Uppsala University

High Performance Programming

1TD062 62013 VT2025

Assignment 4: Parallelization

Author:
Abhinav Ramalingam

March 16, 2025

Contents

1 The Problem

The task is to simulate the motion of N particles in a gravitational system over tiny, discrete time steps. Each particle has a position, velocity, and mass, and their interactions follow Newton's law of gravitation. The initial state is read from a binary file, and for each time step, the net gravitational force on each particle is computed, leading to updates in acceleration, velocity, and position. The final state is written to an output file after a specified number of steps. The simulation runs in two dimensions and has a computational complexity of $O(N^2)$ due to pairwise force calculations.

2 The Solution

2.1 Data Structure

The core data structure used is the `struct Particle` data structure, which represents each body in the simulation. It includes:

- **Position:** `x, y` (spatial coordinates of the particle)
- **Mass:** `mass` (Constant throughout the simulation)
- **Velocity:** `vx, vy` (updates at each time step based on acceleration)
- **Brightness:** `brightness` (A property stored but not actively used in calculations)

Additionally, another data structure called `ThreadData` is used to maintain the information like intervals, pointer to the overall Particle system and other variables needed so that each thread can process just the particles, or 'rows' of computation it is assigned to. `ThreadData` includes:

- **ParticleData pointer:** Points to the simulation's main data
- **Shared velocity pointers (`vx_sp` and `vy_sp`):** Pointers to arrays (`vx_shared` and `vy_shared`) that store the shared 'velocity changes' of particles which is in turn an accumulation of local velocity changes in each thread.
- **DeltaT:** Timestep Size

- **N:** Number of Particles
- **Start and End:** The range of indices that the thread is responsible for processing

2.2 Algorithm

Algorithm 1 Thread Function for N-Body Simulation (`compute_forces`)

```
Input: Thread data (including particle data, velocity arrays, start, and end indices)
Output: Updated velocity arrays for the thread's range
Extract input data (start, end, particles, velocity arrays, etc.)
Initialize local velocity arrays: $local\_vx$, $local\_vy$ (set to zero)
for each particle $i$ from start to end do
    Set local_force$_i$ to zero
    for each particle $j$ (where $i \neq j$) do
        Calculate gravitational force between particles $i$ and $j$
        Add the force to local_force$_i$
    end for
    Update particle $i$'s local velocity using local_force$_i$
end for
Lock(mutex):
for each particle $i$ do
    Add local velocity changes to shared velocity arrays $vx\_sp[i]$, $vy\_sp[i]$
end for
Unlock(mutex):
```

Algorithm 2 Main Function for Parallel N-Body Simulation

```
Input: Initial particle data, number of timesteps, number of threads, timestep size
Output: Updated particle data written to output file
Initialize particles and allocate memory for velocity arrays
Read particle data from input file
Initialize shared velocity arrays $vx\_shared$, $vy\_shared$
Initialize threads array for parallel computation
for each timestep $t$ from 1 to num_steps do
    Reset shared velocity arrays $vx\_shared$, $vy\_shared$ to zero
    Calculate number of rows taken by each thread based on thread ID:
    rows $= 2 * N * (t + 1)/(n\_threads * (n\_threads + 1))$
    for each thread $t$ from 0 to n_threads - 1 do
        Calculate thread's range:
        if $t == n\_threads - 1$ then
            allocate rows till $N - 1$
        else
            allocate rows based on above formula
        end if
        Set up thread data with particle pointers, velocity pointers, range (start, end)
        Create a new thread to execute compute_forces with the thread data
    end for
    for each thread $t$ from 0 to n_threads - 1 do
        Wait for thread $t$ to finish using pthread_join
    end for
    for each particle $i$ do
        Update particle $i$'s velocity and position using $vx\_shared[i]$, $vy\_shared[i]$
    end for
    Reset shared velocity arrays for the next timestep
end for
Write updated particle data to output file
```

The solution is presented in Algorithm 1 (Thread Function) and 2 (Main function which creates threads). The program follows a parallelized Brute-Force Approach (*$O(N^2)$*, as indicated in Figure 1), where distribute fewer rows to earlier threads and more rows to later threads rather than an equal partitioning. This dynamic allocation ensures better load balancing, as each row corresponds to a particle, and each row have one less number of computations than the previous row, due to Newton's Third Law.

