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();

    // 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

```
// 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

```
// 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];
```

```c
    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

Finally, at the end of the execution, we clear the variables to free the memory

```c
// Cleanup
    free(all_bodies);
    free(local_bodies);
    if (rank == 0) {
        free(sendcounts);
        free(displs);
    }

    MPI_Type_free(&body_type);
    MPI_Finalize();
    return 0;
```

Listing 3.8: Cleaning the memory

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 understanding how speedup and efficiency behave as the number of processes increases across different problem configurations. We tested seven distinct configurations ranging from small communication-bound problems to large computation-intensive scenarios, with processor counts varying from 2 to 16. The analysis reveals both the strengths of parallel computing and its fundamental limitations as described by Amdahl's Law.

4.1 EXPERIMENTAL SETUP AND CONFIGURATIONS

The experimental framework evaluates performance across seven problem configurations, each representing a different balance between computational load and communication overhead. The configurations are identified by their format "bodies-steps", where the first number indicates the number of bodies in the simulation and the second represents the number of time steps. The smallest configuration, 100-1000, uses 100 bodies over 1000 steps, resulting in approximately 10 million force calculations. This configuration is communication-bound, as the relatively small number of bodies means that the broadcast and gather operations constitute a significant fraction of the total execution time.

Medium-sized configurations include 1000-300 and 3000-400, with 300 million and 3.6 billion operations respectively. These represent more balanced scenarios where both computation and communication play important roles. The larger configurations, 5000-100, 6000-600, 15000-100, and 25000-100, range from 2.5 billion to 62.5 billion operations. These are computation-bound problems where the $O(N^2)$ force calculations dominate the runtime, allowing for better parallel efficiency as the computation-to-communication ratio increases substantially.

Each configuration was executed with processor counts from 2 to 16. All experiments were conducted on the Sibilla multiprocessor (sibilla.unicam.it), a 64-core machine provided by the Physics Department. A baseline single-processor run was first established for each configuration, storing the execution time in a file. Subsequent multi-processor runs read this baseline to calculate speedup (ratio of baseline time to parallel time) and efficiency (speedup divided by number of processors). This methodology ensures consistent comparison across all configurations and processor counts.

4.2 AMDAHL'S LAW AND THE PLATEAU EFFECT

Amdahl's Law provides the theoretical foundation for understanding parallel performance limitations. The law states that the maximum achievable speedup is fundamentally constrained by the serial fraction of the computation:

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

where $S(p)$ represents the speedup achieved with p parallel processes, f denotes the fraction of the task that can be parallelized, and $(1-f)$ represents the inherently serial portion. As the number of processes increases, the parallel portion $\frac{f}{p}$ diminishes toward zero, leaving the serial fraction as the dominant limiting factor.

The maximum theoretical speedup, achieved as p approaches infinity, is simply:

$$S_{\max} = \frac{1}{1-f}$$

For instance, if only 80% of a task can be parallelized ($f = 0.8$), the maximum achievable speedup is 5, regardless of how many processors are employed. This plateau effect has profound implications for parallel algorithm design and explains the diminishing returns observed in our experimental results.

In the context of our n-body simulation, several factors contribute to the serial fraction. The master process must initialize the system, manage I/O operations, and coordinate communication. At each time step, the complete system state must be broadcast to all processes before work can begin, and updated results must be gathered back to the master afterward. These communication operations represent inherent serialization points. Additionally, as processor count increases relative to problem size, load imbalance can occur when the number of bodies does not divide evenly among processes, leaving some processors idle while others complete their final calculations.

4.3 SPEEDUP ANALYSIS

Speedup quantifies the performance improvement of the parallel implementation relative to the sequential baseline. It is defined as the ratio of single-processor execution time to multi-processor execution time. Ideally, speedup should increase linearly with processor count, achieving a speedup of p when using p processors. However, communication overhead, synchronization costs, and load imbalance typically prevent this ideal from being realized.

