Assignment 4

Ting-Hsuan Lien

Problem

Calculate the evolution of *N* particles in a gravitational simulation.

Solution

Data structure (for particle)

Simply record all information of particle

```
/* Structure to store the state of a particle
 * x: position x
 * y: position y
 * mass: mass of the particle
 * v_x: velocity x
 * v_y: velocity y
 * brightness: brightness of the particle
 */
typedef struct
{
    double x, y, mass, v_x, v_y, brightness;
} Particle;
```

Structure (code)

My code can spilt into 6 main part.

Particle: Contains the definition of the `Particle` structure and function `Print()` for debugging purposes.

ParseArguments: reading input data from files to initialize particle properties.

ReadFile: read input data and create particles.

Simulation: The core of the program where all computations occur.

WriteFile: writing final result in to result.gal

FreeMemory: release all memory in the end.

Algorithm

Algorithm implement in Simulation part. Following are pseudocode.

```
for t in nsteps:
    for i in N:
        F = 0.0;
        for j in N:
            if i != j:
                dx = particles[i]->x - particles[j]->x;
                dy = particles[i]->y - particles[j]->y;
                r = sqrt((dx * dx) + (dy * dy));
                F += particles[j]->mass / (r * r * r);
        F *= -G * particles[i]->mass;
        a = F / particles[i]->mass;
        update paricles[i] velocity
    for i in N:
        update paricles[i] position
```

Performance and Discussion

All experiments run same data in 200 step (which has ref_output).

Original Version (without any optimizations)

INPUT DATA	REAL	USER	SYS
ellipse_N_00010	0m0.008s	0m0.002s	0m0.000s
ellipse_N_00100	0m0.027s	0m0.020s	0m0.000s
ellipse_N_00500	0m0.485s	0m0.456s	0m0.000s
ellipse_N_01000	0m1.890s	0m1.798s	0m0.000s
ellipse_N_02000	0m6.200s	0m7.277s	0m0.000s

In the **Original Version**, no specific optimizations were applied. The graph below clearly shows an **$O(n^2)$** curve.

Figure 1: original version - real time performance

Compile with `-o3`

INPUT DATA	REAL	USER	SYS
ellipse_N_00010	0m0.008s	0m0.001s	0m0.000s
ellipse_N_00100	0m0.015s	0m0.008s	0m0.000s
ellipse_N_00500	0m0.179s	0m0.163s	0m0.000s
ellipse_N_01000	0m0.681s	0m0.642s	0m0.000s
ellipse_N_02000	0m1.261s	0m2.587s	0m0.000s

In this version, no modifications were made to the code; only the `-O3` optimization flag was used during compilation. The execution time was significantly reduced. In the largest test case, the **real time** decreased by **79.6%**.

In Figure 2 the time complexity seem not like **$O(n^2)$**, but no modification of code, the algorithm still **$O(n^2)$**.

Figure 2: Compile with `-o3` version - real time performance

Compile with `-funroll-loops`

INPUT DATA	REAL	USER	SYS
ellipse_N_00010	0m0.009s	0m0.002s	0m0.000s
ellipse_N_00100	0m0.016s	0m0.008s	0m0.000s
ellipse_N_00500	0m0.180s	0m0.164s	0m0.000s
ellipse_N_01000	0m0.688s	0m0.648s	0m0.000s
ellipse_N_02000	0m1.271s	0m2.602s	0m0.001s

In this version, same as last version, just add another compile flag. It didn't effect the runtime (< 0.01s).

Move `F()` into main

INPUT DATA	REAL	USER	SYS
ellipse_N_00010	0m0.008s	0m0.002s	0m0.000s
ellipse_N_00100	0m0.015s	0m0.008s	0m0.000s
ellipse_N_00500	0m0.177s	0m0.152s	0m0.009s
ellipse_N_01000	0m0.671s	0m0.631s	0m0.000s
ellipse_N_02000	0m1.169s	0m2.516s	0m0.010s

Initially, I placed the code in `F()` for easy to modify. In this version, I moved it into `main` function to reduce function call. In the larger test case (N = 2000), this optimization reduced the execution time by 0.1s compared to before.

Figure 3 looks similar to Figure 2. but the peak is slightly lower.

Figure 3: Move `F()` into main - real time performance

Remove if (i != j) in loop

INPUT DATA	REAL	USER	SYS
ellipse_N_00010	0m0.008s	0m0.001s	0m0.000s
ellipse_N_00100	0m0.011s	0m0.005s	0m0.000s
ellipse_N_00500	0m0.103s	0m0.090s	0m0.000s
ellipse_N_01000	0m0.377s	0m0.342s	0m0.009s
ellipse_N_02000	0m1.487s	0m1.406s	0m0.000s

As shown in the pseudocode in *Algorithm*, the `for` loop originally used `if (i != j)` to skip the calculation for the particle itself. Since conditional checks inside loops can slow down execution, I modified the code to avoid using `if` statements inside the loop.

The new version pseudocode is following:

```
for t in nsteps:
    for i in N:
        F = 0.0;
        for j in range(0, i):
            dx = particles[i]->x - particles[j]->x;
            dy = particles[i]->y - particles[j]->y;
            r = sqrt((dx * dx) + (dy * dy));
            F += particles[j]->mass / (r * r * r);
        for j in range(i+1, N):
            dx = particles[i]->x - particles[j]->x;
            dy = particles[i]->y - particles[j]->y;
            r = sqrt((dx * dx) + (dy * dy));
            F += particles[j]->mass / (r * r * r);
        F *= -G * particles[i]->mass;
        a = F / particles[i]->mass;
        update paricles[i] velocity
    for i in N:
        update paricles[i] position
```

In **Figure 4**, the runtime for **"ellipse_N_01000"** shows a significant reduction, but the other cases did not change much. Notably, the runtime for "ellipse_N_02000" even increased.

Figure 4: Remove if (i != j) in loop - real time performance

Change for to do-while

INPUT DATA	REAL	USER	SYS
ellipse_N_00010	0m0.008s	0m0.001s	0m0.000s
ellipse_N_00100	0m0.012s	0m0.005s	0m0.000s
ellipse_N_00500	0m0.102s	0m0.090s	0m0.000s
ellipse_N_01000	0m0.383s	0m0.357s	0m0.000s
ellipse_N_02000	0m1.499s	0m1.408s	0m0.010s

In this version, I changed most of the loops from `for` to `do-while`, but it did not improve performance.

