TÉCNICO LISBOA

Parallel and Distributed Computing

MEEC

OpenMP

Authors:

Miguel Ferreira (113289) miguel.r.ferreira@tecnico.ulisboa.pt
José Rodrigues (113234) jose.p.rodrigues@tecnico.ulisboa.pt
Miguel Pereira (103254) miguelquinapereira@tecnico.ulisboa.pt

GROUP 26

2024/2025 – 2 Semester, P1

1 Introduction

The goal of this project is to develop and analyze a particle simulation system using parallel and distributed computing techniques. The simulation models the movement of particles in a 2D space where they interact based on gravitational forces.

This report focuses on the first two stages of the project:

- **Serial Implementation** – A baseline version of the simulation written in C, this implementation provides a reference for evaluating the benefits of parallelization.
- **OpenMP Implementation** – A parallel version of the simulation using **OpenMP** to leverage shared-memory multiprocessing.

By comparing these two implementations, we aim to evaluate the impact of parallelization on execution time and scalability while maintaining accuracy in the simulation results.

2 Serial Implementation

We aimed to implement the most efficient serial solution possible for particle simulation. To optimize the code, instead of checking interactions between all particles, we defined the cell for each particle and then traversed all the cells to check for collisions within the particles' cells., reducing the number of interactions to be checked.

Furthermore, we optimized the force calculation by updating the opposite forces of the particles using the third Newton law. The gravitational forces between two particles are considered symmetric, and we update the forces as the opposite of the calculated force for each particle.

2.1 Updates in serial implementation

Compared to the previous serial version, we corrected the force update to ensure proper symmetry. Instead of subtracting the already updated forces of par[px], we now directly subtract fx and fy for par[py].

In the updated version, we introduced `n_par`, which directly tracks the number of particles in each cell. This eliminates the need for a search loop, improving efficiency and simplifying the code, and it's important because if we want to parallelize the particles that are within the cell, we need to Know how many are there to use the pragmas.

Additionally, we have now correctly implemented collision detection for cases involving triple collisions.

3 OpenMP Implementation

3.1 Parallelization with OpenMP

The simulation involves a large number of particles that interact with each other, making it a computationally intensive problem. To leverage multi-core processors, OpenMP directives were used to parallelize critical sections of the code.

3.2 Type of Decomposition Used

We employed spatial decomposition, where the simulation space was divided into a uniform grid. Each particle was assigned to a specific grid cell based on its position, and parallelization was performed per grid cell. This approach facilitated efficient parallelization by reducing the number of particle-pair interactions. We tried to parallelize the particles too, but the performance decreased drastically. For future implementations, particularly using distributed memory approaches like MPI, dividing particles across multiple cores or nodes could offer better scalability and performance.

3.3 Synchronization Concerns and Motivation

We encountered some difficulties with our grid calculation function due to synchronization issues, particularly related to managing concurrent updates to shared data. Initially, we attempted to resolve these issues using the OpenMP atomic directive. However, this proved ineffective because the critical section was more extensive than what the atomic directive could efficiently handle.

To address this, we transitioned to using explicit locks. Although this approach required allocating and managing a lock for each cell of the grid, it substantially improved performance after proper implementation. Consequently, this solution not only effectively resolved the synchronization issues but also enhanced the overall performance of our simulation significantly.

We also used in other parts of the code other strategies such as dynamic scheduling and nowait clauses to minimize unnecessary synchronization.

In addition, atomic and critical operations were used to ensure safe updates to shared variables, particularly in collision count and calculating X, Y and M from center of mass. This prevented race conditions and ensured data integrity, although at the cost of some performance overhead. Here we have an example of how we used the atomic variable for counting collisions.

```
#pragma omp atomic
collision_count++;
```

3.4 Load Balancing

Load balancing was achieved using dynamic scheduling to distribute work among threads more efficiently. The key improvements were the following:

- Using `schedule(dynamic, 10)` to distribute workload dynamically and prevent load imbalance.

```
    #pragma omp parallel
    #pragma omp for schedule(dynamic, 10) private(grid_x, grid_y) nowait
        for(int i = 0; i< num_particles; i++){

            par[i].Fx = 0;
            par[i].Fy = 0;
            if (par[i].alive == 0)
                continue;
                ...
```

- Parallelizing nested loops with `collapse(2)` to improve workload distribution across threads.

```
#pragma omp parallel
#pragma omp for private(delta_x, delta_y) schedule(dynamic, 10) collapse(2) nowait
    for(int idx_x = 0; idx_x < grid_size; idx_x++){
        for(int idx_y = 0; idx_y < grid_size; idx_y++){

            for(int j = 0; j<cm[idx_x][idx_y].n_par; j++){

            int px = cm[idx_x][idx_y].par_index[j];

            ...
```

- Reducing the number of synchronization points to allow independent thread execution.

