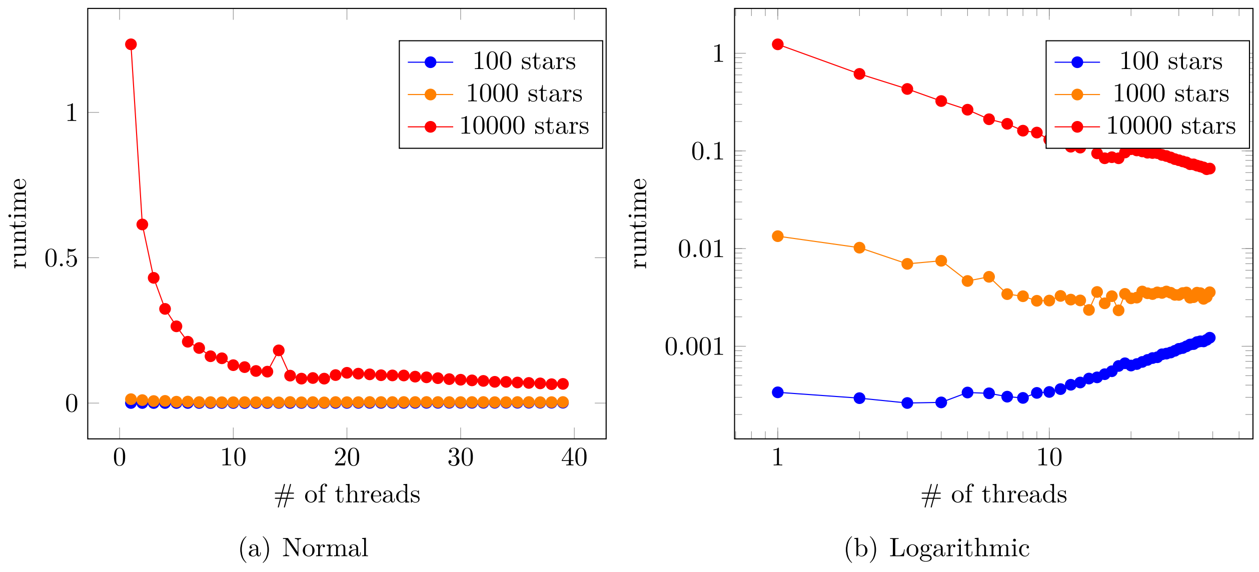


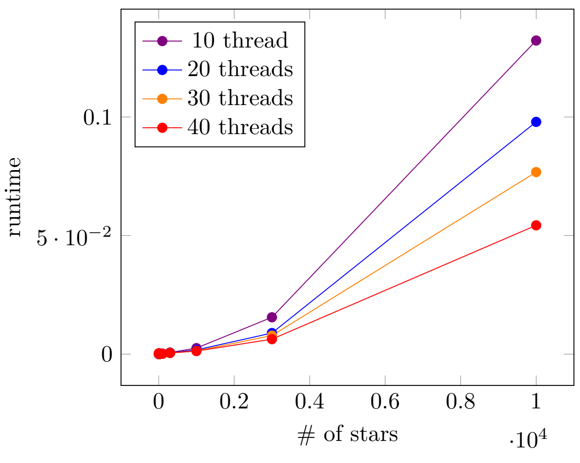
Figure 13: Pthread program runtime v.s. # threads

| # threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 100 | 0.00034 | 0.00029 | 0.00026 | 0.00027 | 0.00034 | 0.00033 | 0.00030 | 0.00030 | 0.00033 | 0.00034 |
| 1000 | 0.01340 | 0.01021 | 0.00698 | 0.00750 | 0.00466 | 0.00514 | 0.00342 | 0.00325 | 0.00292 | 0.00294 |
| 10000 | 1.23433 | 0.61448 | 0.43082 | 0.32402 | 0.26411 | 0.21101 | 0.18941 | 0.16114 | 0.15411 | 0.13041 |
| # threads | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 100 | 0.00036 | 0.00040 | 0.00043 | 0.00047 | 0.00048 | 0.00052 | 0.00055 | 0.00063 | 0.00067 | 0.00063 |
| 1000 | 0.00327 | 0.00300 | 0.00295 | 0.00235 | 0.00360 | 0.00275 | 0.00325 | 0.00233 | 0.00344 | 0.00310 |
| 10000 | 0.12368 | 0.11042 | 0.10809 | 0.18131 | 0.09441 | 0.08409 | 0.08620 | 0.08407 | 0.09631 | 0.10414 |
| # threads | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 100 | 0.00066 | 0.00069 | 0.00072 | 0.00076 | 0.00077 | 0.00083 | 0.00084 | 0.00086 | 0.00090 | 0.00094 |
| 1000 | 0.00315 | 0.00363 | 0.00349 | 0.00342 | 0.00356 | 0.00351 | 0.00363 | 0.00353 | 0.00336 | 0.00335 |
| 10000 | 0.10117 | 0.09883 | 0.09583 | 0.09514 | 0.09491 | 0.09091 | 0.08865 | 0.08568 | 0.08240 | 0.08030 |
| # threads | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 |
| 100 | 0.00096 | 0.00100 | 0.00104 | 0.00105 | 0.00110 | 0.00112 | 0.00112 | 0.00116 | 0.00122 | 0.00125 |
| 1000 | 0.00351 | 0.00355 | 0.00314 | 0.00318 | 0.00353 | 0.00350 | 0.00306 | 0.00319 | 0.00357 | 0.00366 |
| 10000 | 0.07812 | 0.07625 | 0.07308 | 0.07282 | 0.07073 | 0.06926 | 0.06775 | 0.06496 | 0.06595 | 0.06768 |

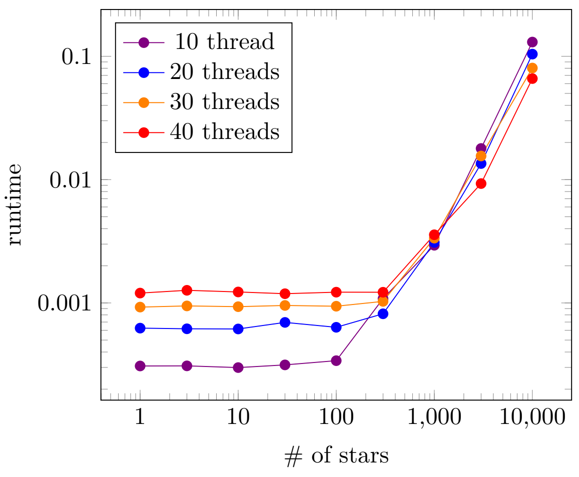
Table 3: runtime of pthread program (Part 1)

The image on the left illustrates that when the problem is large, the time spent on parallelizing the simulation using Pthread decreases as more threads are called. And the image is roughly in the form of an inverse proportional function. But from the logarithmic coordinate system on the right, we can see that this time reduction is not concluded for all problem scopes. For example, when there are only 100 particles in the system, as the number of threads exceeds a certain value, the time used for the computation no longer decreases, but increases. There are many possible reasons for this phenomenon, one of which could be that when adding cores, the communication time between processes becomes longer, resulting in an overall performance degradation.

### Performance Under Different Problem Sizes



(a)



(b)

Figure 14: pthread program runtime v.s. problem size

| # stars | 1 | 3 | 10 | 30 | 100 | 300 | 1000 | 3000 | 10000 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.00005 | 0.00006 | 0.00005 | 0.00008 | 0.00034 | 0.00139 | 0.01340 | 0.11237 | 1.23433 |
| 10 | 0.00031 | 0.00031 | 0.00030 | 0.00031 | 0.00034 | 0.00108 | 0.00294 | 0.01789 | 0.13041 |
| 20 | 0.00062 | 0.00062 | 0.00062 | 0.00070 | 0.00063 | 0.00082 | 0.00310 | 0.01350 | 0.10414 |
| 30 | 0.00093 | 0.00095 | 0.00093 | 0.00095 | 0.00094 | 0.00103 | 0.00335 | 0.01554 | 0.08030 |
| 40 | 0.00126 | 0.00126 | 0.00124 | 0.00152 | 0.00125 | 0.00127 | 0.00366 | 0.00931 | 0.06768 |

Table 4: pthread program runtime (part 2)

Then, the performance of the Pthread program varies at different problem sizes. Broadly speaking, when the problem size increases, the running time becomes longer, and as we can see from the image on the right, when the problem size increases to a certain level. Increasing the number of threads reduces the running time, but does not change the algorithm time complexity.

It is interesting to note that in the right graph, when the problem size is small, we can see that the running time tends to be a constant and does not increase as the problem size increases. And the more threads, the longer the algorithm takes to run, which indicates that the exchange of information between Pthread threads is more time consuming and affects the overall running efficiency.