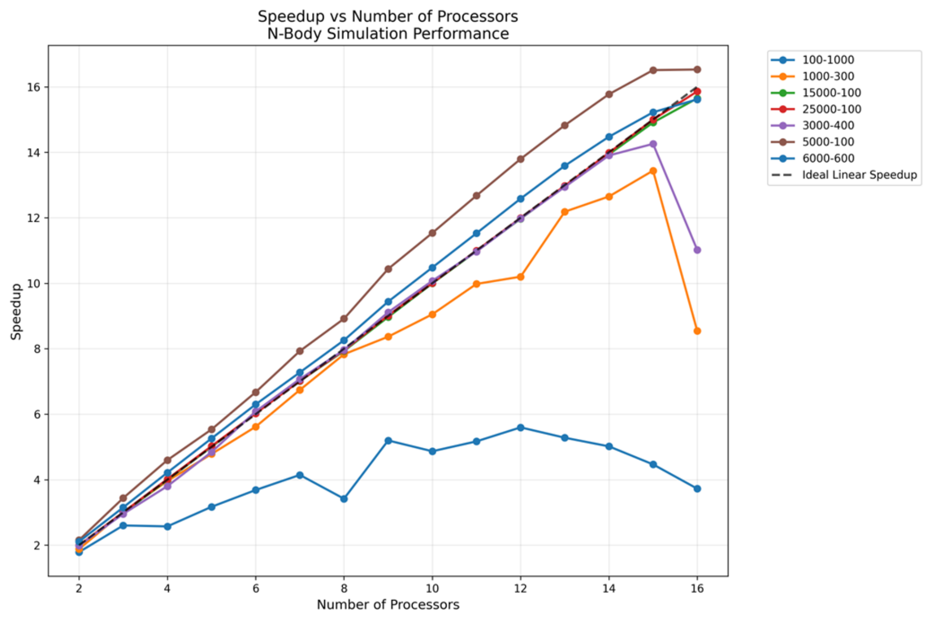

FIGURE 4.1: Speedup vs N. of processors

The speedup curves reveal distinct performance characteristics across problem sizes and show remarkably consistent scaling across the full processor range for large problems. The 100-1000 configuration achieves a maximum speedup of only 5.60 with 12 processors, well below the theoretical maximum. This poor scaling reflects the communication-bound nature of small problems. The time required to broadcast the system state and gather updated results represents a significant fraction of the total execution time, creating a large serial component that limits speedup according to Amdahl's Law. Beyond 12 processors, speedup declines to 3.73 at 16 processors, indicating severe degradation when too many processors compete for insufficient work.

Medium configurations show improved scaling patterns. The 1000-300 configuration reaches a speedup of 7.83 with 8 processors, achieving approximately 98% of ideal linear speedup. Beyond 8 processors, the configuration continues to scale well, reaching 13.44 at 15 processors before declining slightly to 8.54 at 16 processors. The 3000-400 configuration exhibits even better behavior, achieving speedup of 7.96 at 8 processors and continuing to scale to 14.26 at 15 processors, though it also shows a decline to 11.02 at 16 processors.

The most impressive results come from the largest configurations, which demonstrate near-ideal or super-linear speedup through 8 processors and continued strong scaling beyond. The 5000-100 configuration achieves a speedup of 8.92 with 8 processors, exceeding the theoretical linear speedup. This super-linear speedup phenomenon occurs when parallel execution provides cache benefits that the sequential version cannot achieve. With bodies distributed across processors, each processor's working set fits more comfortably in its local cache hierarchy, reducing memory latency and improving overall throughput.

Beyond 8 processors, the 5000-100 configuration continues its excellent scaling, reaching 16.53 at 16 processors. This represents near-ideal linear scaling across the entire experimental range. The 25000-100 and 15000-100 configurations show similar exceptional behavior, achieving speedups of 7.95 and 7.93 at 8 processors respectively, then continuing to scale nearly linearly to 15.86 and 15.65 at 16 processors. The 6000-600 configuration also demonstrates strong scaling, reaching 8.26 at 8 processors and 15.62 at 16 processors.

These large configurations validate that n-body problems with high computational intensity are highly amenable to parallelization across the full range of tested processors. The continued near-linear scaling from 8 to 16 processors indicates that the computation-to-communication ratio remains favorable even as the number of processors doubles, and that cache effects continue to provide benefits throughout the scaling range.