Move `Destroy()` into main

INPUT DATA	REAL	USER	SYS
ellipse_N_00010	0m0.008s	0m0.001s	0m0.000s
ellipse_N_00100	0m0.011s	0m0.005s	0m0.000s

INPUT DATA	REAL	USER	SYS
ellipse_N_00500	0m0.098s	0m0.086s	0m0.000s
ellipse_N_01000	0m0.363s	0m0.336s	0m0.000s
ellipse_N_02000	0m1.407s	0m1.328s	0m0.000s

Similar to *Move `F()` into main*, I moved `Destroy()` to `main`. It had a slight positive effect.

Reduce useless calculation

INPUT DATA	REAL	USER	SYS
ellipse_N_00010	0m0.008s	0m0.001s	0m0.000s
ellipse_N_00100	0m0.011s	0m0.005s	0m0.000s
ellipse_N_00500	0m0.098s	0m0.081s	0m0.009s
ellipse_N_01000	0m0.356s	0m0.335s	0m0.010s
ellipse_N_02000	0m1.428s	0m1.410s	0m0.000s

Lastly, I combined some redundant calculations and used faster operators. However, the effect was still not significant.

After modification, the new code is following:

```
for t in nsteps:
    for i in N:
        F = 0.0;
        for j in range(0, i):
            dx = particles[i]->x - particles[j]->x;
            dy = particles[i]->y - particles[j]->y;
            r = sqrt((dx * dx) + (dy * dy));
            F += particles[j]->mass / (r * r * r);
        for j in range(i+1, N):
            dx = particles[i]->x - particles[j]->x;
            dy = particles[i]->y - particles[j]->y;
            r = sqrt((dx * dx) + (dy * dy));
            F += particles[j]->mass / (r * r * r);
        particles[i]->v += - G * F * delta_t;
    for i in N:
        update paricles[i] position
```

Modify Array

INPUT DATA	REAL	USER	SYS
ellipse_N_00010	0m0.007s	0m0.001s	0m0.000s
ellipse_N_00100	0m0.010s	0m0.004s	0m0.000s
ellipse_N_00500	0m0.095s	0m0.090s	0m0.000s
ellipse_N_01000	0m0.341s	0m0.340s	0m0.000s
ellipse_N_02000	0m1.328s	0m1.351s	0m0.000s
ellipse_N_03000	0m1.494s	0m1.520s	0m0.000s

```c
typedef struct
{
    double* x;
    double* y;
    double* mass;
    double* v_x;
    double* v_y;
    double* brightness;
} Particle;
```

Modify the structure for better vectorization. The execution time was significantly reduced, especially in the case of "**ellipse_N_03000**", where the **real time** decreased from **0m3.118s** to **0m1.494s**.

Reduce the number of computations

INPUT DATA	REAL	USER	SYS
ellipse_N_00010	0m0.007s	0m0.001s	0m0.000s
ellipse_N_00100	0m0.011s	0m0.005s	0m0.000s
ellipse_N_00500	0m0.104s	0m0.100s	0m0.000s
ellipse_N_01000	0m0.388s	0m0.386s	0m0.010s
ellipse_N_02000	0m1.527s	0m1.578s	0m0.000s
ellipse_N_03000	0m1.703s	0m1.769s	0m0.000s

Compared to the previous version, where velocity and position updates were performed in a separate loop after force calculations, this version integrates the updates directly within the force computation loop.

The execution time slightly increased, especially for larger cases (**N = 2000, 3000**).

Vectorized

INPUT DATA	REAL	USER	SYS
ellipse_N_00010	0m0.009s	0m0.002s	0m0.000s
ellipse_N_00100	0m0.007s	0m0.002s	0m0.000s
ellipse_N_00500	0m0.031s	0m0.025s	0m0.001s
ellipse_N_01000	0m0.095s	0m0.090s	0m0.000s
ellipse_N_02000	0m0.340s	0m0.353s	0m0.000s
ellipse_N_03000	0m0.387s	0m0.391s	0m0.011s

By introducing additional variables, modifications within the **for j** loop no longer directly affect `F_x[i]` and `F_y[i]`, allowing for **vectorization**. After **vectorization**, the execution time was significantly reduced, especially for larger test cases.

Use Pthreads to parallelize

Original pthreads with mutex in for

INPUT DATA	REAL	USER	SYS
ellipse_N_00010	0m0.023s	0m0.000s	0m0.010s
ellipse_N_00100	0m0.033s	0m0.005s	0m0.005s
ellipse_N_00500	0m0.265s	0m0.029s	0m0.003s
ellipse_N_01000	0m0.916s	0m0.219s	0m0.003s
ellipse_N_02000	0m3.664s	0m2.787s	0m0.008s
ellipse_N_03000	0m3.965s	0m3.654s	0m0.019s

The table above shows the results for a **single-threaded run**. Due to **mutex**, the parallel version runs **almost 10 times slower** than the serial version.

The code section is following.

```
void* CalculateForce(void* arg)
{
    int start = ((ThreadData*)arg)->start;
    int end = ((ThreadData*)arg)->end;
    double* local_F_x = (double*)calloc(N, sizeof(double));
    double* local_F_y = (double*)calloc(N, sizeof(double));
    for (int i = start; i < end; ++i)
    {
        double F_i = 0.0;
        double F_j = 0.0;
        for (int j = i+1; j < N; ++j)
        {
            double dx = particles.x[i] - particles.x[j];
            double dy = particles.y[i] - particles.y[j];

            double r = sqrt((dx * dx) + (dy * dy)) + EPSILON;
            double inv_r3 = 1.0 / (r * r * r);

            double f_i = particles.mass[j] * inv_r3;
            double f_j = particles.mass[i] * inv_r3;

            F_i += f_i * dx;
            F_j += f_i * dy;
            pthread_mutex_lock(&lock);
            particles.F_x[j] -= f_j * dx;
            particles.F_y[j] -= f_j * dy;
            pthread_mutex_unlock(&lock);
        }

        pthread_mutex_lock(&lock);
        particles.F_x[i] += F_i;
        particles.F_y[i] += F_j;
        pthread_mutex_unlock(&lock);
    }

    pthread_exit(NULL);
}
```

