University of Camerino

School of Science and Technology

Master Degree in Computer Science

Parallel and distributed Programming: Parallelization of the N bodies problem

Supervisor

Prof. Andrea Polini

Students

Francesco Finucci

Academic Year 2024-2025

Contents

Part I

PROLOGUE

1 Introduction

The N-body problem, a cornerstone of computational physics and astrophysics, aims to predict the motion of a system of interacting bodies under mutual gravitational forces. Due to its computational complexity, solving this problem efficiently requires advanced numerical methods and high-performance computing techniques. In this project, we address this challenge by developing a parallel algorithm using the C programming language and the Message Passing Interface (MPI) framework. The goal is to estimate the dynamics of the N-body problem through the Runge-Kutta 4th-order (RK4) integration method, renowned for its balance between accuracy and computational cost.

Parallelization is a crucial strategy for addressing the computational demands of the N-body problem. By distributing the workload across multiple processors, we aim to significantly reduce the execution time while maintaining accuracy. MPI provides a robust and scalable platform for implementing this distributed computation, enabling efficient communication between processes in a high-performance computing environment.

To evaluate the quality of the developed algorithm, we utilize key performance metrics such as **efficiency** and **speedup**. Efficiency measures the effectiveness of resource utilization across multiple processors, while speedup quantifies the improvement in computation time relative to a sequential implementation. These metrics provide valuable insights into the scalability and performance of the parallel algorithm, guiding further optimizations.

This project not only contributes to advancing computational solutions for the N-body problem but also demonstrates the power of parallel computing techniques in addressing complex scientific problems. The results of this work can be extended to other domains where high-dimensional numerical simulations are essential, such as fluid dynamics, molecular dynamics, and cosmological simulations.

Part II

DISTRIBUTION STRATEGY & IMPLEMENTATION

2

Chosen Parallelization Strategy

When working on the parallel algorithm for the N-body problem using the Runge-Kutta 4th-order method, I had to think about how to split up the tasks among the processes. I looked at two main ways of doing this, and in this chapter, I'll explain what they are and why I picked one over the other.

2.1 DYNAMIC WORKLOAD DISTRIBUTION

The first option I considered was having each process communicate with a master process to get tasks one at a time. Here's how it works:

1. A process asks the master for a task.
2. The master gives the process a task to work on.
3. The process finishes the task and tells the master it's done.
4. This repeats until all the tasks are finished.

The good thing about this approach is that it's flexible. If some tasks take longer than others, the master can adjust by giving more work to processes that finish quickly. But there's a big downside: all this back-and-forth communication between the processes and the master takes time. In this project, each task takes about the same amount of time to compute, so the extra communication would just slow everything down without adding much benefit.

2.2 STATIC WORKLOAD DISTRIBUTION

The second option was to divide the tasks equally among the processes right from the start. In this setup:

1. The master splits the tasks into equal parts and gives each process its share.
2. Each process works on its assigned tasks without needing to check back with the master.
3. When a process finishes its tasks, it sends the results to the master.

This approach reduces communication to the bare minimum. Each process only talks to the master twice: once to get its tasks and once to report the results. Because all the tasks take about the same amount of time, dividing the workload evenly ensures that every process has the same amount of work to do. This keeps everything running smoothly and efficiently.

2.3 RATIONALE FOR THE CHOSEN APPROACH

I decided to go with the static workload division because it's the best fit for this project. Since all the tasks take about the same time to compute, there's no real need for dynamic load balancing, which is the main advantage of the first approach. If I'd used the dynamic method, the constant communication between the processes and the master would have added a lot of overhead and slowed things down. That's not worth it when the tasks are already balanced.

With the static method, the processes can get straight to work without wasting time on extra communication. This makes the algorithm more efficient overall. It's also easier to implement because the processes don't need to keep checking in with the master. They just get their tasks, do the work, and report back when they're done.

In the end, the static workload division worked well because it matched the characteristics of the problem. By keeping the communication overhead low and taking advantage of the uniform task times, this approach made the algorithm faster and simpler to manage.

3 Project explanation

In this chapter, I'll go over the code I wrote for solving the N-body problem using the Runge-Kutta 4th-order method. I'll explain how the code is structured, what each part does, and how it all comes together to solve the problem. My goal is to break it down in a way that's easy to follow, so you can understand both the logic behind the solution and how the parallelization was implemented.

I'll start by walking through the main sections of the code, including how the data is set up and how the tasks are distributed among the processes. Then, I'll dive into how the Runge-Kutta method works in this context and how the results are collected at the end. By the end of this chapter, you should have a clear picture of how the solution works and how everything fits together.