Alternative approaches, such as the Barnes-Hut Algorithm (*O(N log N)*), could significantly reduce computational complexity by approximating distant particle interactions using a quadtree. However, these require additional data structures and implementation complexity. Rather than optimizing upfront, we prioritized correctness by first implementing a simple brute-force version.

Alternative parallelization strategies were also considered. Instead of dividing the outer loop over particles (i) among threads, we could have parallelized the inner loop (j), dividing each row of computations to different threads. Another possibility was using a dynamic task-based approach where threads pick up individual rows as they finish processing previous ones. However, these methods are more challenging, requiring either more frequent mutex locking (which would hurt performance) or more advanced task division/sync techniques. Given these complexities, we opted for a simpler and more elegant parallelization strategy that ensures correctness while performing efficient load balancing.

This allows for a step-by-step approach to serial optimization followed by two methods of parallelization, namely **PThreads** and **OpenMP**, making it easier to observe speed-up effects as different optimizations are introduced progressively.

2.3 Testing and Benchmarking the Solution

The mathematical correctness of our solution was tested with the given `compare_gal_files.c` program which takes in two gal files and a size value as arguments and compares whether all the physical values (mass, velocities, positions, brightness) match between the two files. A max difference of 0.0000 indicated *SUCCESS*.

For measuring the performance of our program, we used the `omp_get_wtime()` function to get the current time at 4 instants - start of the program, after input file is copied into a Data Structure in the program, after the processing has been done for `timesteps` time intervals, and after the processed Data Structure is written back into an output file created called `result.gal`.

Thus, the results are presented in a table comparing execution times across different implementations. The table includes runtime measurements with and without compiler flags for the following versions:

- Non-optimized code
- Serial optimized code
- Parallelized with `PThreads` and `OpenMP` for different number of threads

After each serial optimization and parallelization attempt, `compare_gal_files.c` was used to verify that the results remained consistent with the original implementation.

Here, only the middle processing part (between the 2nd and 3rd instance of recording clock value) is taken in account since it was observed that reading and writing contributes to very little runtime (in the order of 10^{-3} seconds).

3 Performance and Discussion

The base program was executed without any optimizations. The parameters used for this run were:

- **Number of particles (N)**: 5000
- **Number of time steps (nsteps)**: 100
- **Time step size (delta_t)**: 0.00001
- **Graphics flag**: 0 (no graphical output)

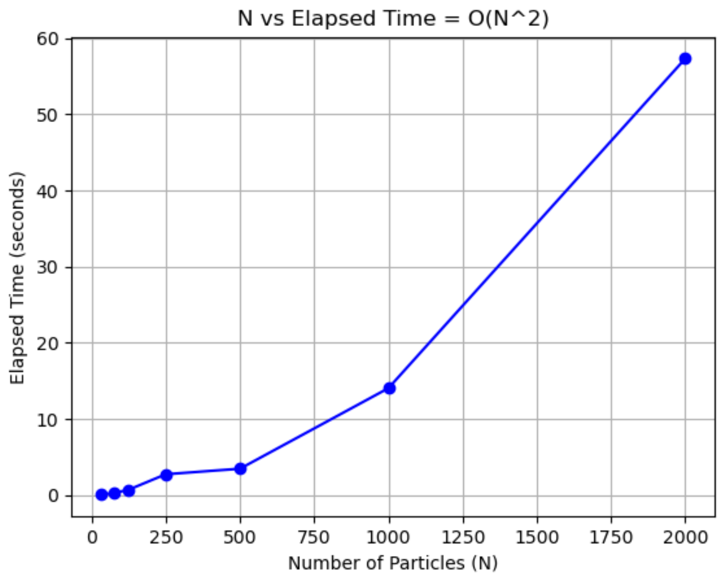


Figure 1: Base program performance graph

3.1 Optimization 1: Removing Function calls

A profiler known as `gprof` user used, and the observed results show that:

- **72.56%** time taken by `gravitational_force` – **29.30 seconds**
- **22.39%** time taken by `gravitational_force_net` – **9.04 seconds**

Also, examining the call graph gives us an information that:

- `gravitational_force_net` – called **300000** times
- `gravitational_force` – called **899700000** times

This shows that there is high overhead involved in a function call. Since the program is a stand-alone one, removing those function calls (while still maintaining readability through proper comments, indentation, dividing into section, etc.) gives a speed-up in code execution.