Pthreads with local variables in `CalculateForce()`

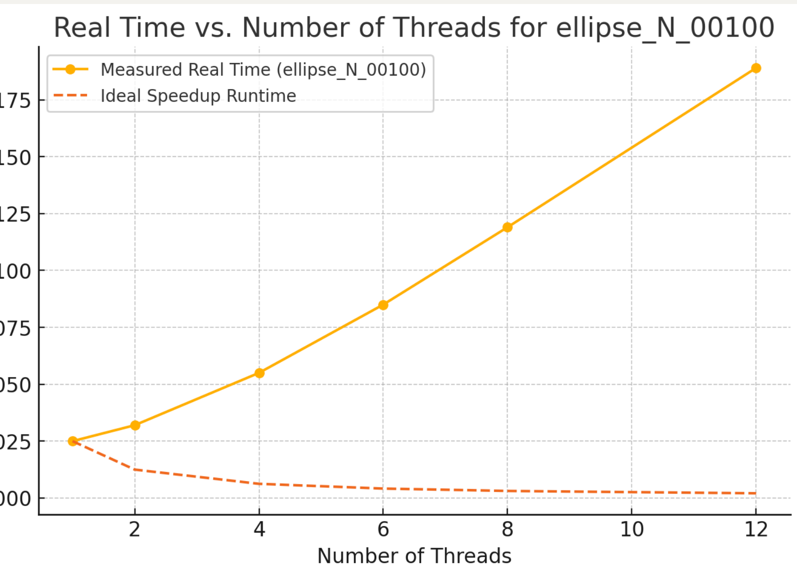
INPUT DATA	REAL	USER	SYS	POS_MAXDIFF
ellipse_N_00010	0m0.023s	0m0.000s	0m0.010s	0.000000000000
ellipse_N_00100	0m0.025s	0m0.000s	0m0.011s	0.000000000000
ellipse_N_00500	0m0.048s	0m0.011s	0m0.000s	0.000000000000
ellipse_N_01000	0m0.120s	0m0.015s	0m0.002s	0.000000000000

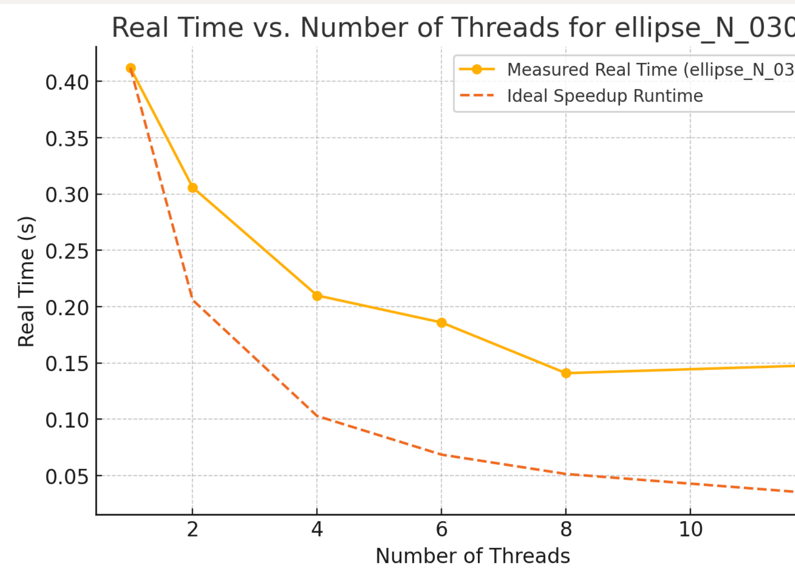
INPUT DATA	REAL	USER	SYS	POS_MAXDIFF
ellipse_N_02000	0m0.386s	0m0.050s	0m0.003s	0.000000000000
ellipse_N_03000	0m0.412s	0m0.099s	0m0.000s	0.000000000000

To avoid excessive **mutex locks** blocking all threads, **local variables** were used to store `F_x` and `F_y`, which were updated at the end instead of within the loop. This optimization significantly reduced execution time, making the parallel version (with single thread) **only slightly slower** than the serial version.

The code section is following.

```
void* CalculateForce(void* arg)
{
    int start = ((ThreadData*)arg)->start;
    int end = ((ThreadData*)arg)->end;
    double* local_F_x = (double*)calloc(N, sizeof(double));
    double* local_F_y = (double*)calloc(N, sizeof(double));
    for (int i = start; i < end; ++i)
    {
        double F_i = 0.0;
        double F_j = 0.0;
        for (int j = i+1; j < N; ++j)
        {
            double dx = particles.x[i] - particles.x[j];
            double dy = particles.y[i] - particles.y[j];

            double r = sqrt((dx * dx) + (dy * dy)) + EPSILON;
            double inv_r3 = 1.0 / (r * r * r);

            double f_i = particles.mass[j] * inv_r3;
            double f_j = particles.mass[i] * inv_r3;

            F_i += f_i * dx;
            F_j += f_i * dy;
            local_F_x[j] -= f_j * dx;
            local_F_y[j] -= f_j * dy;
        }
        local_F_x[i] += F_i;
        local_F_y[i] += F_j;
    }
    pthread_mutex_lock(&lock);
    for (int i = start; i < end; ++i)
    {
        particles.F_x[i] = local_F_x[i];
        particles.F_y[i] = local_F_y[i];
    }
    pthread_mutex_unlock(&lock);

    free(local_F_x);
    free(local_F_y);
    pthread_exit(NULL);
}
```