3.1 CODE ANALYSIS

In the following code section we initialize the bodies randomly and then copy it inside the `all_bodies` variable that will be later shared between all of the ranks.

We then calculate the distributions, so the number of bodies to share and the starting index from which the array will be divided. Using this method, at most there will be processes that handle 1 body more than the others.

```c
if (rank == 0) {
    // Initialize the system with random bodies
    Body initial_bodies[NUM_BODIES];
    initialize_random_bodies(initial_bodies, NUM_BODIES, 10)
;

    // Copy the initial bodies to the all_bodies array
    memcpy(all_bodies, initial_bodies, NUM_BODIES * sizeof(
Body));

    // Calculate distribution
    int base_count = NUM_BODIES / size;
    int remainder = NUM_BODIES % size;

    // The distribution is calculated as follows:
    // - Each process will handle base_count bodies
    // - The first remainder processes will handle one
additional body
    // - The displacements are calculated based on the
sendcounts
    // - The displacements are the starting index of the
bodies for each process
```

```c
        // - The sendcounts are the number of bodies each process will handle
        displs[0] = 0;
        for (int i = 0; i < size; i++) {
            sendcounts[i] = base_count + (i < remainder ? 1 : 0);
            if (i > 0) {
                displs[i] = displs[i-1] + sendcounts[i-1];
            }
            printf("Process %d will handle %d bodies starting at index %d\n",
                   i, sendcounts[i], displs[i]);
        }
    }
```

Listing 3.1: Initialization and distribution

The following is the function that randomly initializes the bodies. We use a seed so that we can add determinism inside the random process

```c
// Function to initialize a given number of bodies randomly
void initialize_random_bodies(Body *bodies, int num_bodies, unsigned int seed) {
    // Seed the random number generator with the provided seed
    srand(seed);

    for (int i = 0; i < num_bodies; i++) {
        // Generate a random mass between 1e20 and 1e30 kg
        bodies[i].mass = random_double(1e20, 1e30);

        // Generate random positions within a cubic region of size 1e13 meters
        bodies[i].position[0] = random_double(-1e13, 1e13);
        bodies[i].position[1] = random_double(-1e13, 1e13);
        bodies[i].position[2] = random_double(-1e13, 1e13);

        // Generate random velocities within a range of approx 5e4 meters per second
        bodies[i].velocity[0] = random_double(-5e4, 5e4);
        bodies[i].velocity[1] = random_double(-5e4, 5e4);
        bodies[i].velocity[2] = random_double(-5e4, 5e4);
    }
}
```

Listing 3.2: Random generation of the bodies

The `bodies` variable is defined using a custom MPI datatype

```c
// Define the Body struct
// This struct represents a body with mass, position, and velocity
typedef struct {
    double mass;
    double position[3];
    double velocity[3];
} Body;

// Create MPI datatype for Body struct
MPI_Datatype create_body_datatype() {
    // Create MPI datatype for Body struct
    MPI_Datatype body_type;
    // Define the blocklengths, types, and offsets for the struct members
    int blocklengths[] = {1, 3, 3};
    MPI_Aint offsets[3];
    MPI_Datatype types[] = {MPI_DOUBLE, MPI_DOUBLE, MPI_DOUBLE};

    // The three offsets are the addresses of the mass, position, and velocity members
    offsets[0] = offsetof(Body, mass);
    offsets[1] = offsetof(Body, position);
    offsets[2] = offsetof(Body, velocity);

    // Create the struct type
    MPI_Type_create_struct(3, blocklengths, offsets, types, &
    body_type);
    MPI_Type_commit(&body_type);

    return body_type;
}
```

Listing 3.3: MPI datatype definition

The number of bodies to be handled by each process is then scattered to all ranks inside the `local_count` variable, that is different for each rank

```
    // Scatter sendcounts to all processes
    MPI_Scatter(sendcounts, 1, MPI_INT, &local_count, 1, MPI_INT
    , 0, MPI_COMM_WORLD);
```

Listing 3.4: Scatter of the counts

Finally, we enter the main loop computation. We allocate the `local_bodies` array that will contain the sub-array for each rank. We also start the timer using `MPI_Wtime`.

The computation is then divided into 4 different steps:

1. **Broadcasting Positions:** At the start of each iteration, the current positions of all the bodies are shared with all the processes. This is done using `MPI_Bcast`, which ensures that every process has the updated positions it needs to perform calculations.

2. **Distributing Work:** Next, the workload is divided among the processes using `MPI_Scatterv`. This function sends a specific subset of the bodies to each process based on the predefined send counts and displacements. This way, each process only works on its assigned portion of the problem.