## 3.4 Performance of OpenMP Implementation

### Performance Under Different Number of Threads

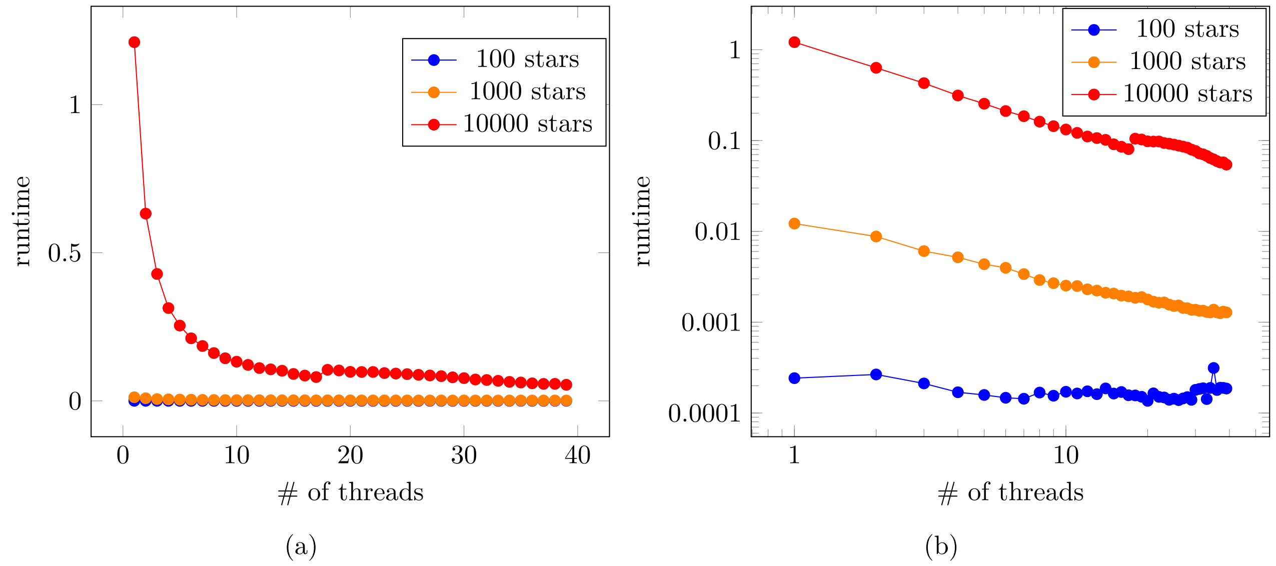


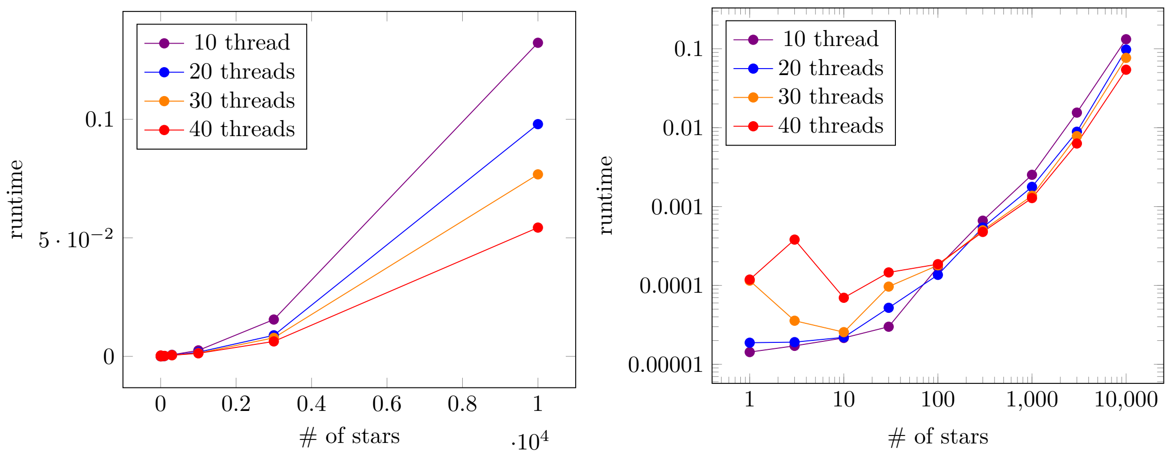
Figure 15: OpenMP program runtime v.s. # threads

| # threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 100 | 0.00024 | 0.00027 | 0.00021 | 0.00017 | 0.00016 | 0.00015 | 0.00014 | 0.00017 | 0.00016 | 0.00017 |
| 1000 | 0.01218 | 0.00878 | 0.00606 | 0.00518 | 0.00435 | 0.00397 | 0.00338 | 0.00291 | 0.00269 | 0.00253 |
| 10000 | 1.21067 | 0.63205 | 0.42834 | 0.31323 | 0.25417 | 0.21118 | 0.18524 | 0.16142 | 0.14386 | 0.13226 |
| # threads | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 100 | 0.00016 | 0.00017 | 0.00016 | 0.00019 | 0.00016 | 0.00017 | 0.00016 | 0.00016 | 0.00015 | 0.00014 |
| 1000 | 0.00250 | 0.00230 | 0.00223 | 0.00211 | 0.00207 | 0.00196 | 0.00192 | 0.00186 | 0.00189 | 0.00178 |
| 10000 | 0.12142 | 0.11074 | 0.10643 | 0.10171 | 0.09086 | 0.08542 | 0.08041 | 0.10484 | 0.10264 | 0.09795 |
| # threads | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 100 | 0.00016 | 0.00015 | 0.00015 | 0.00014 | 0.00014 | 0.00014 | 0.00014 | 0.00015 | 0.00014 | 0.00018 |
| 1000 | 0.00168 | 0.00164 | 0.00165 | 0.00156 | 0.00151 | 0.00153 | 0.00143 | 0.00142 | 0.00137 | 0.00137 |
| 10000 | 0.09751 | 0.09764 | 0.09389 | 0.09220 | 0.09029 | 0.08792 | 0.08576 | 0.08350 | 0.07947 | 0.07677 |
| # threads | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 |
| 100 | 0.00018 | 0.00019 | 0.00014 | 0.00019 | 0.00031 | 0.00018 | 0.00019 | 0.00019 | 0.00019 | 0.00039 |
| 1000 | 0.00134 | 0.00134 | 0.00129 | 0.00128 | 0.00138 | 0.00128 | 0.00126 | 0.00131 | 0.00128 | 0.00142 |
| 10000 | 0.07214 | 0.07048 | 0.06774 | 0.06416 | 0.06200 | 0.05925 | 0.05732 | 0.05748 | 0.05432 | 0.05240 |

Table 5: openmp program runtime (Part 1)

When using OpenMP to parallelize the computation of N-body simulation, the algorithm takes less time as the number of calling processes increases. Also, in the logarithmic coordinate system on the right, we can see that even when the problem size is small, the algorithm running time still decreases with the number of processes, which is related to the fact that OpenMP uses less information passing between processes in its implementation.

**Performance Under Different Problem Sizes**



(a) (b)

Figure 16: OpenMP program runtime v.s. problem size

| | 1 | 3 | 10 | 30 | 100 | 300 | 1000 | 3000 | 10000 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.00000 | 0.00001 | 0.00001 | 0.00004 | 0.00024 | 0.00140 | 0.01218 | 0.10846 | 1.21067 |
| 10 | 0.00001 | 0.00002 | 0.00002 | 0.00003 | 0.00017 | 0.00066 | 0.00253 | 0.01553 | 0.13226 |
| 20 | 0.00002 | 0.00002 | 0.00002 | 0.00005 | 0.00014 | 0.00054 | 0.00178 | 0.00889 | 0.09795 |
| 30 | 0.00012 | 0.00004 | 0.00003 | 0.00010 | 0.00018 | 0.00050 | 0.00137 | 0.00781 | 0.07677 |
| 40 | 0.00028 | 0.00006 | 0.00017 | 0.00020 | 0.00039 | 0.00054 | 0.00142 | 0.00664 | 0.05240 |

Table 6: openmp program runtime (part 2)

Moreover, when we analyze the relationship between runtime and problem size, we can also see that the runtime does get larger as the problem size increases, and similar to the previous parallel methods, the time complexity of the algorithm remains unchanged when the problem size is large, even though the addition of new threads reduces the runtime. One difference from the previous implementations is that the OpenMP program no longer uses a constant time when the problem is small. This is most likely related to the fact that the OpenMP program does not perform a lot of message passing during the implementation.