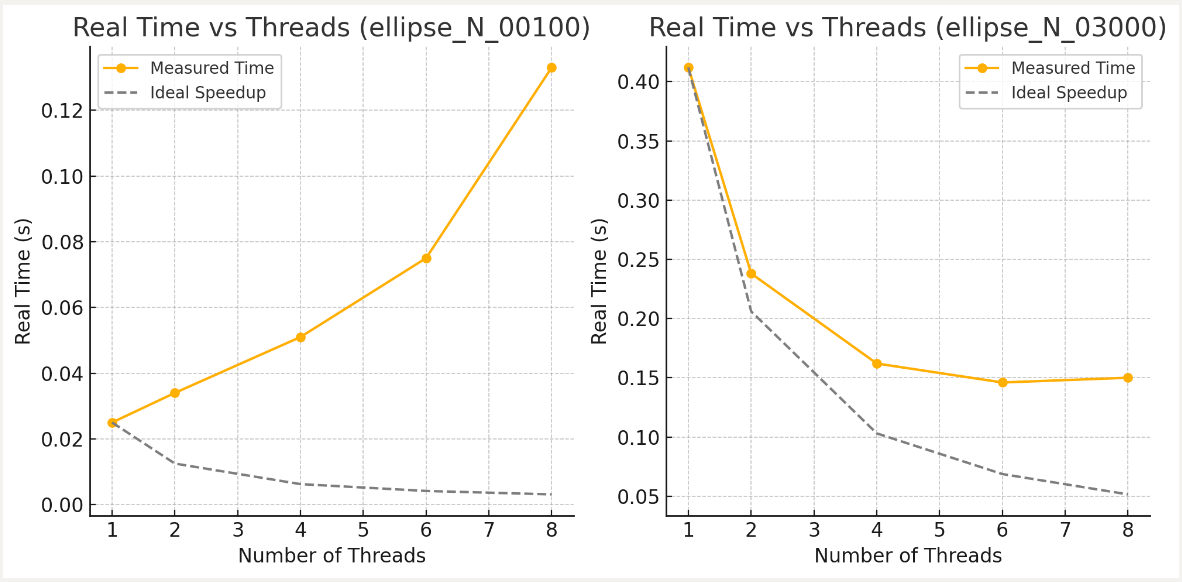
In smaller cases (**N = 100**), parallelization not only fails to speed up the execution but actually makes it significantly slower. This is likely due to the overhead of thread management outweighing the benefits of parallel computation.

For larger cases, when the number of threads is ≤ **6** (matching the number of CPU cores), speedup is observed, albeit not reaching the ideal linear speedup. However, the performance still shows a reasonable improvement.

Workload balance

To ensure that each thread has a balanced workload, calculate the triangular area to determine the range of `i` that each thread should handle.

```
    // split calculation area into n_threads
    int batch_size = N * N * 0.5 / n_threads;
    // int linear_batch_size = N / n_threads;
    int index = 0;
    int area = 0;
    for (int i = 0; i < n_threads; ++i)
    {
        data[i].start = index;
        area = 0;
        while (area < batch_size && index < N)
        {
            area += N - index;
            index++;
        }
        data[i].end = index;
        // data[i].start = i * linear_batch_size;
        // data[i].end = (i + 1) * linear_batch_size;
    }
    data[n_threads - 1].end = N;
```

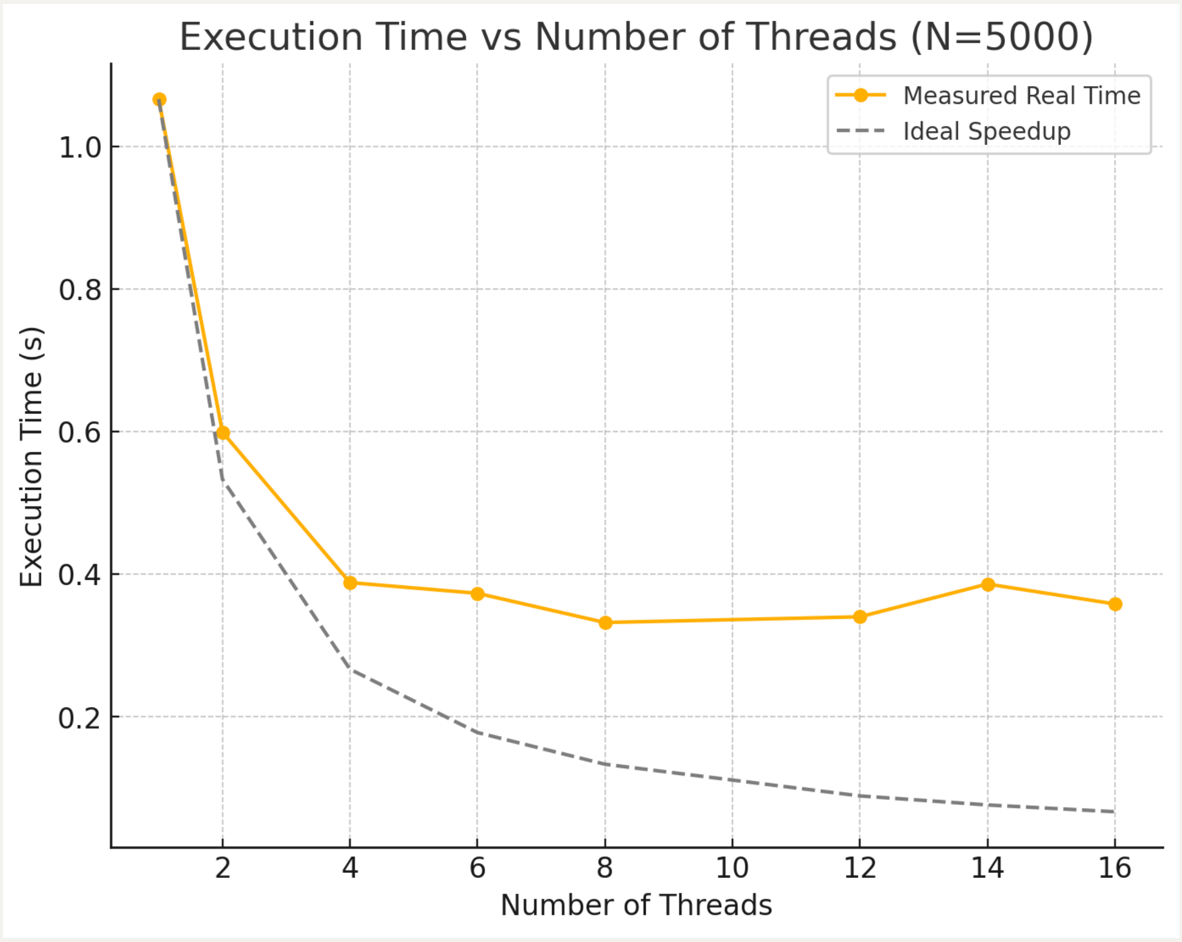


In smaller cases, parallelization still performs poorly. However, after balancing the workload, the parallel performance in larger cases is much closer to the ideal.