3. **Updating Positions:** Each process then computes the new positions of its assigned bodies using the Runge-Kutta 4th-order method. The `update_body_positions` function takes care of these calculations, using the positions of all the bodies to determine the forces and update the local bodies.

4. **Gathering Results:** After the local computations are done, the updated positions of the bodies are sent back to the master process using `MPI_Gatherv`. This step collects the results from all the processes and combines them into the `all_bodies` array, which contains the positions of all the bodies.

5. **Repeating the Process:** The loop then moves to the next time step, repeating the process until all the steps are completed.

At the end of the loop, the end computation time is measured using `MPI_Wtime` again, which records the time it took for the entire simulation to run. We then use this for analyzing the performance of the algorithm.

```
    // Allocate local array used by each process to store bodies
    Body *local_bodies = malloc(local_count * sizeof(Body));

    double start = MPI_Wtime();
```

```c

    // Main simulation loop
    for (int step = 0; step < NUM_STEPS; step++) {
        // Broadcast the updated positions to all processes for the next iteration
        MPI_Bcast(all_bodies, NUM_BODIES, body_type, 0, MPI_COMM_WORLD);

        // Scatter bodies to processes
        // Each process will receive a subset of the bodies based on the sendcounts and displacements
        MPI_Scatterv(all_bodies, sendcounts, displs, body_type,
                     local_bodies, local_count, body_type,
                     0, MPI_COMM_WORLD);

        // Update positions using runge kutta integration to approximate new positions
        update_body_positions(local_bodies, local_count, all_bodies, rank);

        // Gather updated bodies from all processes inside of all_bodies
        MPI_Gatherv(local_bodies, local_count, body_type,
                    all_bodies, sendcounts, displs, body_type,
                    0, MPI_COMM_WORLD);
    }

    double end = MPI_Wtime();
```

Listing 3.5: Main loop

Inside the `update_body_positions` method we use a brute force approach. We take each body inside the `local_bodies` array and we compate it with each body inside `all_bodies`. We then compute the acceleration of the local body w.r.t. the compared body and finally we use all of the accelerations to use Runge-Kutta's integration method to update the positions

```c
// Update the positions of all bodies in the simulation
void update_body_positions(Body *local_bodies, int local_count, Body *all_bodies, int rank) {
    // Compute the net acceleration on each body
    // We use the local bodies to update the positions with all other bodies
    for (int i = 0; i < local_count; i++) {
        double net_acceleration[3] = {0};
        for (int j = 0; j < NUM_BODIES; j++) {
            double acceleration[3];
            compute_acceleration(&local_bodies[i], &all_bodies[j], acceleration);
            for (int k = 0; k < 3; k++) {
                net_acceleration[k] += acceleration[k];
            }
        }
        // Update the position of the body using the Runge-Kutta integration
        runge_kutta_step(&local_bodies[i], net_acceleration);
    }
}
```

Listing 3.6: Brute force approach to update the positions

The following snippet contains the code for computing the accelerations and runge kutta

```c
// Compute the acceleration between two bodies
void compute_acceleration(Body *body1, Body *body2, double acceleration[3]) {
    // Compute the distance between the two bodies across each dimension
    double dx = body2->position[0] - body1->position[0];
    double dy = body2->position[1] - body1->position[1];
    double dz = body2->position[2] - body1->position[2];

    // Compute the distance between the two bodies using the euclidean distance formula
    double distance = sqrt(dx*dx + dy*dy + dz*dz);

    // Compute the magnitude of the gravitational force between the two bodies
    // The formula is G * m2 / (r^2 + epsilon), where G is the gravitational constant,
    double magnitude = (G * body2->mass) / (distance * distance + 1e-10);

    // Compute the acceleration components using the formula a = (magnitude * d[x, y, z]) / (r + epsilon)
    acceleration[0] = magnitude * dx / (distance + 1e-10);
    acceleration[1] = magnitude * dy / (distance + 1e-10);
    acceleration[2] = magnitude * dz / (distance + 1e-10);
}

// Perform a single step of the Runge-Kutta integration
void runge_kutta_step(Body *body, double net_acceleration[3]) {
    // Compute the four Runge-Kutta steps for velocity and position
    double k1v[3], k1x[3];
    double k2v[3], k2x[3];
    double k3v[3], k3x[3];
    double k4v[3], k4x[3];

    // Compute the four Runge-Kutta steps for velocity and position
    for (int i = 0; i < 3; i++) {
        k1v[i] = net_acceleration[i] * DT;
        k1x[i] = body->velocity[i] * DT;
        k2v[i] = net_acceleration[i] * DT;
        k2x[i] = (body->velocity[i] + 0.5 * k1v[i]) * DT;
        k3v[i] = net_acceleration[i] * DT;
        k3x[i] = (body->velocity[i] + 0.5 * k2v[i]) * DT;
        k4v[i] = net_acceleration[i] * DT;
        k4x[i] = (body->velocity[i] + k3v[i]) * DT;
    }

    // Update the velocity and position using the Runge-Kutta steps
    for (int i = 0; i < 3; i++) {
        body->velocity[i] += (k1v[i] + 2*k2v[i] + 2*k3v[i] + k4v[i]) / 6;
        body->position[i] += (k1x[i] + 2*k2x[i] + 2*k3x[i] + k4x[i]) / 6;
    }
}
```

