HIGH PERFORMANCE PROGRAMMING
ASSIGNMENT 4

Yuanjing Yang, Linjia Zhong

March 2025

1 Problem Description

In assignment 3, we completed a simulation program for the N-body problem. The program simulates and predicts the motion of particles by calculating the forces between them.

In the simulationStep() function: The outer loop iterates over all particles (N), and the inner loop iterates over the remaining particles (N-1 on average).This results in an overall time complexity of $O(N^2)$.

In the serial optimization of Assignment 3, we reduced the complexity from $O(N^2)$ to $O(N^2/2)$ by calculating the mutual forces symmetrically, and we further implemented serial optimizations by employing techniques such as removing loop invariants, loop optimization, reducing conditional branches and reducing loop overhead.

Serial Computation is a computing model in which tasks are executed sequentially, with each task starting only after the previous one has been completed. The instructions run in order on a single processing unit, without executing multiple tasks in parallel.Compared to Parallel Computation, Serial computation is generally less efficient and leads to a waste of computing resources, especially on modern computers that have multiple CPUs or multi-core processors.

To achieve higher computational efficiency, we applied parallel optimization to the code in Assignment 3 by using Pthreads and OpenMP, improving execution speed and making better use of computing resources.

2 Code Analyse

The program accept six input arguments as follows: [1]
`galsim N filename nsteps delta_t graphics n_threads`
where the input arguments have the following meaning:
N is the number of stars/particles to simulate
`filename` is the filename of the file to read the initial configuration from
nsteps is the number of timesteps δ_t is the timestep Δt
`graphics` is 1 or 0 meaning graphics on or off.
`n_threads` is the number of threads to use.

2.1 Data structure

Continued the Assignment 3, we used SOA in Pthread and Openmp to improve the memery access, storing the properties of particles in seperate arrary, which allowed vectorized operations. In pthread, we use thread data to store specific information in thread, like pointer to the particle data, and start and end indices for this thread, local acceleration arrays, and so on.

2.2 Code structure

We follow the instruction of Assignment 4, the code is divided into four parts: Reading input data(readfile), Parallel computation, Writing output file, Graphics.

2.3 Pthreads Optimization and Parallelization

Pthreads(POSIX threads) is used on multi-core machines and other shared memory computers. It uses a shared address space model based on threads, all threads have access to global data. Memory coherence handled by hardware but requires explicit synchronization and protection of shared variables from multiple updates. So Pthreads still requires manual division of work and synchronization between threads, making it a low-level programming model. [2]

We divided the work among multiple threads$NUM_THREADS$, and achieving load balancing by stride, and each thread processes every $NUM\_THREADS$-th particles. For avoiding race condition, we used local acceleration arrays, and add a reduction step turn local acceleration arrays to global arrays.

2.4 OpenMP Optimization and Parallelization

OpenMP is an open specification for multi-core machines and other shared memory computers, offering a high-level model based on threads with a shared address space, all threads have access to global data. OpenMP simplifies the process of parallelization by allowing the programmer to insert compiler directives to parallelize loops and computations automatically. [2]

We used dynamic scheduling to distribute iterations across threads(load balancing), the reduction ensures that the acceleration components are updated safely, and we use schedule(static) in updating velocity and position seperately due to it distribues the work evenly across threads.

3 Performance and Disscussion

According to Amdahl's Law: [3]

$$\text{Speedup}_{\text{parallel}}(f, n) = \frac{1}{(1-f) + \frac{f}{n}}$$

f is the fraction of the program that can be parallelized.
n is the number of processors or cores used in parallel computation.
$1 - f$ is the fraction of the program that must be executed sequentially.

When f is small, optimizations will have little effect.As S approaches infinity, speedup is bound by

$$\frac{1}{1-f}$$

So in multi-threads parallel computing, initially as the number of threads increases, execution time or resource consumption may decrease rapidly. But eventually, it will reach a bottleneck.