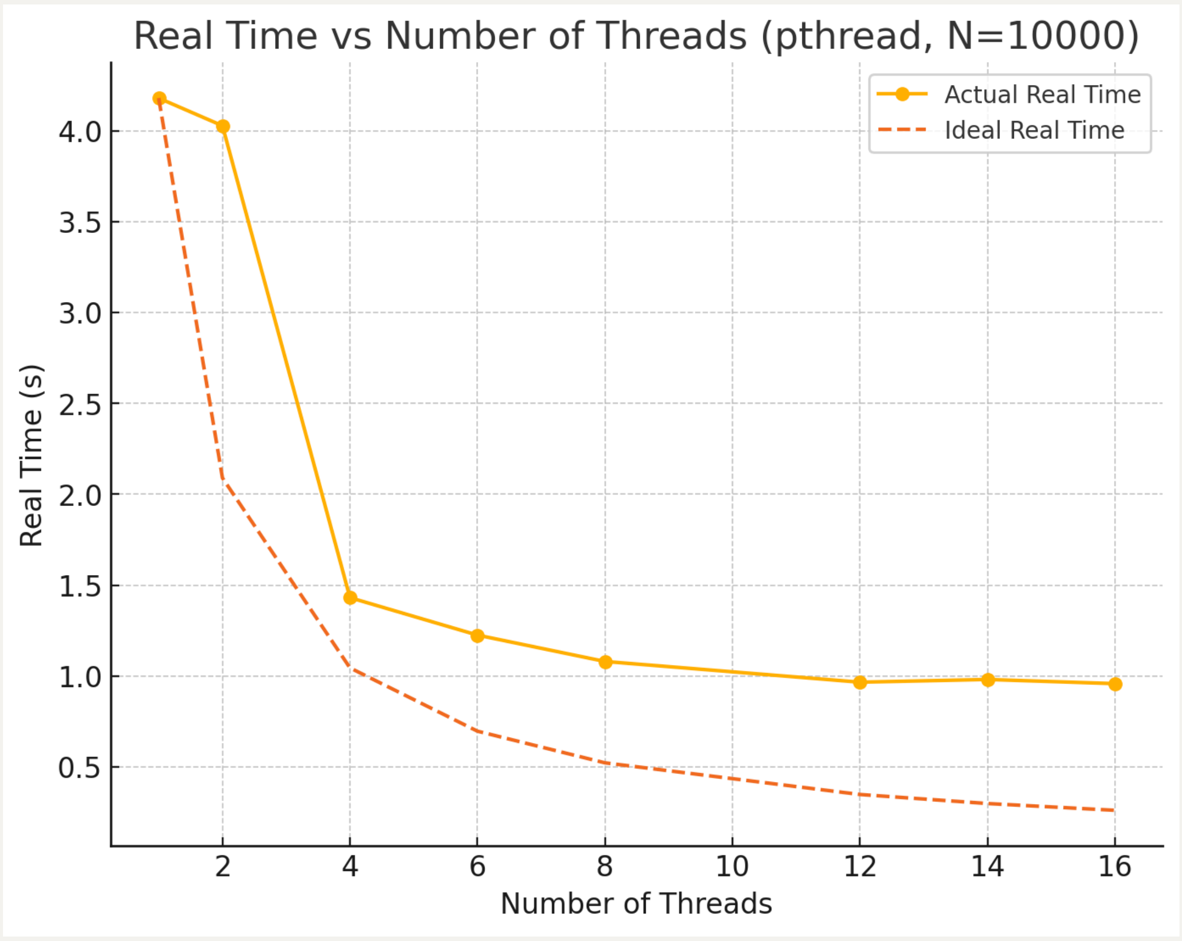
ellipse_N_05000 and ellipse_N_10000

THREADS	REAL TIME	USER TIME	SYS TIME
1	0m1.066s	0m0.486s	0m0.025s
2	0m0.599s	0m0.345s	0m0.004s
4	0m0.388s	0m0.212s	0m0.012s
6	0m0.373s	0m0.270s	0m0.015s
8	0m0.332s	0m0.291s	0m0.017s

THREADS	REAL TIME	USER TIME	SYS TIME
12	0m0.340s	0m0.272s	0m0.028s
14	0m0.386s	0m0.375s	0m0.053s
16	0m0.358s	0m0.385s	0m0.032s



When `N_Threads=8`, there is the best performance.



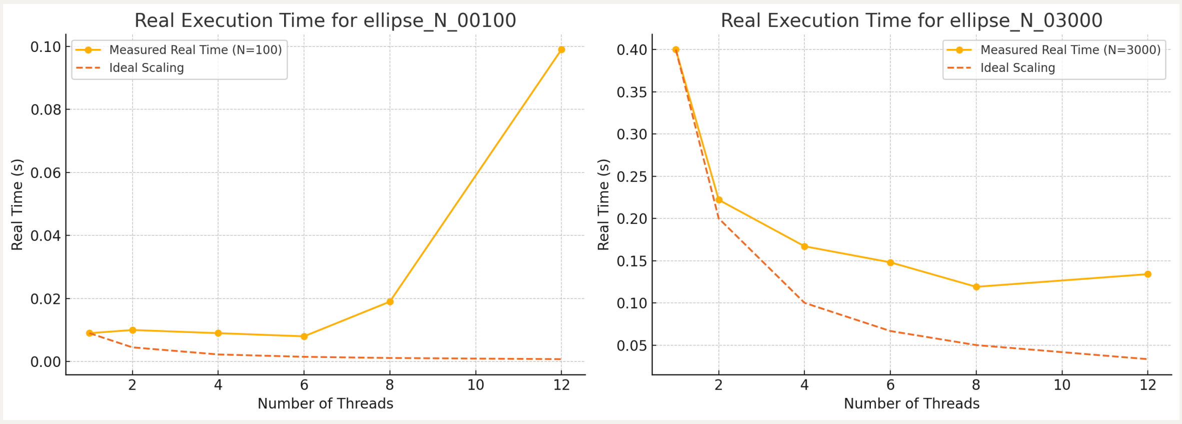
When ***N*** increasing to 10000, the runtime more close to ideal runtime.