Listing 3.7: Computing accelerations and runge kutta

Part III

RESULTS & CONCLUSIONS

4

Analysis of Results

In this chapter, we analyze the results of the parallel implementation of the n-body problem. The focus will be on how elapsed time, speed-up, and efficiency behave as the number of processes increases. Finally, we will present a combined analysis of all metrics.

4.1 Amdahl's Law and the Plateau Effect

Amdahl's Law is a fundamental principle in parallel computing that helps us understand the theoretical speed-up achievable when parallelizing a task. It is defined by the equation:

$$S(p) = \frac{1}{(1-f) + \frac{f}{p}}$$

where $S(p)$ is the speed-up achieved with p parallel processes, f is the fraction of the task that can be parallelized, and $(1-f)$ represents the serial portion of the task.

The law highlights that no matter how many processes we add, the speed-up is ultimately limited by the serial portion of the computation. As p increases, the term $\frac{f}{p}$ diminishes, leaving the serial portion $(1-f)$ as the dominant factor. This results in a plateau in performance, where adding more processes yields little to no improvement.

For example, if 80% of a task is parallelizable ($f = 0.8$), the maximum achievable speed-up is:

$$S_{\text{max}} = \frac{1}{1-f} = \frac{1}{0.2} = 5$$

This plateau effect underscores the importance of reducing the serial fraction to maximize the benefits of parallelization. It serves as a reminder that even with perfect parallel algorithms, real-world limitations such as synchronization and communication overhead can cap the overall performance.

4.2 Elapsed Time

Elapsed time refers to the total time taken by the parallel algorithm to complete the computation. It is one of the most direct measures of performance. From the results, we can observe that as the number of processes increases, the elapsed time decreases. This is because the workload is divided among more processes, and each process handles only a part of the computation.

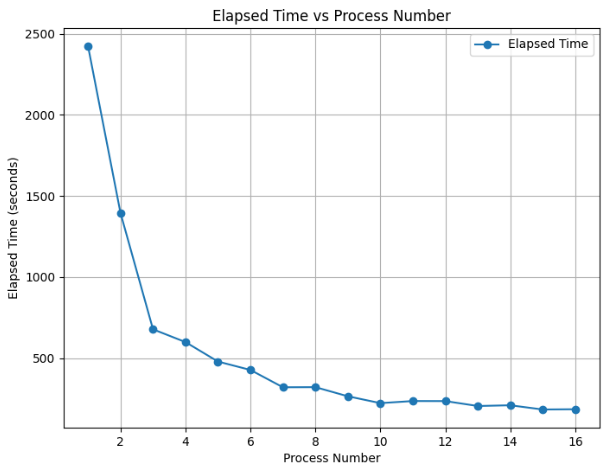

FIGURE 4.1: Elapsed time as a function of the number of processes.

The trend indicates that adding more processes improves the computation time, but this improvement slows down as we approach the limit of effective parallelization.

4.3 SPEED-UP

Speed-up measures how much faster the parallel implementation is compared to the sequential one. It is defined as the ratio of the elapsed time of the sequential algorithm to that of the parallel algorithm. Ideally, the speed-up should increase linearly with the number of processes, but due to factors like communication overhead and load imbalance, this is not always the case.

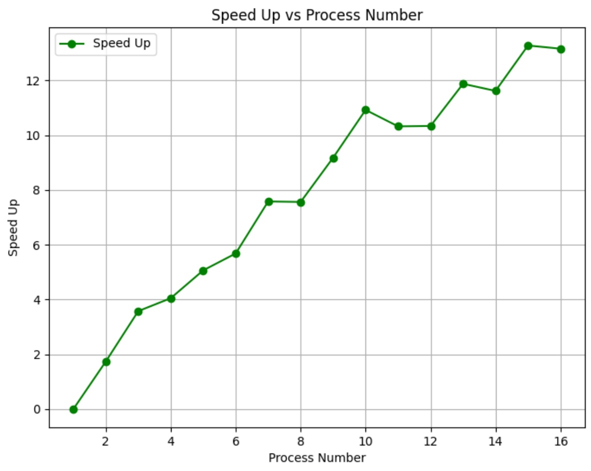

FIGURE 4.2: Speed-up as a function of the number of processes.