These optimizations helped improve performance by ensuring a more even distribution of computational tasks across available threads.

3.5 Performance Results

The tables presented provide performance metrics for parallel computing using different thread counts across three example simulations. The first table showcases the execution times in seconds for serial and parallel processing, with varying thread counts (1, 2, 4, and 6). To run these examples, we used the P2 computer from Lab2, and in all the examples run, the results obtained were correct.

Example	Serial	1 Thread	2 Threads	4 Threads	6 Threads
parsim 3 5000 50 1000000 500	731.9s	869.9s	469.6s	283.8s	276.8s
parsim -11 3500 20 500000 10	128.4s	148.8s	72.5s	40.7s	40.6s
parsim 3 5000 50 1000000 300	439.9s	522.1s	280.6s	169.4s	168.2s

Table 1: Serial and Parallel Times in seconds

The second table illustrates the speedup achieved for each example when using multiple threads compared to the serial execution.

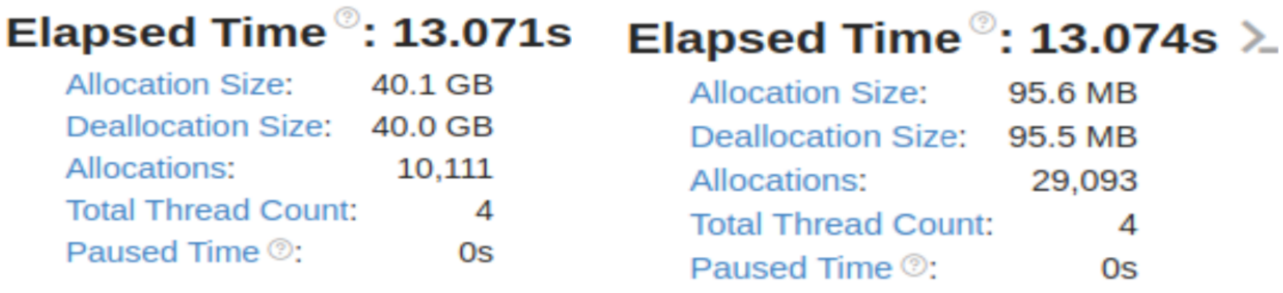
Example	1 Threads	2 Threads	4 Threads	6 Threads
parsim 3 5000 50 1000000 500	0.84	1.56	2.58	2.64
parsim -11 3500 20 500000 100	0.86	1.77	3.15	3.16
parsim 3 5000 50 1000000 300	0.84	1.57	2.60	2.61

Table 2: SpeedUp for each example with different threads

These comparisons highlight the efficiency improvements gained through parallel processing for each specific example. As the number of threads increased, execution time decreased, leading to improved speedup. However, beyond four threads, the scalability gains became minimal, suggesting diminishing returns. We are not achieving the ideal speedups because of false sharing. This invalidates the cache lines and the program runs slower.

3.6 Memory usage

We also optimized the memory usage, in the example ./parsim 1 5000 100 1000000 100 we were using $40GB$ of memory and now we are using $95.6MB$.



```
Elapsed Time: 13.071s
    Allocation Size:       40.1 GB
    Deallocation Size:     40.0 GB
    Allocations:            10,111
    Total Thread Count:          4
    Paused Time:                0s

Elapsed Time: 13.074s
    Allocation Size:       95.6 MB
    Deallocation Size:     95.5 MB
    Allocations:            29,093
    Total Thread Count:          4
    Paused Time:                0s
```

Figure 1: Bad Memory usage vs Good Memory Usage

4 Conclusion

In conclusion, our work demonstrated that careful synchronization and appropriate load balancing strategies are critical to achieving optimal performance in parallel simulations. By effectively implementing spatial decomposition and dynamically scheduled workloads, we significantly reduced execution times compared to the serial version. Explicit locks provided the necessary synchronization with minimal performance overhead, resulting in a scalable and efficient parallel simulation. Future work may explore further optimization strategies, like MPI.