## 3.5 Performance of GPU Offloading (GPU)

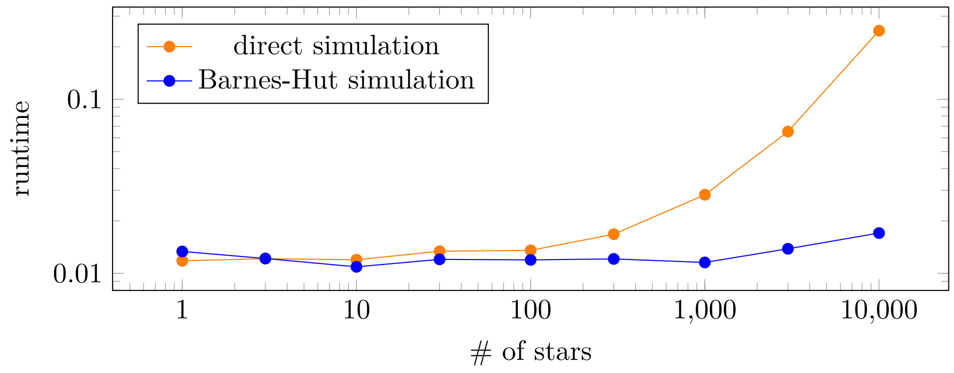


Figure 17: This figure illustrates the relationship between problem size & runtime in CUDA program, the y-axis is displayed in logarithmic mode, it is obvious that at 10,000, there is a surge in runtime

| Method | 1 | 3 | 10 | 30 | 100 | 300 | 1000 | 3000 | 10000 | 30000 |
|---|---|---|---|---|---|---|---|---|---|---|
| Direct | 0.01180 | 0.01213 | 0.01196 | 0.01339 | 0.01355 | 0.01677 | 0.02826 | 0.06518 | 0.24764 | 1.93870 |
| Barnes-Hut | 0.01335 | 0.01216 | 0.01089 | 0.01202 | 0.01194 | 0.01209 | 0.01155 | 0.01383 | 0.01703 | 0.03224 |