In the plot above, we can see that the speed-up increases with the number of processes, showing that the parallelization is effective. However, beyond a certain point, the curve starts to plateau due to diminishing returns.

4.4 EFFICIENCY

Efficiency is a metric that evaluates how effectively the processes are being utilized. It is calculated as the speed-up divided by the number of processes. High efficiency indicates that the workload is well-balanced and there is minimal overhead.

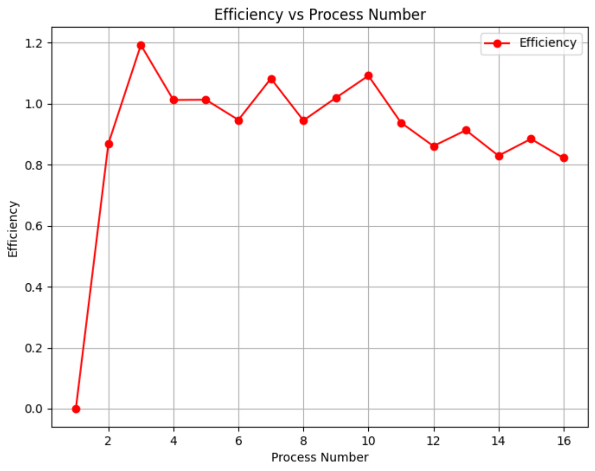

FIGURE 4.3: Efficiency as a function of the number of processes.

From the graph, it is clear that efficiency starts to drop as the number of processes increases. This is expected since adding more processes introduces overhead due to communication and synchronization. Despite this, the algorithm maintains relatively good efficiency for a significant range of processes.

4.5 CALCULATION OF THE METRICS

To calculate the metrics we first run the algorithm using only one processor to calculate and store the baseline time into a file. This baseline file will then be read and used in order to compute the metrics

```c
    if (rank == 0) {
        double elapsed = end - start;
        printf("---------------------N of processors used: %d
---------------------\n", size);
        printf("Elapsed time: %.6f seconds\n", elapsed);

        if (size == 1) {
            FILE *file = fopen("baseline_time.txt", "w");
            if (file) {
                fprintf(file, "%.6f\n", elapsed);
                fclose(file);
                printf("Baseline time saved.\n");
            }
        } else {
            FILE *file = fopen("baseline_time.txt", "r");
            if (file) {
                double baseline;
                fscanf(file, "%lf", &baseline);
                fclose(file);

                double speedup = baseline / elapsed;
                double efficiency = speedup / size;
                printf("Speedup: %.6f\n", speedup);
                printf("Efficiency: %.6f\n", efficiency);
            }
        }
        printf("
    -------------------------------------------------------------------\
    n");
    }
```

Listing 4.1: Saving the baseline and computing the metrics

5 Conclusion

This project has highlighted the transformative potential of parallel computing in addressing computational challenges. By analyzing elapsed time, speed-up, and efficiency, we demonstrated the benefits of parallelization while also uncovering its inherent limitations. The significant reduction in elapsed time with increasing processes showcased the capability of parallel algorithms to accelerate computations. However, diminishing returns beyond a certain point underscored the challenges posed by synchronization and communication overheads.

The analysis of speed-up provided insights into the scalability of parallel systems. While near-linear improvements were observed at lower process counts, the plateauing effect highlighted the constraints imposed by Amdahl's Law. These findings emphasized the importance of minimizing sequential bottlenecks to achieve greater performance gains. Similarly, efficiency declined as processes increased, reflecting the balance between scalability and resource utilization. This observation reinforced the need for algorithmic strategies that mitigate overhead and balance workloads effectively.

Beyond the numerical results, the project emphasized the importance of managing trade-offs in parallel computing. While parallelization offers significant performance enhancements, it also introduces complexities that must be carefully addressed. The observed limits of speed-up and efficiency highlight the need for continued innovation in algorithm design and resource management.

In conclusion, this project provided a deeper understanding of the principles and challenges of parallel computing. It demonstrated the power of parallelization to meet growing computational demands while underscoring the importance of balancing efficiency and scalability. The insights gained here serve as a foundation for future exploration and innovation in high-performance computing.