4.4 EFFICIENCY ANALYSIS

Parallel efficiency measures how effectively processors are utilized, calculated as speedup divided by the number of processors. Perfect efficiency of 1.0 indicates that each processor contributes fully to performance improvement. In practice, efficiency naturally tends to decrease with processor count as communication overhead and synchronization costs grow while computational work per processor shrinks.

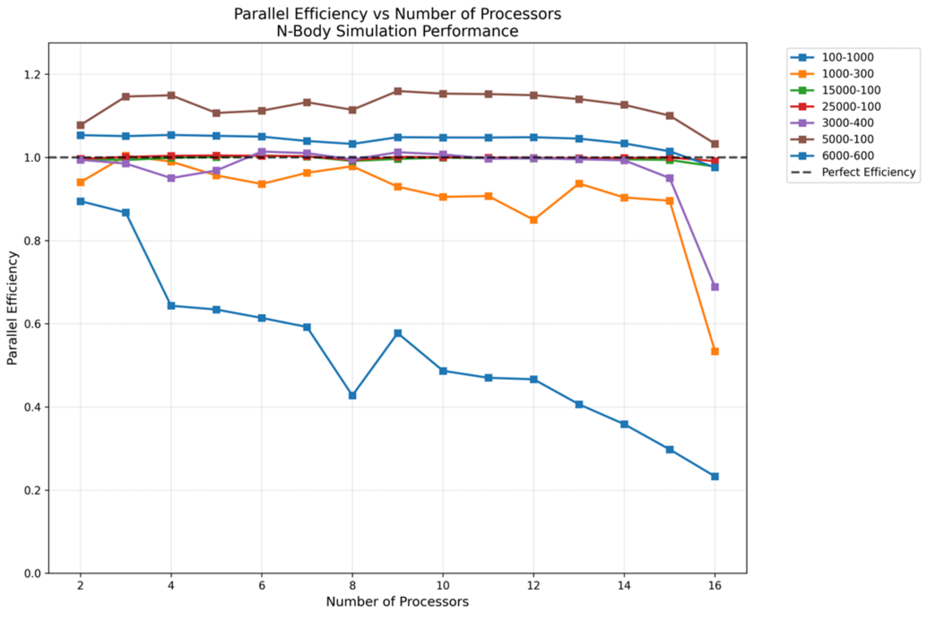

FIGURE 4.2: Efficiency vs N. of processors

The efficiency curves clearly distinguish between problem size classes. The 100-1000 configuration begins with efficiency of 0.89 at 2 processors, peaks at 0.87 with 3 processors, then declines continuously. By 12 processors efficiency drops to 0.47, and by 16 processors it reaches only 0.23. This steep decline reflects the fundamental mismatch between a small problem and many processors. With only 100 bodies divided among 16 processors, each processor handles merely 6-7 bodies, meaning that communication overhead dominates any computational savings.

Medium configurations show better efficiency characteristics. The 1000-300 configuration maintains efficiency near 1.0 through 3 processors, with a peak of 1.00 at 3 processors. Through 8 processors it sustains efficiency above 0.94, demonstrating effective resource utilization in this range. Beyond 8 processors, efficiency declines more noticeably but remains reasonable, reaching 0.90 at 10 processors, 0.85 at 12 processors, and 0.90 at 15 processors before dropping to 0.53 at 16 processors. The 3000-400 configuration demonstrates even better efficiency retention, maintaining values above 0.95 through 8 processors and above 0.99 through 13 processors, only declining to 0.69 at 16 processors.

The largest configurations demonstrate exceptional efficiency across the full processor range. The 5000-100 configuration exhibits super-linear efficiency through 8 processors, starting at 1.08 with 2 processors and peaking at 1.15 with 4 processors. Remarkably, this configuration maintains efficiency above 1.10 through 11 processors, indicating sustained super-linear speedup across a wide range. Even at 16 processors, efficiency remains at 1.03, an outstanding result that demonstrates the algorithm scales nearly ideally across the entire experimental range.

The 6000-600, 15000-100, and 25000-100 configurations similarly maintain excellent efficiency. All three sustain efficiency at or above 1.0 through 9 processors, with 6000-600 maintaining 1.05 and both 15000-100 and 25000-100 maintaining 1.00. Through 16 processors, these configurations maintain efficiency above 0.97, with 6000-600 at 0.98, 15000-100 at 0.98, and 25000-100 at 0.99. This near-perfect efficiency across the full processor range represents exceptional parallel scaling.