## 3.6 Comparison between MPI w&wo OpenMP(Bonus)

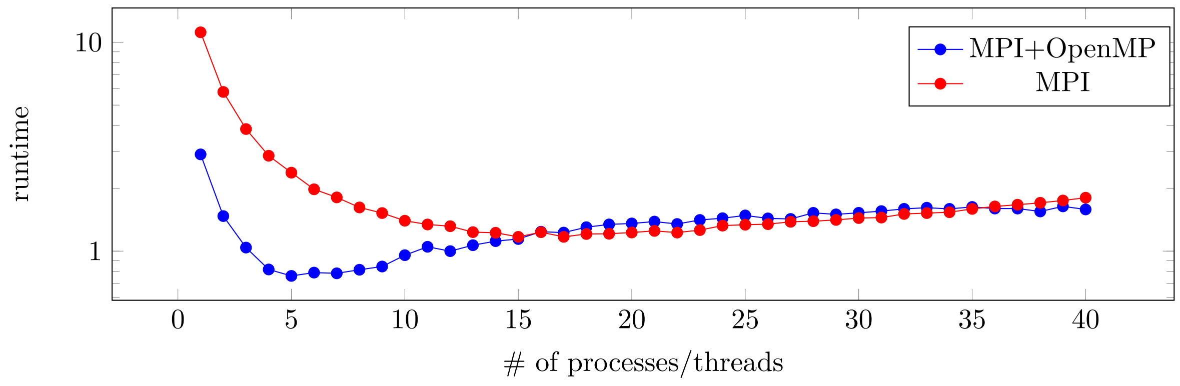


Figure 18: $N = 30,000$

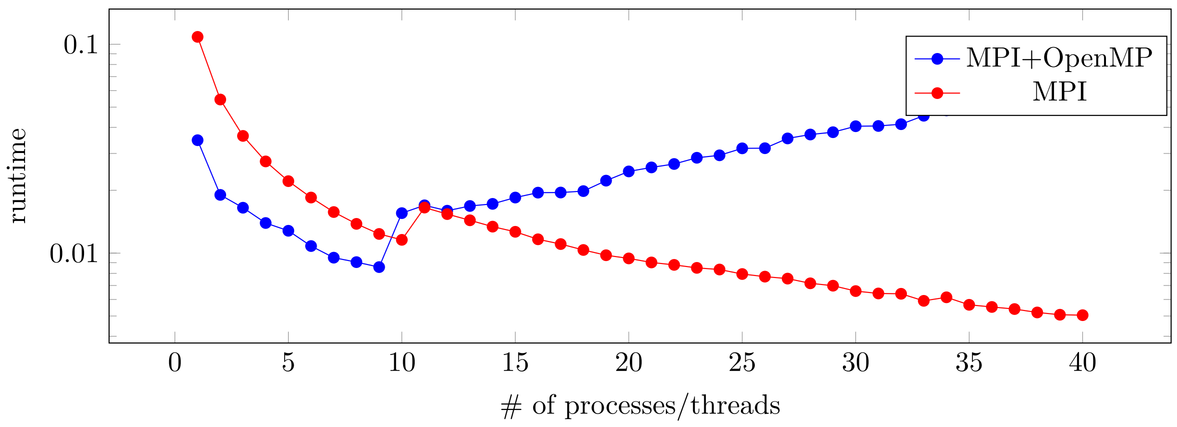


Figure 19: $N = 3,000$

From the two figures above, it is shown that, MPI with OpenMP is more suitable for large-scale simulation. Most of its advantages are when processor number is small. So it can be infer that this method, though has a higher therotical efficiency, it suffers a lot in communication.

## 3.7 Some Comparison On Different Parallel Implementaions

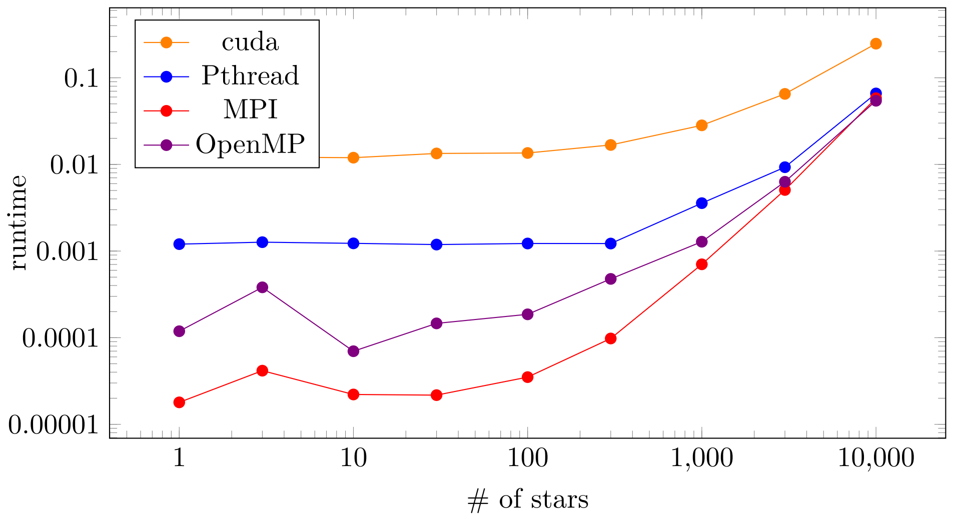


Figure 20: This graph shows the running time versus problem size for different parallel methods with the 40 processors/threads (except CUDA)

From this picture we can see that when the problem size is large, the processing time is similar at 40 cores, regardless of the running method, except at CUDA. At the same time, although their running times are different, the final slope tends to be the same as the problem increases, which shows that they have the same complexity of time.

When the problem size is small, the running times of different implementations vary widely, where CUDA has the longest running time and mpi has the shortest running time. These differences gradually decrease after the problem size reaches a certain threshold.