First we use Pthreads to parallelize our code and input different numbers of threads to find the best possible performance. We choose the input case **ellipse_N_03000**, with $\Delta t = 10^{-5}$ and 100 timesteps as an example.

The running time decreases rapidly as the number of threads increases from 1 to 3. Then, it maintains a steady decline when the number of threads is between 3 and 8. Execution time decreases as the number of threads increases and stabilizes when the number of threads approaches the number of available cores on the computer.

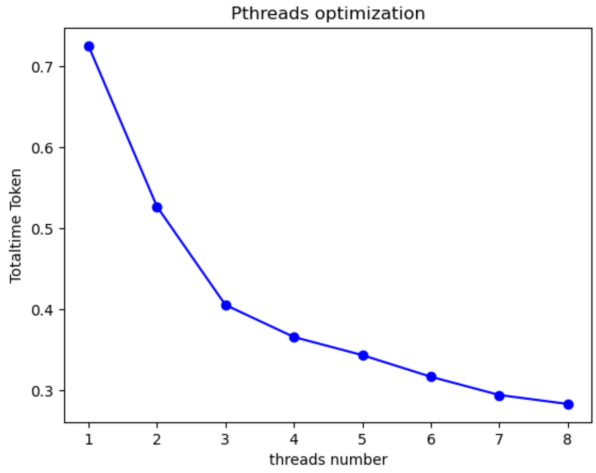

Figure 1: Pthreads Running time of in different numbers of threads.

The results of OpenMP parallel optimization are as follows. The running time trend of OpenMP optimization is similar to Pthreads. From 1 to 4 threads, the execution time decreases significantly, indicating a strong initial parallelization effect. From 5 to 8 threads, the rate of decrease slows down , and there is even a slight fluctuation between 6 and 7 threads, which may be due to resource contention (such as memory bandwidth) or overhead from task scheduling.

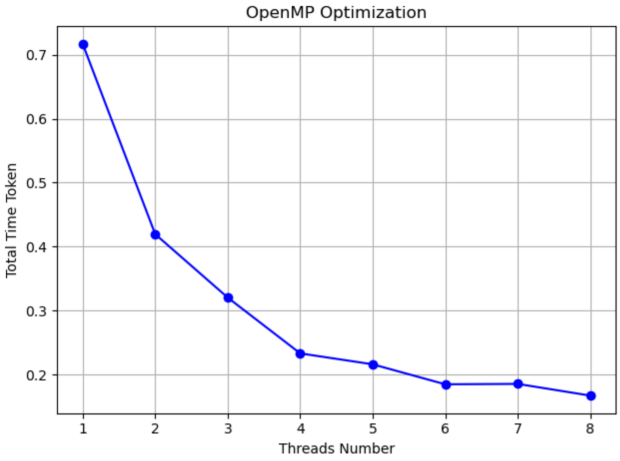

Figure 2: OpenMP Running time of in different numbers of threads

The overall trend shows a decrease in execution time as the number of threads increases, indicating that the program benefits from parallelization.
We use this speedup function to investigate the speedup ratio of different numbers of threads and data scales:

$$\text{Speedup} = \frac{T_1}{T_n}$$

If the speedup is ideal, then with n threads, the execution time should be

$$T_n = \frac{T_1}{n}$$

However, in practice, parallel overhead may limit the actual speedup, preventing it from reaching the ideal value.

We choose the input case **ellipse_N_05000**,**ellipse_N_08000** and **ellipse_N_10000**, with $\Delta t = 10^{-5}$ and 100 time steps. The plots are the following.

We can observe that the speedup values for the same number of threads are relatively close in different dataset sizes.And in our experiment, the optimization with OpenMP achieves better speedup compared to Pthreads.

