HPP-A4 Report

March 7th, 2025

1 Introduction

In this assignment, I solved the gravitational N-body problem which involves calculating the pairwise interactions between N particles under Newton's law of gravitation. This report describes how some optimizations techniques can be used to improve performance while still keeping $O(N^2)$ algorithmic complexity. Moreover, the report provides an overview of how optimizations were applied on top of baseline implementations along with timing measurements.

2 The Problem

The simulation is based on Newton's law of gravitation modified with a Plummer sphere smoothing to avoid singularities. The gravitational force between two particles is given by:

$$\mathbf{F}_{ij} = -G\frac{m_i m_j}{(r_{ij}+\varepsilon)^3}\mathbf{r}_{ij} \tag{1}$$

where:

- $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$ is the vector displacement.
- $r_{ij} = \|\mathbf{r}_{ij}\|$ is the Euclidean distance between particle i and j.
- ε is a small constant to avoid singularities.

Particle positions and velocities are updated using the symplectic Euler method:

$$\mathbf{v}_i^{n+1} = \mathbf{v}_i^n + \Delta t\, \mathbf{a}_i^n,$$
$$\mathbf{x}_i^{n+1} = \mathbf{x}_i^n + \Delta t\, \mathbf{v}_i^{n+1}.$$

3 The Solution

3.1 Data Structures

The simulation is implemented using a structured data representation. At first, the core data structure used for storing particle attributes was the `particle_t` struct:

```
typedef struct Particle {
    double pos_dx;
    double pos_dy;
    double m_diff;
    double vel_dx;
    double vel_dy
    double b_diff;
} particle_t;
```

However, this data Array of Struct data layout was suboptimal and was later changed to the following during optimization:

```
typedef struct ParticleSystem {
    double* pos_dx;
    double* pos_dy;
    double* m_diff;
    double* vel_dx;
    double* vel_dy;
    double* b_diff;
    double* tmp_fx;
    double* tmp_fy;
    int n_particles;
} particle_system_t;
```

Moreover, force arrays `Fx` and `Fy` were added to store the accumulation of particle forces during optimizations.

3.2 Code Structure

The program follows a structured implementation consisting of:

- `calculateParticleMotion (Serial)`: Computes gravitational forces, updates velocities, and updates positions.
- `main`: Orchestrates file handling, time stepping, and optionally renders the graphical simulation.
- `galsim_worker (Pthreads)` : Used to create threads and run the simulation loop for a subset of particles
- `barrier_init`, `barrier_wait`, `barrier_destroy (Pthreads)`: Similar to pthreads barrier but created due to lack of support for these barriers by macOS gcc@12.
- `process_galsim_omp (OpenMP)`: Executes the simulation loop in parallel using OpenMP directives

3.3 Serial Implementations

3.3.1 Initial

In the initial implementation of the algorithm using AoS data layout, I iterated on all pairs of particles and used a conditional check to prevent particle self-interaction, and calculated the accumulated force in local variables and used them to update the velocity of the particles in outer loop. Finally, velocity was used to calculate the new position outside the computation loops.

```
for (int i = 0; i < N; i++) {
    double Fx = 0.0, Fy = 0.0;
    for (int j = 0; j < N; j++) {
        if (i == j) continue; // Skip self-interaction
        double dx = particles[i].pos_dx - particles[j].pos_dx;
        double dy = particles[i].pos_dy - particles[j].pos_dy;
        double r = sqrt(dx * dx + dy * dy) + epsilon;
        double inv_r3 = 1.0 / (r * r * r);
        Fx += dx * particles[j].m_diff * inv_r3;
        Fy += dy * particles[j].m_diff * inv_r3;
    }
    particles[i].vel_dx += Gdelta * Fx;
    particles[i].vel_dy += Gdelta * Fy;
}
```

```
for (i = 0; i < N; i++){
    particles[i].pos_dx += delta_t*particles[i].vel_dx;
    particles[i].pos_dy += delta_t*particles[i].vel_dy;
}
```

But there is a fundamental drawback in the implementation of this algorithm i.e. each pair (i, j) is computed twice leading to redundant computations and the `if (i == j)` condition introduces unnecessary branching inside the nested loop.