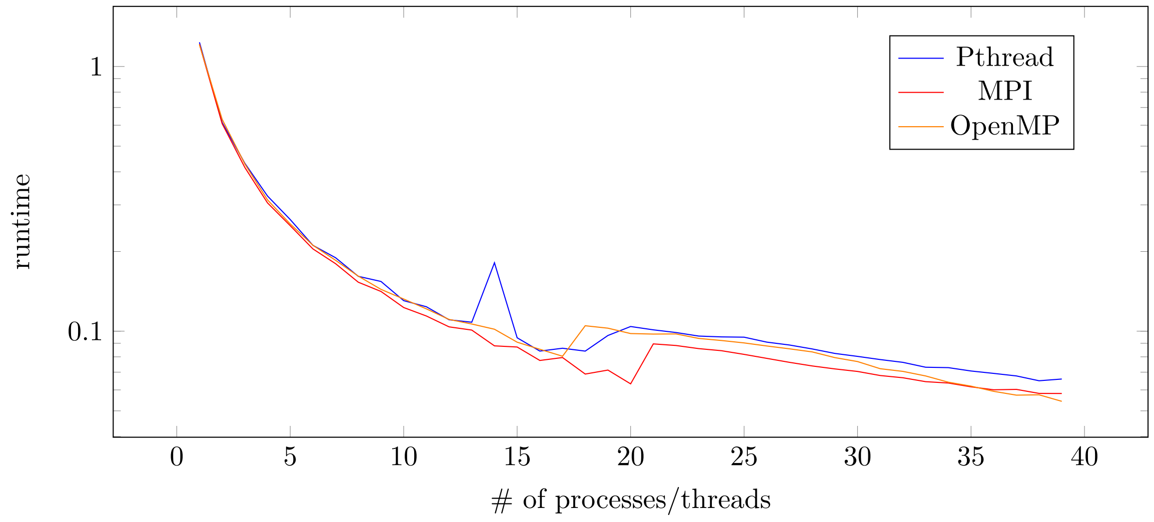


Figure 21: This figure shows the running time of different parallel programs versus the number of computational processors invoked for a data size of 1,000, and it can be seen that there is no significant difference in the running time of different parallel programs for larger data ranges.

## 3.8 Comparison on Sequential & Parallel Implementaions

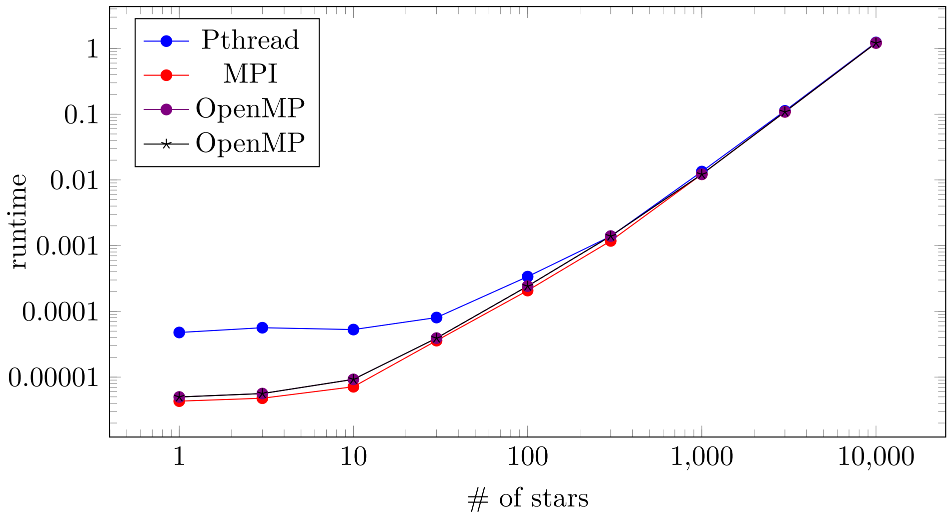
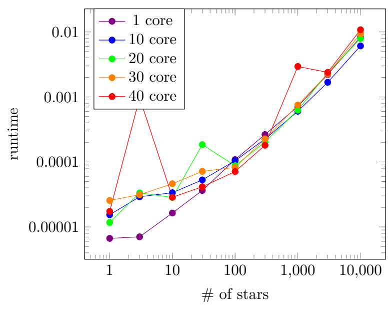


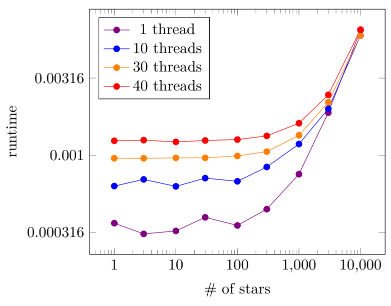
Figure 22: This graph shows the running time of the sequential program, and all parallel-style programs at a single processor as a function of problem size

As can be seen from this graph, the parallel-style program runs in a similar time to the sequential-style program under single processor conditions, except for Pthread, which runs longer with small data. This further provides a basis for the performance improvement of parallel algorithms.

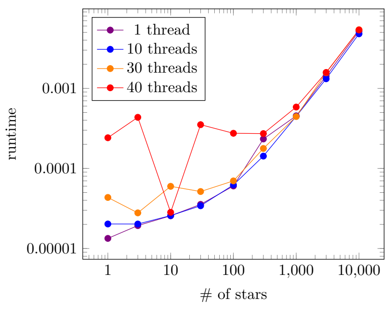
### 3.9 Discussion On Barnes-Hut Simulation



(a) MPI



(b) Pthread



(c) OpenMP

Figure 23: Selected Plots of Barnes-Hut Simulation

Since Barnes-Hut Simulation provides a time complexity of $O(N \log(N))$, the problem scale in experiments above seems to be too small for this method to show advantage in parallelizing. Due to the time limit, I did not conduct further experiments but guess that since $O(\log(N))$ is much like a large constant in sequential programs, the Barnes-Hut simulation cannot benefit from parallelizing until $\log(N)$ increases to a large number ranther than a constant.