3.2 Optimization 2: Reducing Expensive Computations

Our program can be further optimized by reducing various computations or replacing them by their lighter counterparts. The followed optimizations were implemented:

1. Replacing the `pow` function call without multiplication.
2. Replacing (* 2) with $\ll 1$, (* 6) with $(\ll 1) + (\ll 2)$
3. Removing redundantly calculated values
4. Declaring unchanging variables as constant for efficiency
5. Moving Loop invariant computations outside loop, for e.g., moving i-dependent values outside j-loop.

3.3 Optimization 3: Using Newton's Third Law

According to Newton's Third Law, the force exerted on particle j by particle i is equal in magnitude but opposite in direction to the force exerted on particle i by particle j. Leveraging this property, we can represent the forces as an i-j matrix and compute only the upper triangular portion. The corresponding lower triangular values can then be obtained by simply negating the computed forces, effectively reducing the number of force calculations by half.

This approach leads to increased dynamic heap memory allocations because, when calculating Force-on-i-by-j, we need to remember Force-on-j-by-i too to avoid recomputation. This induces overhead due to the need of creating **auxilliary arrays**. The program still speed up, although not exactly by half.

Another benefit of this optimization is that after calculating all the j-loop values of a particular i-th particle, it is not necessary to revisit computation for that same particle as everything is already precomputed by then. Because of that, an extra loop for updating the values separately s not required.

3.4 Optimization 4: Mass cancellations

In the force computation, mass was multiplied and the same mass was divided while computing acceleration. This redundancy was removed. Along with this optimization, the part of acceleration computation which was loop invariant, was moved outside the loop.

3.5 Optimization 5: Structure-of-Arrays

Additional optimizations were applied to improve performance. First, the **Structure of Arrays (SoA)** format was used instead of the Pure Array format which was used before. This improves **cache utilization**.

Additionally in this optimization step, using auxiliary arrays (as in Optimization 3) were avoided by storing and directly updating the particle properties (such as velocity, etc.) in the same arrays during each timestep, rather than calculating and storing intermediate results in separate arrays.

3.6 Parallelization 1: PThreads

In `PThreads`, thread management is handled manually, requiring explicit allocation of work to each thread, manual thread creation and synchronization mechanisms, and explicit thread joining. This adds complexity compared to `OpenMP`, where parallelism can be achieved with simpler pragmas.

In the simulation, the ‘changes’ in velocity are independent of previous velocities. However, the ‘changes’ in position depend on the computed forces, which in turn are influenced by the previous positions, as gravity varies with distance. As a result, each thread maintains its own array, accumulating the local ‘velocity changes’. After computing all interactions, these local velocity changes are safely added to a shared ‘velocity change’ array using mutexes. Once all threads finish their computations for a timestep, the shared velocity changes are applied to the main particle data structure, updating the global velocities. Finally, the positions are updated using the new velocities, and the shared velocity change arrays are reset for the next set of local ‘velocity changes’.

As discussed in Algorithms 1 and 2, our process follows newton’s laws where each row has one less computation than the previous row, requiring the need for a more load-balanced approach rather than just dividing equal rows of computations between threads. The total number of interactions follows the summation $(N-1)+(N-2)+\cdots+1=\frac{(N-1)N}{2}$, meaning earlier rows contribute significantly more computations than later ones. Thus, we allocate rows dynamically based on thread ID t using the formula:

$$\text{rows}_t = \frac{2N}{n_{\text{threads}}(n_{\text{threads}}+1)}(t+1) \tag{1}$$

This ensures that the workload remains approximately balanced across all threads.