3.3.2 Optimized

To remove redundancy, I leveraged Newton's Third Law such that each unique pair (i, j) is computed only once.

$$\mathbf{F}_{ij} = -\mathbf{F}_{ji} \tag{2}$$

```
for (int i = 0; i < N; i++){
    double pos_x_i = particles[i].pos_dx;
    double pos_y_i = particles[i].pos_dy;
    double m_diff_i = particles[i].m_diff;
    for (int j = i+1; j < N; j++){
        const double dx = pos_x_i - particles[j].pos_dx;
        const double dy = pos_y_i - particles[j].pos_dy;
        const double r = dx * dx + dy * dy;
        const double denom = sqrt(r) + epsilon;
        const double d3 = denom * denom * denom;
        const double inv_r3 = 1.0 / d3;
        const double Fxij = dx * inv_r3;
        const double Fyij = dy * inv_r3;
        Fx[i] += particles[j].m_diff * Fxij;
        Fy[i] += particles[j].m_diff * Fyij;
        Fx[j] -= m_diff_i * Fxij;
        Fy[j] -= m_diff_i* Fyij;
    }
}

for (int i = 0; i < N; i++) {
    particles[i].vel_dx += Gdelta * Fx[i];
    particles[i].vel_dy += Gdelta * Fy[i];
    particles[i].pos_dx += delta_t*particles[i].vel_dx;
    particles[i].pos_dy += delta_t*particles[i].vel_dy;
}
```

To remove the branching in nested loop and to compute interactions simultaneously, I introduced separate `Fx` and `Fy` arrays for force accumulation. These arrays were created in `main` and zero-initialized (using `memset`) in calculateParticleMotion. These force accumulations are then used outside the computation loops to update the velocities and positions of the particles in a single loop. This optimized implementation of the algorithm results in elimination of the redundant conditional check and significantly reduces the number of force computations as each pair is processed only once.

3.4 Parallel Implementations

In order to further reduce the execution time of the simulation, I developed parallel implementations of the previously implemented serial code using shared memory models. In this approach, multiple threads

concurrently operate on the particle data, allowing the computation to remain $O(N^2)$ while reducing the overall runtime. The following sections describe the basic implementations using shared memory models: Pthreads and OpenMP.

3.4.1 Pthreads

For the Pthreads implementation, I employed POSIX threads to explicitly manage parallelism. The simulation data is divided among a fixed number of threads, with each thread processing a distinct subset of particles. To support this implementation, I introduced several data structures to manage shared simulation parameters and thread-specific data clearly.

The shared simulation parameters and synchronization primitives were grouped together in the following structure:

```
typedef struct simulation_context_t {
    particle_system_t* system;          // Pointer to particle data structure
    int nsteps;                         // Number of simulation steps
    double delta;                       // Simulation time step size
    int nthreads;                       // Total number of threads
    barrier_t barrier;                  // Custom barrier for synchronization
    volatile int next_index;            // Shared index for dynamic allocation
    pthread_mutex_t next_index_mutex;   // Mutex protecting the shared index
    thread_data_t* thread_args;         // Array holding each thread's data
} simulation_context_t;
```

Each thread also has its own private data, encapsulated within the following structure:

```
typedef struct thread_data {
    int thread_id;                      // Unique identifier of the thread
    simulation_context_t* context;      // Pointer to the shared simulation
        context
    double* thread_tmp_fx;              // Thread-local force accumulation (x-
        direction)
    double* thread_tmp_fy;              // Thread-local force accumulation (y-
        direction)
} thread_data_t;
```

Since I wrote this code on my personal machine having macOS, which did not support GCC built-in barriers, I implemented a custom barrier using a mutex and a condition variable as follows:

```
typedef struct {
    pthread_mutex_t mutex;              // Mutex for barrier synchronization
    pthread_cond_t cond;                // Condition variable for thread signaling
    int nthreads;                       // Number of threads to synchronize
    int waiting;                        // Number of threads currently waiting
    int state;                          // State variable for toggling barrier
        phases
} barrier_t;
```

In this design, a dynamic work allocation strategy was employed using the shared indices (guarded by a mutex) to assign iterations at runtime, more specifically CHUNK-sized iterations to reduce locking overhead. The custom barrier synchronizes threads at critical stages, such as after local force accumulation and before updating velocities and positions. Each thread maintains its own temporary force arrays (for both x and y directions) to store computed force contributions, ensuring safe parallel updates without race conditions.

3.4.2 OpenMP

For the OpenMP implementation, a much simpler and cleaner approach was adopted compared to the pthreads version. OpenMP allowed parallelization without explicit thread management or manual barriers. Each simulation step begins by resetting the temporary force arrays (`tmp_fx` and `tmp_fy`) to zero using the `#pragma omp single` directive combined with `memset`, as this operation is efficient and does not significantly benefit from parallelization.