Use OpenMP to parallelize

```
void CalculateForce()
{
    ...
    #pragma omp critical
    {
        for (int i = 0; i < N; ++i)
        {
            particles.F_x[i] += local_F_x[i];
            particles.F_y[i] += local_F_y[i];
        }
    }
    ...
}
int main()
{
    ...
    // using OpenMP to parallelize the loop
        #pragma omp parallel for num_threads(n_threads)
        for (int j = 0; j < n_threads; ++j)
        {
            CalculateForce(data[j].start, data[j].end);
        }

        for (int i = 0; i < N; ++i)
        {
            particles.v_x[i] += - G * particles.F_x[i] * delta_t;
            particles.v_y[i] += - G * particles.F_y[i] * delta_t;
            particles.x[i] += particles.v_x[i] * delta_t;
            particles.y[i] += particles.v_y[i] * delta_t;
        }
    ...
}
```

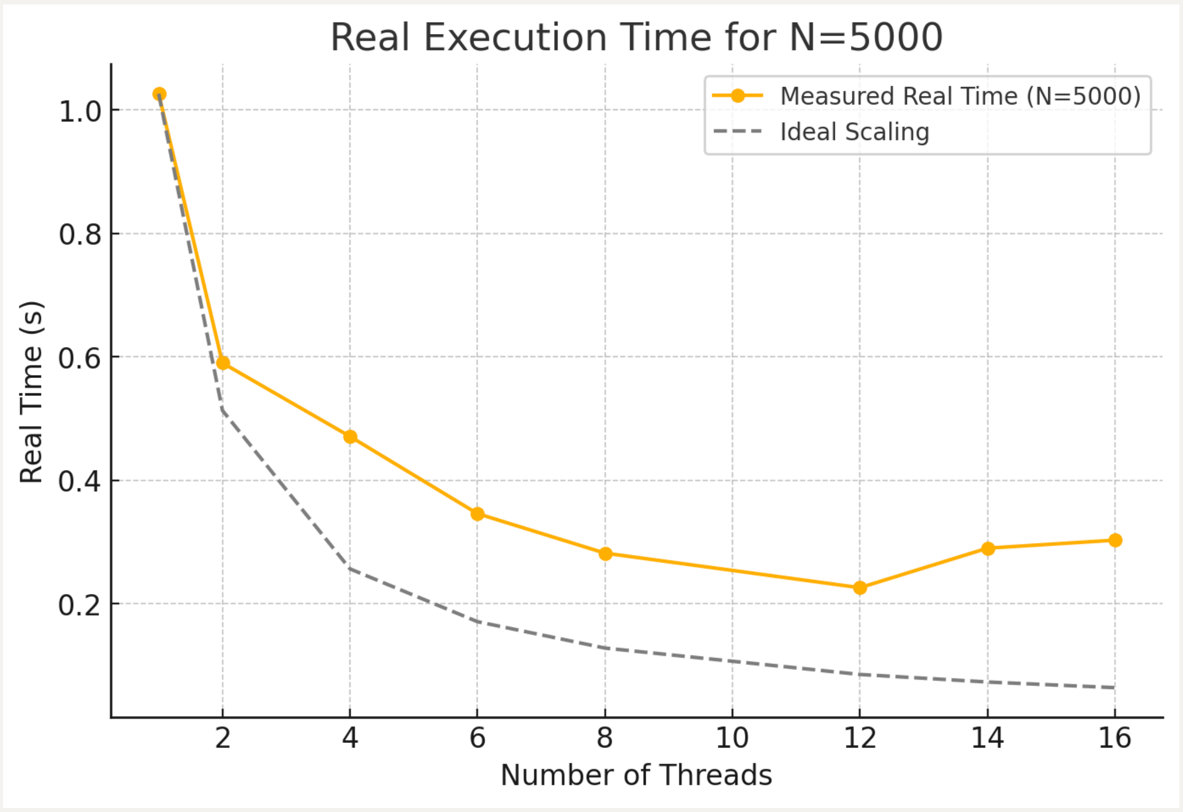
INPUT DATA	REAL	USER	SYS
ellipse_N_00010	0m0.008s	0m0.002s	0m0.000s
ellipse_N_00100	0m0.009s	0m0.003s	0m0.000s
ellipse_N_00500	0m0.031s	0m0.023s	0m0.000s
ellipse_N_01000	0m0.096s	0m0.087s	0m0.000s
ellipse_N_02000	0m0.353s	0m0.331s	0m0.010s
ellipse_N_03000	0m0.400s	0m0.384s	0m0.000s

This is the execution result using `nthreads=1`. It can be observed that the execution time does not differ significantly from the serial and pthreads versions.



We can see that in the **smaller case (N=100)**, parallelization still performs poorly. However, the overall performance using **OpenMP** is significantly better than **pthreads**.