3.7 Parallelization 2: OpenMP

Unlike `PThreads`, `OpenMP` simplifies parallelization by automatically handling thread creation, management, workload distribution, and thread deletion through the use of simple pragmas. This eliminates the need for manually assigning work to threads, explicitly creating and synchronizing threads, and ensuring thread joining.

In `PThreads`, we manually manage local 'velocity changes' in separate arrays for each thread, which are then accumulated onto a shared 'velocity change' variable. This requires using mutexes to safely update the global shared variable. However, in `OpenMP`, the `reduction` clause handles the creation and *safe* accumulation of local data structures and combining the results into the shared data structure with a `+` operator.

To manage workload distribution effectively, OpenMP offers different **scheduling policies**: **Static**, **Dynamic**, and **Guided**. Static scheduling divides the work into fixed chunks. Dynamic scheduling assigns smaller chunks to threads as needed, adjusting dynamically to the workload. Guided scheduling works similarly but adjusts chunk sizes in a decreasing manner as work progresses. Based on our experiments, we found that **Dynamic scheduling** provided the shortest runtime. This is likely because static scheduling did not load balance well, while guided scheduling, introduced some overhead due to the gradual management of chunk sizes.

3.8 Results comparison

In The following GCC compiler flags are used for optimization:

```
gcc -O3 -march=native -ffast-math -o galsim galsim.c -lm
```

- `-O3` : Enables aggressive optimizations for better performance.

- `-march=native` : Optimizes the code for the specific CPU architecture of the compiling machine.
- `-ffast-math` : Enables aggressive floating-point optimizations, allowing the compiler to break strict IEEE floating-point rules for better performance.

These flags collectively enhance execution speed by optimizing memory access, instruction scheduling, and floating-point operations.

As discussed in Section 2.3, the following Tables 1, 2, and 3 represent the results obtained after every optimization and parallelization attempt (for 5000 particles after 100 timesteps).

Additionally, Figure 2a and Figure 2b shows the plot for runtime comparison for 3000-particles, 100-timesteps and 5000-particles, 100-timesteps respectively.

Optimization	**Runtime without Flags**	**Runtime with Flags**
Non-Optimized	580.419317	57.568249
Serial Optimized	43.999377	22.218220

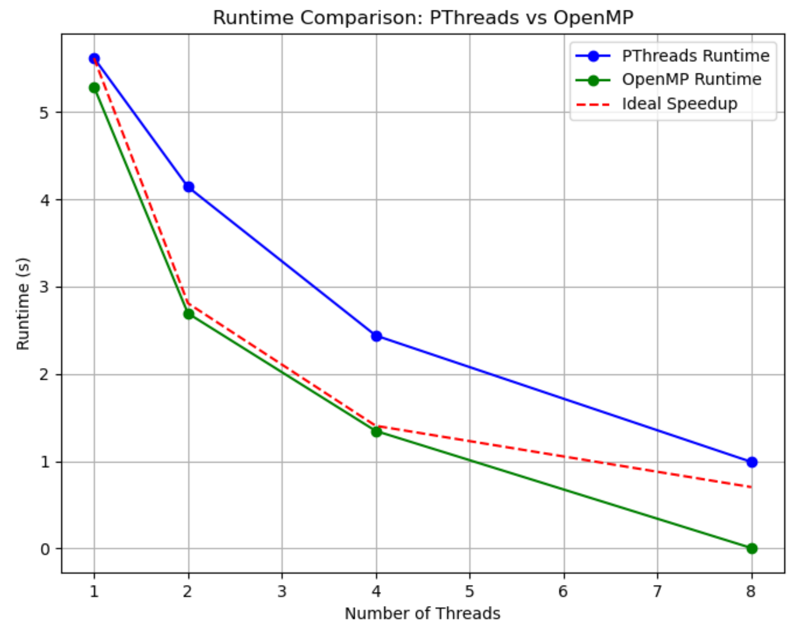
Table 1: Runtime comparison for serial optimizations