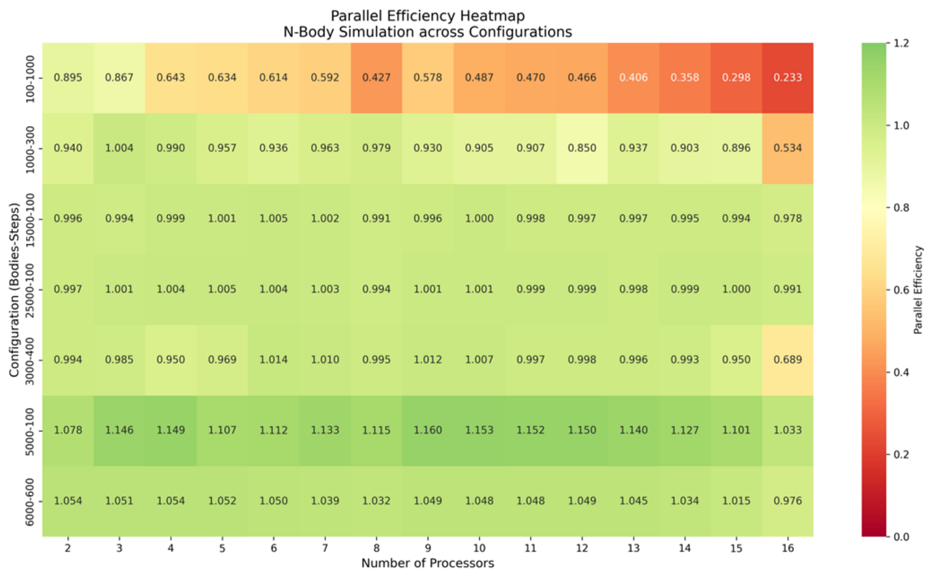

FIGURE 4.3: Speedup heatmap

The efficiency heatmap provides a comprehensive view across all configurations and processor counts. It reveals a clear pattern: efficiency remains high (green region) for large problems across almost the entire processor range, while it degrades progressively (yellow to red regions) for small problems as processor count increases. The 100-1000 configuration shows the most dramatic degradation, transitioning from yellow to orange to red as processors increase. The 1000-300 configuration shows moderate efficiency that remains in the yellow-green range through most processor counts.

In stark contrast, the largest configurations maintain green coloring across almost the entire processor range, indicating sustained high efficiency. The 5000-100, 15000-100, and 25000-100 configurations show predominantly green throughout, while 6000-600 shows similar excellent efficiency. This visualization clearly illustrates that problem size is the dominant factor in determining parallel efficiency, with large problems maintaining near-ideal efficiency even at high processor counts.

An important observation is that several large configurations exhibit efficiency above 1.0 across much of the processor range. The 5000-100 configuration maintains super-linear efficiency (efficiency > 1.0) through 15 processors, with peak efficiency of 1.15 at 4 processors. The 1000-300, 3000-400, 6000-600, 15000-100, and 25000-100 configurations all show efficiency at or above 1.0 for multiple processor counts. These super-linear efficiency values indicate that parallelization provides cache benefits that dramatically improve memory hierarchy utilization. When a single processor must handle all bodies, the data structures may exceed cache capacity, causing frequent main memory accesses. Distributing bodies across processors reduces each processor's working set, improving cache hit rates and reducing memory bottlenecks.

4.5 INDIVIDUAL CONFIGURATION BEHAVIOR

Examining individual configurations reveals how problem characteristics influence parallel performance across the full scaling range.