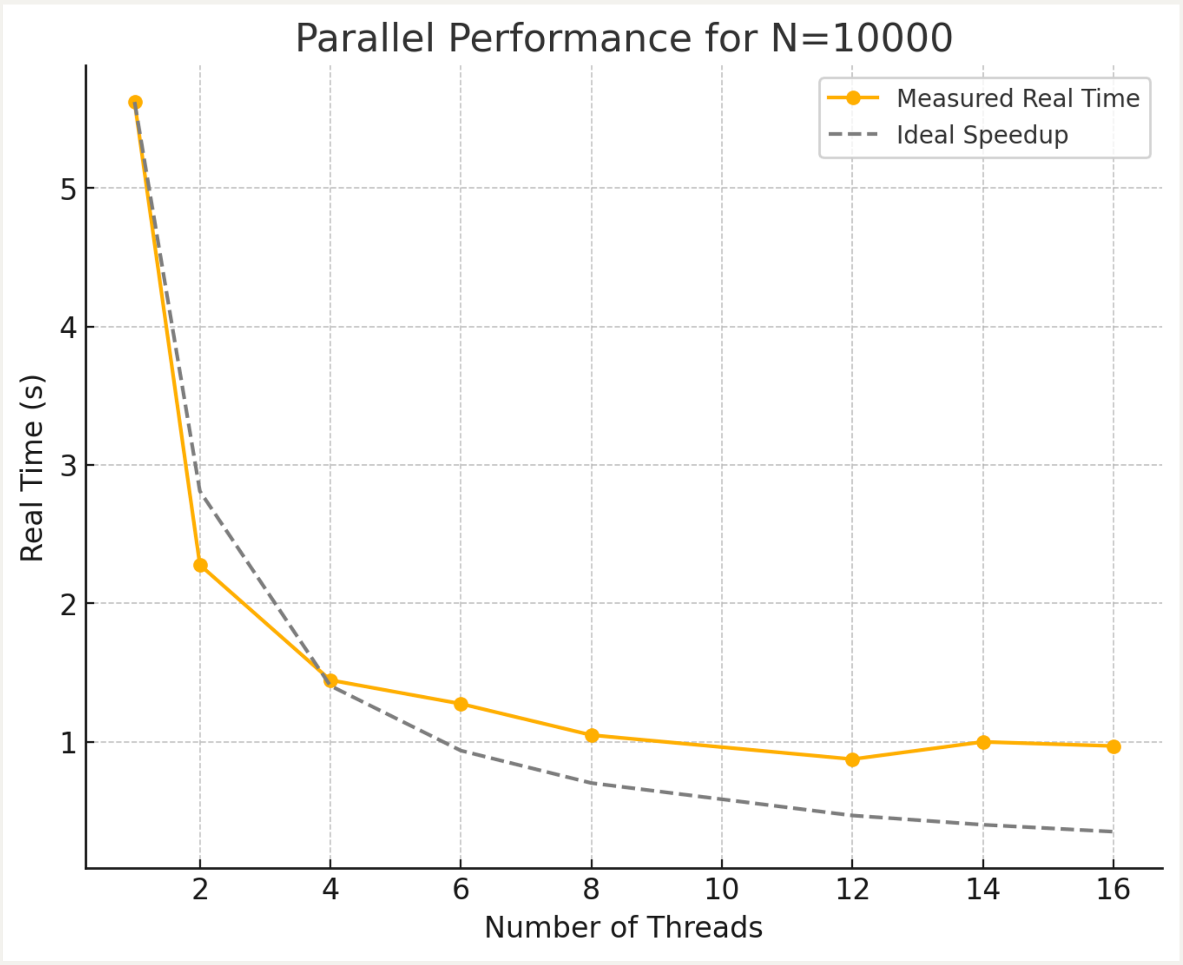
ellipse_N_05000



THREADS	REAL TIME (S)	USER TIME (S)	SYS TIME (S)
1	1.026	1.040	0.010
2	0.590	1.195	0.000
4	0.471	1.889	0.000
6	0.346	2.028	0.021
8	0.282	2.211	0.000
12	0.226	2.596	0.021
14	0.290	2.072	0.042
16	0.303	1.910	0.075

In `N=5000`, we can see `N_Threads=12` has the best performance.

ellipse_N_10000



THREADS	REAL TIME (S)	USER TIME (S)	SYS TIME (S)
1	5.621	4.217	0.010
2	2.278	4.656	0.031
4	1.446	5.806	0.011
6	1.275	7.717	0.052
8	1.049	8.428	0.042
12	0.875	10.322	0.063
14	1.000	8.213	0.177
16	0.970	8.134	0.133

In `N=10000`, we can see `N_Threads=12` also has the best performance.

Discussion

Overall, **OpenMP** is easier to implement and performs better than **pthreads**. **Pthreads** suffers from high thread management overhead, making execution sometimes even **slower than serial**, especially for smaller cases. Its performance with multiple threads is far from ideal due to synchronization and workload imbalance.

OpenMP, on the other hand, scales better and reduces execution time more effectively. In larger cases, parallelization is more effective. For **N=10000**, the speedup curve closely follows the ideal up to **N_THREADS=12**, showing significant improvements.

However, it still **doesn't reach ideal speedup**, likely due to memory access patterns, scheduling overhead, or Amdahl's Law.

Environment

Run on Windows 11 wsl.

CPU: AMD Ryzen 5 7535HS with Radeon Graphics

References

- ChatGPT: https://chatgpt.com/

Appendix

- GitHub: https://github.com/sophie8909/High-performance-programming

Pthreads (linear split workload version) Tables

n_Threads = 2

INPUT DATA	REAL	USER	SYS	POS_MAXDIFF
ellipse_N_00010	0m0.033s	0m0.000s	0m0.021s	0.000000000000
ellipse_N_00100	0m0.032s	0m0.010s	0m0.010s	0.000000000000
ellipse_N_00500	0m0.041s	0m0.014s	0m0.009s	0.000000000000
ellipse_N_01000	0m0.086s	0m0.020s	0m0.002s	0.000000000000

INPUT DATA	REAL	USER	SYS	POS_MAXDIFF
ellipse_N_02000	0m0.282s	0m0.047s	0m0.001s	0.000000000000
ellipse_N_03000	0m0.306s	0m0.066s	0m0.000s	0.000000000000

n_Threads = 4

INPUT DATA	REAL	USER	SYS	POS_MAXDIFF
ellipse_N_00010	0m0.055s	0m0.000s	0m0.041s	0.000000000000
ellipse_N_00100	0m0.055s	0m0.008s	0m0.033s	0.000000000000
ellipse_N_00500	0m0.056s	0m0.022s	0m0.018s	0.000000000000
ellipse_N_01000	0m0.070s	0m0.026s	0m0.017s	0.000000000000
ellipse_N_02000	0m0.195s	0m0.052s	0m0.009s	0.000000000000
ellipse_N_03000	0m0.210s	0m0.067s	0m0.004s	0.000000000000

n_Threads = 6

INPUT DATA	REAL	USER	SYS	POS_MAXDIFF
ellipse_N_00010	0m0.089s	0m0.007s	0m0.068s	0.000000000000
ellipse_N_00100	0m0.085s	0m0.021s	0m0.049s	0.000000000000
ellipse_N_00500	0m0.084s	0m0.030s	0m0.042s	0.000000000000
ellipse_N_01000	0m0.093s	0m0.048s	0m0.047s	0.000000000000
ellipse_N_02000	0m0.179s	0m0.081s	0m0.029s	0.000000000000
ellipse_N_03000	0m0.186s	0m0.070s	0m0.004s	0.000000000000

n_Threads = 8

INPUT DATA	REAL	USER	SYS	POS_MAXDIFF
ellipse_N_00010	0m0.119s	0m0.016s	0m0.092s	0.000000000000
ellipse_N_00100	0m0.119s	0m0.019s	0m0.087s	0.000000000000
ellipse_N_00500	0m0.093s	0m0.023s	0m0.069s	0.000000000000
ellipse_N_01000	0m0.115s	0m0.064s	0m0.058s	0.000000000000
ellipse_N_02000	0m0.148s	0m0.107s	0m0.032s	0.000000000000
ellipse_N_03000	0m0.141s	0m0.071s	0m0.010s	0.000000000000

n_Threads = 12

INPUT DATA	REAL	USER	SYS	POS_MAXDIFF
ellipse_N_00010	0m0.169s	0m0.030s	0m0.151s	0.000000000000
ellipse_N_00100	0m0.189s	0m0.029s	0m0.174s	0.000000000000
ellipse_N_00500	0m0.185s	0m0.055s	0m0.131s	0.000000000000
ellipse_N_01000	0m0.184s	0m0.081s	0m0.130s	0.000000000000
ellipse_N_02000	0m0.205s	0m0.185s	0m0.094s	0.000000000000
ellipse_N_03000	0m0.148s	0m0.156s	0m0.062s	0.000000000000