Threads	**Runtime without Flags**	**Runtime with Flags**
1	54.093419	16.433368
2	30.483780	9.310231
4	18.150312	5.688957
8	10.305372	3.527355

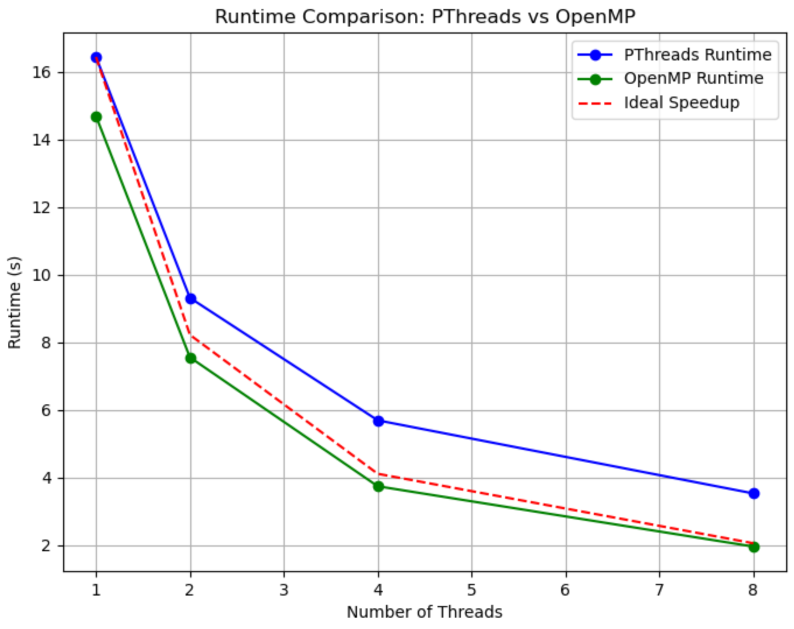
Table 2: Runtime comparison for `PThreads` parallelization

Threads	**Runtime without Flags**	**Runtime with Flags**
1	61.991293	14.685363
2	31.057286	7.551738
4	15.547214	3.739856
8	7.874317	1.956039

Table 3: Runtime comparison for `OpenMP` parallelization



(a) Runtime comparison for 3000 particles, 100 timesteps.



(b) Runtime comparison for 5000 particles, 100 timesteps.

Figure 2: Runtime comparison between PThreads and OpenMP for different particle counts.

Note to Consider:

`OpenMP` offers a significant advantage in terms of ease of implementation, allowing for efficient parallelization with minimal effort through simple pragma directives. This automated nature allows for quicker optimization compared to `PThreads`, which, while providing greater control over thread management, requires more manual work in terms of thread creation, synchronization, and load balancing. Although `PThreads` can offer finer control and optimization potential, `OpenMP` is generally preferred for its efficiency in achieving parallelization with less complexity and effort.

References

[1] OpenAI, *ChatGPT*, `https://chat.openai.com`. (Used for automating trivial calculations and code reformatting, but not for code generation.)

[2] Uppsala University, *Lecture 4: Serial Code Optimization*, `https://uppsala.instructure.com/courses/95091/pages/lecture-notes?module_item_id=1285027`. (Used for concepts of Serial Optimization)

[3] Uppsala University, *Lecture 5: PThreads 1*, `https://uppsala.instructure.com/courses/95091/pages/lecture-notes?module_item_id=1285027`. (Used for concepts of PThreads)

[4] Uppsala University, *Lecture 6: PThreads 2*, `https://uppsala.instructure.com/courses/95091/pages/lecture-notes?module_item_id=1285027`. (Used for advanced concepts of PThreads)

[5] Uppsala University, *Lecture 7: OpenMP Compiler Directives*, `https://uppsala.instructure.com/courses/95091/pages/lecture-notes?module_item_id=1285027`. (Used for OpenMP compiler directives)

[6] Uppsala University, *Lecture 8: OpenMP Case Studies*, `https://uppsala.instructure.com/courses/95091/pages/lecture-notes?module_item_id=1285027`. (Used for OpenMP case studies)