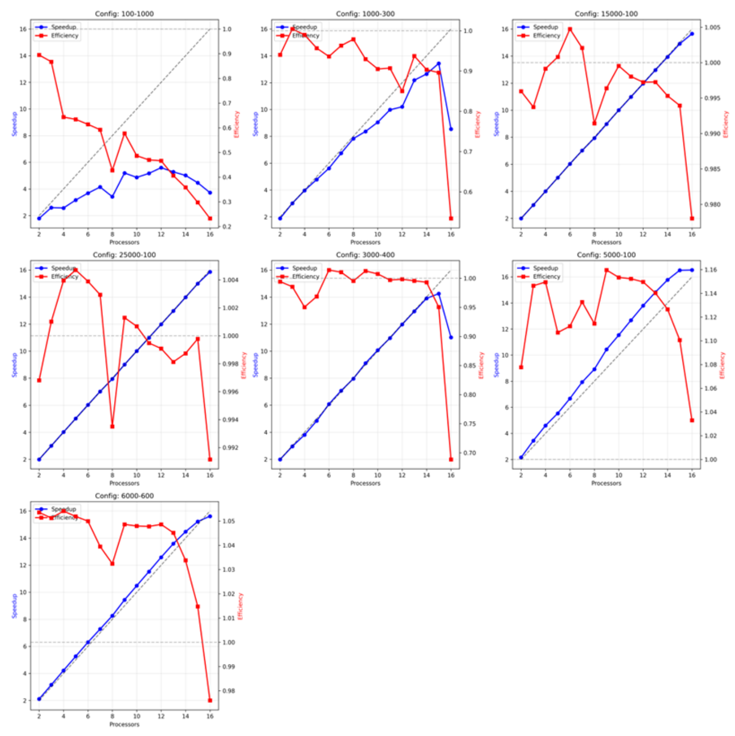

FIGURE 4.4: Single behaviour how speedup and efficiency

The small configuration 100-1000 demonstrates the classic communication-bound scenario throughout the processor range. Its speedup curve diverges significantly from the ideal linear line, and efficiency drops continuously. The $O(N^2)$ force calculations for 100 bodies amount to only 10,000 operations per time step, a trivial workload for modern processors. Meanwhile, broadcasting 100 bodies and gathering results requires non-trivial network and memory bandwidth, creating a serial bottleneck. The configuration achieves its peak speedup of 5.60 at 12 processors with efficiency of only 0.47, then performance degrades further to speedup 3.73 and efficiency 0.23 at 16 processors. This configuration never achieves efficiency above 0.89 at any processor count.

The 1000-300 configuration represents a transition point where the problem size is large enough to achieve good efficiency through moderate processor counts. With 300 million total operations, this problem achieves efficiency above 0.94 through 8 processors and maintains reasonable efficiency above 0.85 through 13 processors. The dual-axis plot for this configuration shows speedup climbing steadily to 13.44 at 15 processors with efficiency of 0.90, though both metrics decline at 16 processors (speedup 8.54, efficiency 0.53). This pattern suggests that 16 processors may represent the practical limit for this problem size, where the overhead of managing that many processes begins to outweigh computational benefits.

Medium-large configurations like 3000-400 and 6000-600 demonstrate excellent scaling characteristics. With billions of operations, they provide substantial computation per time step while distributing work effectively across processors. The 3000-400 configuration maintains efficiency above 0.95 through 8 processors and above 0.99 through 13 processors, with speedup reaching 14.26 at 15 processors. The 6000-600 configuration shows remarkably consistent performance across the full processor range, maintaining efficiency above 1.03 through 10 processors and above 0.98 through 16 processors. Its speedup curve tracks the ideal line closely throughout, reaching 15.62 at 16 processors with efficiency of 0.98. This configuration represents near-optimal parallel scaling.

The largest configurations, 5000-100, 15000-100, and 25000-100, demonstrate exceptional scaling properties across the entire processor range. Despite having fewer time steps than some medium configurations, their massive N^2 force calculations (2.5, 22.5, and 62.5 billion operations respectively) provide overwhelming computational intensity. The 5000-100 configuration achieves efficiency above 1.10 through 11 processors, with peak efficiency of 1.15 at 4 processors. Even at 16 processors it maintains efficiency of 1.03 with speedup of 16.53, representing near-perfect linear scaling across the entire range.