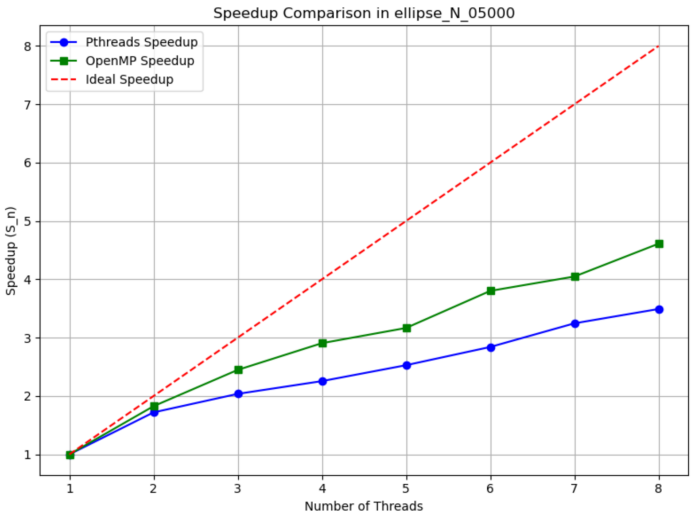

Figure 3: Speedup Comparison in ellipse_N_05000

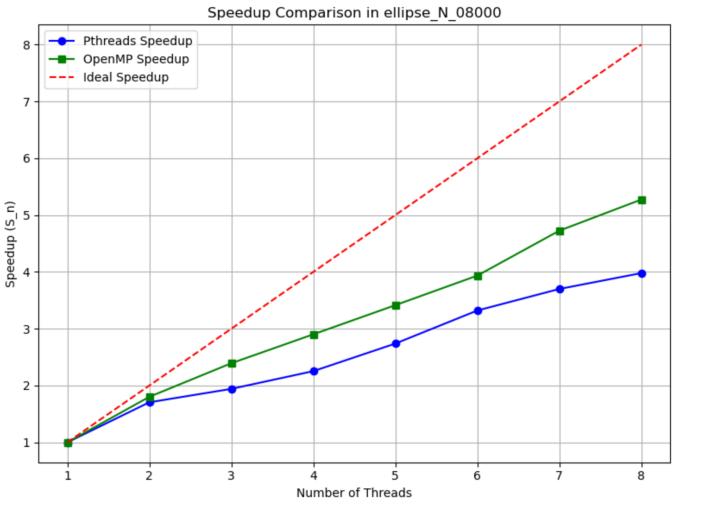

Figure 4: Speedup Comparison in ellipse_N_08000

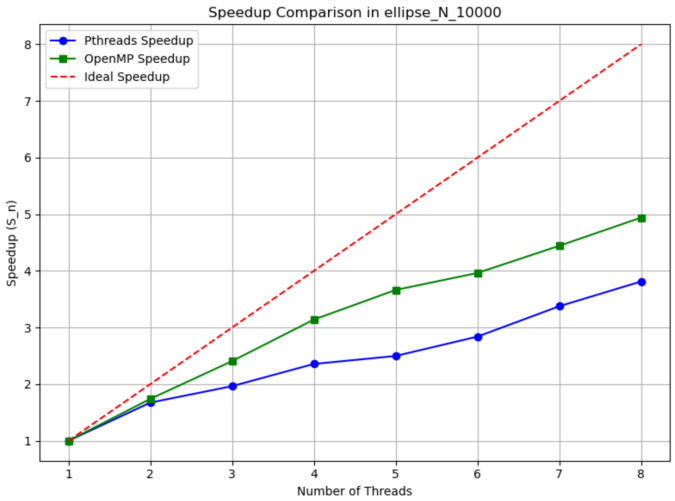

Figure 5: Speedup Comparison in ellipse_N_10000

References

[1] Uppsala University. High performance programming: Assignment 4: Parallelization. Spring 2022, 2022. Course material for High Performance Programming, Uppsala University.

[2] Jarmo Rantakokko. Parallel programming with pthreads on multi-core. Lecture slides for the "High Performance Programming" course, 2025. Uppsala University.

[3] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.