Pthreads (workload balance version) Tables

n_Threads = 2

INPUT DATA	REAL	USER	SYS	POS_MAXDIFF
ellipse_N_00010	0m0.036s	0m0.000s	0m0.022s	0.000000000000
ellipse_N_00100	0m0.034s	0m0.000s	0m0.020s	0.000000000000
ellipse_N_00500	0m0.046s	0m0.010s	0m0.013s	0.000000000000
ellipse_N_01000	0m0.082s	0m0.013s	0m0.011s	0.000000000000
ellipse_N_02000	0m0.230s	0m0.045s	0m0.002s	0.000000000000
ellipse_N_03000	0m0.238s	0m0.058s	0m0.005s	0.000000000000

n_Threads = 4

INPUT DATA	REAL	USER	SYS	POS_MAXDIFF
ellipse_N_00010	0m0.056s	0m0.012s	0m0.030s	0.000000000000
ellipse_N_00100	0m0.051s	0m0.006s	0m0.034s	0.000000000000
ellipse_N_00500	0m0.057s	0m0.005s	0m0.036s	0.000000000000
ellipse_N_01000	0m0.076s	0m0.034s	0m0.010s	0.000000000000
ellipse_N_02000	0m0.157s	0m0.049s	0m0.006s	0.000000000000
ellipse_N_03000	0m0.162s	0m0.052s	0m0.005s	0.000000000000

n_Threads = 6

INPUT DATA	REAL	USER	SYS	POS_MAXDIFF
ellipse_N_00010	0m0.074s	0m0.000s	0m0.068s	0.000000000000
ellipse_N_00100	0m0.075s	0m0.000s	0m0.069s	0.000000000000
ellipse_N_00500	0m0.078s	0m0.009s	0m0.061s	0.000000000000
ellipse_N_01000	0m0.095s	0m0.041s	0m0.050s	0.000000000000
ellipse_N_02000	0m0.153s	0m0.068s	0m0.019s	0.000000000000
ellipse_N_03000	0m0.146s	0m0.053s	0m0.013s	0.000000000000

n_Threads = 8

INPUT DATA	REAL	USER	SYS	POS_MAXDIFF
ellipse_N_00010	0m0.101s	0m0.018s	0m0.078s	0.000000000000
ellipse_N_00100	0m0.133s	0m0.008s	0m0.121s	0.000000000000
ellipse_N_00500	0m0.133s	0m0.008s	0m0.111s	0.000000000000
ellipse_N_01000	0m0.136s	0m0.056s	0m0.072s	0.000000000000
ellipse_N_02000	0m0.177s	0m0.108s	0m0.038s	0.000000000000
ellipse_N_03000	0m0.150s	0m0.084s	0m0.012s	0.000000000000

OpenMP Tables

n_Threads = 2

INPUT DATA	REAL	USER	SYS
ellipse_N_00010	0m0.009s	0m0.003s	0m0.000s
ellipse_N_00100	0m0.010s	0m0.002s	0m0.000s
ellipse_N_00500	0m0.021s	0m0.013s	0m0.007s
ellipse_N_01000	0m0.059s	0m0.097s	0m0.000s
ellipse_N_02000	0m0.206s	0m0.390s	0m0.000s
ellipse_N_03000	0m0.222s	0m0.423s	0m0.000s

n_Threads = 4

INPUT DATA	REAL	USER	SYS
ellipse_N_00010	0m0.009s	0m0.002s	0m0.000s
ellipse_N_00100	0m0.009s	0m0.006s	0m0.000s
ellipse_N_00500	0m0.018s	0m0.024s	0m0.000s
ellipse_N_01000	0m0.042s	0m0.121s	0m0.000s
ellipse_N_02000	0m0.116s	0m0.413s	0m0.010s
ellipse_N_03000	0m0.167s	0m0.587s	0m0.019s

n_Threads = 6

INPUT DATA	REAL	USER	SYS
ellipse_N_00010	0m0.010s	0m0.006s	0m0.000s
ellipse_N_00100	0m0.008s	0m0.005s	0m0.000s
ellipse_N_00500	0m0.020s	0m0.039s	0m0.000s
ellipse_N_01000	0m0.045s	0m0.175s	0m0.000s
ellipse_N_02000	0m0.112s	0m0.605s	0m0.000s
ellipse_N_03000	0m0.148s	0m0.802s	0m0.010s

n_Threads = 8

INPUT DATA	REAL	USER	SYS
ellipse_N_00010	0m0.010s	0m0.011s	0m0.000s
ellipse_N_00100	0m0.019s	0m0.065s	0m0.009s
ellipse_N_00500	0m0.030s	0m0.145s	0m0.010s
ellipse_N_01000	0m0.043s	0m0.263s	0m0.000s
ellipse_N_02000	0m0.110s	0m0.802s	0m0.000s
ellipse_N_03000	0m0.119s	0m0.836s	0m0.020s

n_Threads = 12

INPUT DATA	REAL	USER	SYS
ellipse_N_00010	0m0.055s	0m0.563s	0m0.010s

INPUT DATA	REAL	USER	SYS
ellipse_N_00100	0m0.099s	0m1.046s	0m0.010s
ellipse_N_00500	0m0.087s	0m0.926s	0m0.020s
ellipse_N_01000	0m0.080s	0m0.866s	0m0.009s
ellipse_N_02000	0m0.159s	0m1.717s	0m0.020s
ellipse_N_03000	0m0.134s	0m1.456s	0m0.021s