The force computation loop was parallelized using the `#pragma omp parallel for` directive with a dynamic schedule. The built-in OpenMP `reduction` clause was used to accumulate results safely across threads, automatically creating and merging private copies of the force arrays. Finally, velocities and positions were updated using a statically scheduled parallel loop, as the workload here was evenly distributed.

4 Performance and Discussion

4.1 Experimental Setup

- **Input Data:** In all experiments, the same input file `ellipse_N_02000.gal` with 200 timesteps (`nsteps = 200`) was used.
- **Simulation Parameters:** $\Delta t = 10^{-5}$, 200 timesteps.
- **System Configuration:**
  - CPU: Intel(R) Xeon(R) CPU E5520 @ 2.27GHz (8 cores)
  - OS: Ubuntu 22.04.5 LTS
  - Compiler: GCC 11.4.0 with flags `-O3 -march=native -ffast-math`
- **Measurement Details:** To evaluate the performance improvements, execution times were measured using wall-clock timer, which makes use of `gettimeofday()` function. Timing measurements exclude graphics routines and only include the overall simulation in `nsteps`.
- **Profiling Tools:** In addition to wall-clock timing, some profiling tools were used to analyze performance in detail:
  - **operf and opannotate:** They were used to profile the application and analyze performance on a per-line basis and identify hotspots within the force computation loops.
  - **cachegrind (via valgrind):** To examine cache behavior, cachegrind was used to monitor cache read misses and branch prediction efficiency.

4.2 Optimizations

The goal of this assignment was to optimize our particle simulation code from a baseline implementation to a highly tuned version using both serial and parallel optimizations. In this section, serial optimizations are demonstrated showing the impact of algorithmic and memory-access improvements.

4.2.1 Baseline:

First the baseline program executed in an average of **19.434136 sec**. The code included an `if (i == j)` check to avoid self-interactions, which added unnecessary overhead.

4.2.2 Computations Improvement:

During optimizations, I ran `operf` to see which lines were being computationally expensive and then made some rearrangements to the force calculations. For instance, the nested loop having force calculations was changed from :

```
 21193   5.9772 :            double dx = particles[i].pos_dx - particles[j].pos_dx;
 25202   7.1079 :            double dy = particles[i].pos_dy - particles[j].pos_dy;
 17644   4.9763 :            double denom = sqrt(dx * dx + dy * dy) + epsilon;
160130  45.1628 :            double inv_r3 = 1.0 / (denom * denom * denom);
 83221  23.4715 :            Fx += dx * (particles[j].m_diff * inv_r3);
 16622   4.6880 :            Fy += dy * (particles[j].m_diff * inv_r3);
```

to:

```
 12097   3.5713 :            const double dx = pos_x_i - particles[j].pos_dx;
 12744   3.7623 :            const double dy = pos_y_i - particles[j].pos_dy;
  7889   2.3290 :            const double r = dx * dx + dy * dy;
 10027   2.9602 :            const double denom = sqrt(r) + epsilon;
 29106   8.5928 :            const double d3 = denom * denom * denom;
120359  35.5327 :            const double inv_r3 = 1.0 / d3;
 45477  13.4259 :            const double factor = particles[j].m_diff * inv_r3;
 54992  16.2349 :            Fx += dx * factor;
  8055   2.3780 :            Fy += dy * factor;
```

With the optimized algorithm and above minor optimizations in the computations, I was able to reduce the runtime to **10.063701 sec** on average. The deal breaker was mainly the optimized algorithm here and not the restructuring of the computations, but they still make a minor difference.

4.2.3 SoA and Vectorization:

The above improvements were quite limiting due to Array of Structs (AoS) data layout where related properties were separated by a stride of 6, leading to suboptimal cache utilization. This transformation places similar properties in contiguous memory locations, improving cache efficiency and enabling SIMD (Single Instruction, Multiple Data) parallelization. Therefore, I changed the data layout from AoS to Struct of Arrays (SoA).

The final code takes advantage of the vectorization to further improve computations. Key loops, such as processing particle interactions, and updating velocities and positions, are annotated with `#pragma GCC ivdep`. This directive tells the compiler to disregard assumed loop-carried dependencies, thereby enabling SIMD optimizations.

By vectorizing these loops, multiple elements (i.e., particle data) are processed concurrently in a single iteration, which reduces loop overhead and takes advantage of the processor's vector units. This vectorized approach combined with cache-friendly data layout reduced the runtime to **4.978700 sec** on average, and contributed quite alot to the performance gains observed in the final implementation.