## 4 Conclusion

After implementing different parallel-style algorithms and comparing them, I have come to three basic conclusions: the first is that parallel-style algorithms do have an advantage in time compared to serial methods. The second point is that for general variational algorithms, in multi-body problems, they are able to grow with time and get faster as the number of processors increases. The third point is that for the N-body problem, optimizing its time complexity is far more efficient than parallelism.

I personally benefit a lot in this project, for I gained a lot experience in handling parallization and analyzing data.

# Appendix A. How to Compile & Run the Program

## Compile:

The compilation can be done by calling `make`: If you want to compile all implementaions with gui:

```
$make gui
```

If you want to compile all implementaions without gui:

```
$make no_gui
```

If compiling a specific implementation:

1. CUDA (with GUI): `$make cudag`
2. CUDA (without GUI): `$make cuda`
3. MPI (with GUI): `$make mpig`
4. MPI (without GUI): `$make mpi`
5. Pthread (with GUI): `$make pthreadg`
6. Pthread (without GUI): `$make pthread`
7. OpenMP (with GUI): `$make openmpg`
8. OpenMP (without GUI): `$make openmp`
9. Sequential (with GUI): `$make seqg`
10. Sequential (without GUI): `$make seq`
11. MPI + OpenMP (with GUI): `$make bonusg`
12. MPI + OpenMP (without GUI): `$make bonus`

## Run:

For all the compiled executables above, running directly will start simulation with $N = 200$, $iter = 1000$, and $\#proc = 1$ (if this option available). For example, `$./cuda` will run cuda program without gui and start from $N = 200$, $n\_iter = 1000$.

Additionally, you can choose to use the following commands to run with custom options:

**Run with GUI**

| | |
|---|---|
| CUDA | `./cudag <# bodys> <# iters>` |
| CUDA (using Barnes-Hut) | `./cudag <# bodys> <# iters> 1` |
| MPI | `mpirun -np <# procs> ./mpig <# bodys> <# iters>` |
| MPI (with Barnes-Hut) | `mpirun -np <# procs> ./mpig <# bodys> <# iters> 1` |
| MPI + OpenMP | `mpirun -np <# procs> ./bonusg <# bodys> <# iters>` |
| MPI + OpenMP (with Barnes-Hut) | `mpirun -np <# procs> ./bonusg <# bodys> <# iters> 1` |
| Pthread | `./pthreadg <# bodys> <# iters> <#threads>` |
| Pthread (with Barnes-Hut) | `./pthreadg <# bodys> <# iters> <#threads> 1` |
| OpenMP | `./openmpg <# bodys> <# iters> <#threads>` |
| OpenMP (with Barnes-Hut) | `./openmpg <# bodys> <# iters> <#threads> 1` |
| Sequential | `./seqg <# bodys> <# iters>` |
| Sequential (with Barnes-Hut) | `./seqg <# bodys> <# iters> 1` |

**Run without GUI**

| | |
|---|---|
| CUDA | `./cuda <# bodys> <# iters>` |
| CUDA (using Barnes-Hut) | `./cuda <# bodys> <# iters> 1` |
| MPI | `mpirun -np <# procs> ./mpi <# bodys> <# iters>` |
| MPI (with Barnes-Hut) | `mpirun -np <# procs> ./mpi <# bodys> <# iters> 1` |
| MPI + OpenMP | `mpirun -np <# procs> ./bonus <# bodys> <# iters>` |
| MPI + OpenMP (with Barnes-Hut) | `mpirun -np <# procs> ./bonus <# bodys> <# iters> 1` |
| Pthread | `./pthread <# bodys> <# iters> <#threads>` |
| Pthread (with Barnes-Hut) | `./pthread <# bodys> <# iters> <#threads> 1` |
| OpenMP | `./openmp <# bodys> <# iters> <#threads>` |
| OpenMP (with Barnes-Hut) | `./openmp <# bodys> <# iters> <#threads> 1` |
| Sequential | `./seq <# bodys> <# iters>` |
| Sequential (with Barnes-Hut) | `./seq <# bodys> <# iters> 1` |