4.3 Parallel Optimizations

Before reaching the optimized parallelized solution described in implementations section, I started with a basic version of the implementation. Building on that foundation, I experimented with various modifications to enhance thread synchronization and load balancing.

4.3.1 Pthreads

In the initial implementation, I created a fixed number of threads (equal to `nthreads`) at the start of the simulation, and the entire `nsteps` loop was executed within the worker function. In this design, each thread was assigned a fixed portion of the particle array. Before the end of each time step, a synchronization barrier was placed to ensure that all threads' local force computations were merged to perform the velocity and acceleration updates before moving to the next `nstep` iteration. The static work allocation led to load imbalance because some threads finished processing their assigned particles sooner than others.

To address the imbalance, I implemented a dynamic work allocation strategy. Instead of statically assigning a set of particles to each thread, I created `start_i` and `end_i` protected by a mutex so that each thread can dynamically fetch the next CHUNK of iterations to process. Once a thread finishes computing the forces for its assigned chunk, it immediately retrieves the next available block of particles. As before, each thread maintained its own private force arrays; after all threads completed their computations, a synchronization barrier ensured that these private arrays were merged before updating the velocities and accelerations.

Synchronization was handled using barriers at key points in the simulation:

- A barrier before entering the dynamic work loop to reset the shared index.
- A barrier immediately after the computation loop to ensure all threads have finished their force computations.
- A barrier after the reduction step to make sure all temporary force arrays were merged into one.
- A final barrier after the velocity and acceleration update loop, before moving on to the next time step.

In the final refinement, I merged the reduction loop with the velocity and acceleration update loop. This allowed me to remove one barrier, reducing the total number of barriers to three.

With this implementation, I was able to bring the overall runtime down to **0.72123 sec** on average. After tuning the CHUNK size, I found that 8 was a good number and the runtime averaged around **0.69113 sec**.

4.3.2 OpenMP

Initially, I tried to follow the pthreads-like approach in OpenMP implementation and defined parallel directive at the outermost loop and used private arrays to do the manual reduction. But this approach was unnecessary and the same result was easily achievable with just compiler directives.

The computationally intensive force calculation loop was parallelized using:

```
#pragma omp parallel for schedule(dynamic,8) reduction(+:tmp_fx[:N], tmp_fy[:N])
```

Dynamic scheduling was chosen over static and guided alternatives to minimize load imbalance, with experiments showing good performance at a chunk size of 8.

The inner loop over particles was explicitly vectorized using the directive:

```
#pragma omp simd
```

This allowed the compiler to utilize SIMD instructions, significantly improving arithmetic performance.

These combined optimizations ultimately led to a significant runtime reduction, achieving an average execution time of **0.67182 sec**.

4.4 Results

The following plots generated with a Python script (using `matplotlib`), present average execution times for different values of N and confirm the quadratic scaling behavior. The script runs the simulation for various N, extracts the timing data using regular expressions, and plots both the actual measurements and a theoretical $O(N^2)$ curve for comparison. The following plot in Figure 1 was created using nonoptimized serial code just to illustrate the $O(N^2)$ algorithmic complexity.

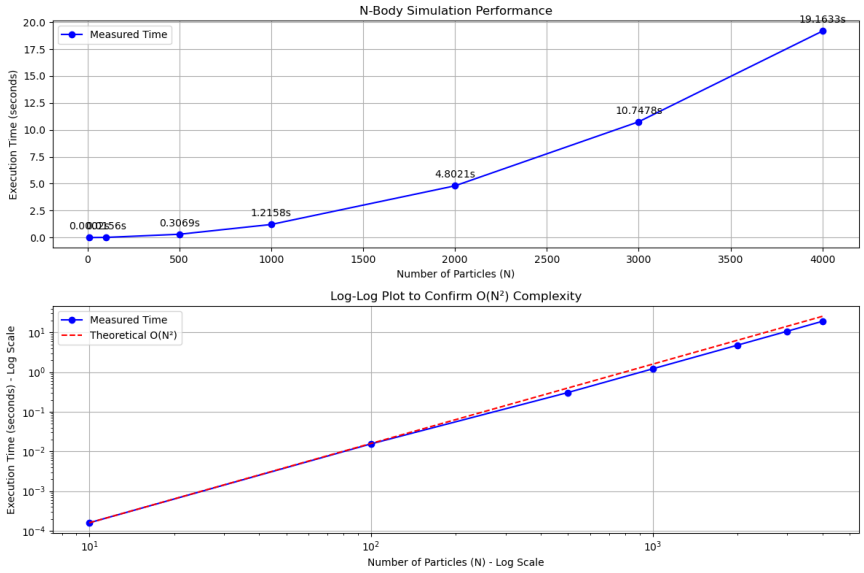

Figure 1: Execution time as a function of N.

After parallelization

To evaluate the performance improvements achieved through parallelization, I measured the speedup of the OpenMP version of the code. For these experiments, the executable built from the OpenMP implementation was used. The simulation was run in 200 time steps with two different problem sizes: $N = 2000$ and $N = 5000$.

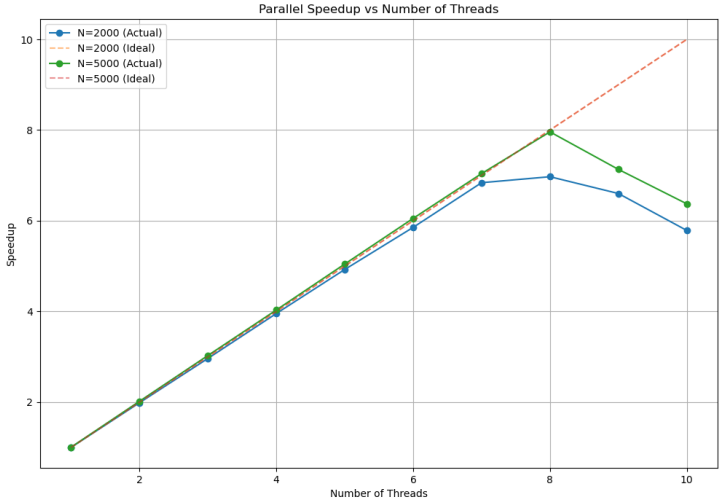

Figure 2: Parallel speedup vs Number of threads (2000 vs 5000 particles in 200 time steps)

Figure 2 shows two speedup plots: one for $N = 2000$ and one for $N = 5000$. In these plots, the number of threads is on the x-axis and the achieved speedup is on the y-axis. A dashed line represents the ideal speedup, where the speedup is equal to the number of threads.

For $N = 2000$, the speedup is nearly ideal up to 8 threads, with diminishing returns observed beyond that because there 8 cores on the processor used in these experiments. For $N = 5000$, the results are closer

to the ideal speedup curve. This confirms the expectation that larger problem sizes tend to scale better, as the additional computation per thread helps to mask the overhead of parallelization.

4.5 Discussion

Through a combination of algorithmic refinements, data layout improvements, and the use of vectorization, I was able to achieve a performance improvement in the serial implementation from nearly 20 seconds down to just over 5 seconds. This progression showed the significant impact that careful optimization of both algorithms and memory access patterns.

Building on the improvements achieved in the serial version, I used the optimized serial code as the base for the parallel implementations. Both the Pthreads and OpenMP versions were designed to maximize the amount of work performed per thread by carefully managing load balancing and synchronization. The Pthreads version achieved an average runtime of approximately 0.69 seconds. On the other hand, the OpenMP version not only simplified the code by removing the need for explicit thread management structures but also achieved excellent load balancing. This approach yielded an average runtime of about 0.67 seconds.

5 Conclusion

This report has presented the implementation and optimization of a gravitational N-body simulation. The integration of serial optimizations with parallel implementations led to significant performance gains in the simulation. Initially, through algorithmic refinements, data layout improvements, and the application of vectorization techniques, the serial execution time was reduced from nearly 20 seconds to just over 5 seconds. Building upon this optimized serial foundation, parallel implementations using Pthreads and OpenMP were developed. Both approaches highlighted the importance of dynamic scheduling and load balancing to maximize thread utilization and minimize synchronization overhead. As a result, the Pthreads implementation achieved an average runtime of approximately 0.69 seconds, while the OpenMP implementation further reduced the runtime to about 0.67 seconds. These results demonstrate how effective serial optimizations and parallelization strategies are in achieving good performance.

6 References

During the implementation of this assignment, the following materials were studied and examined repeatedly:

- Laboratory materials: Lab04, Lab05, Lab06, Lab07, Lab08, Lab09, Lab10, and Lab11.
- Lecture slides: SerialCodeOpt.pdf, Pthreads part 1, part 2, and OpenMP Compiler directives.
- Optimizing Software in C++: An Optimization Guide for Windows, Linux, and Mac Platforms by Agner Fog.
- An Introduction to Parallel Programming by Peter Pacheco.