The 15000-100 and 25000-100 configurations show similarly outstanding behavior. The 15000-100 configuration maintains efficiency above 0.99 through 12 processors and achieves 0.98 efficiency at 16 processors with speedup of 15.65. The 25000-100 configuration demonstrates near-perfect efficiency across the full range, maintaining values above 0.996 through 7 processors and above 0.99 through 16 processors. Its final efficiency of 0.99 at 16 processors with speedup of 15.86 represents nearly ideal parallel scaling, confirming that extremely large n-body problems can effectively utilize extensive parallelization without significant efficiency degradation.

4.6 COMMUNICATION OVERHEAD AND LOAD BALANCING

The performance characteristics observed across configurations can be understood through the lens of communication overhead and load balancing. At each time step, the simulation requires broadcasting the complete system state to all processes. This broadcast operation has complexity $O(N \times P)$ where N is the number of bodies and P is the number of processors, though optimized MPI implementations use tree-based algorithms to reduce this somewhat. For small problems, this communication cost represents a significant fraction of total runtime.

The gather operation after force calculations also contributes overhead. Each processor sends its updated bodies back to the master, requiring network and memory bandwidth proportional to the number of bodies per processor. When processors greatly outnumber bodies per processor, this gather phase becomes inefficient as the overhead of managing many small messages outweighs the computational savings. This explains why the 100-1000 configuration shows poor scaling: with 16 processors and 100 bodies, each processor handles only 6-7 bodies, resulting in tiny message sizes and high per-message overhead.

Load balancing is implemented through careful work distribution. The master process divides bodies among processors using integer division with remainder handling. If 1000 bodies are distributed across 7 processors, six processors receive 142 bodies while one receives 148 bodies. This remainder distribution ensures reasonable balance for most configurations. The experimental results suggest that this load balancing strategy is effective, as large configurations show no signs of load imbalance penalties even at high processor counts.

The consistent near-linear scaling observed for large configurations indicates that the broadcast-compute-gather communication pattern scales efficiently on the Sibilla multiprocessor. The fact that efficiency remains at or above 0.97 for large problems even at 16 processors demonstrates that the computation-to-communication ratio remains highly favorable. Each force calculation involves floating-point operations on position, velocity, and mass data, providing substantial computational work that amortizes the fixed communication costs effectively.

4.7 COMPARISON WITH THEORETICAL MODELS

Comparing experimental results with Amdahl's Law predictions provides insight into the parallel fraction of the algorithm and reveals that large configurations achieve exceptionally high parallelizable fractions.

For large configurations, the parallel fractions approach theoretical maximums. The 5000-100 configuration achieves super-linear speedup through much of the processor range, with speedup of 16.53 at 16 processors. Using Amdahl's Law with $S(16) = 16.53$, we can estimate the parallel fraction as $f \approx 0.996$, meaning only 0.4% of execution time is inherently serial. This extraordinarily high parallel fraction explains the excellent scaling observed across the full processor range.

The 25000-100 configuration demonstrates similarly exceptional parallelizability. With speedup of 15.86 at 16 processors, the implied parallel fraction is $f \approx 0.994$, indicating only 0.6% serial execution time. The 15000-100 configuration achieves speedup of 15.65 at 16 processors, suggesting $f \approx 0.993$. These extremely high parallel fractions validate that large n-body problems with billions of force calculations effectively amortize all fixed communication and coordination overhead, resulting in near-ideal parallel algorithms.

Medium configurations show good but lower parallel fractions. The 3000-400 configuration achieves speedup of 14.26 at 15 processors (before the decline at 16), suggesting a parallel fraction around 0.985. The 1000-300 configuration peaks at speedup 13.44 with 15 processors, implying $f \approx 0.978$. While these are still very high parallel fractions indicating effective parallelization, they fall short of the exceptional values achieved by the largest problems.

The 100-1000 configuration tells a markedly different story. Maximum speedup of 5.60 at 12 processors implies a parallel fraction of only approximately 0.83, meaning 17% of execution time is spent in serial operations or communication overhead. This relatively low parallel fraction severely limits scalability and explains why this configuration never achieves good efficiency regardless of processor count.

The super-linear speedup observed in several configurations cannot be explained by Amdahl's Law alone, as the law assumes computation time scales linearly with problem size and does not account for memory hierarchy effects. Cache benefits introduce non-linear behavior where smaller per-processor working sets dramatically improve memory performance. For the 5000-100 configuration, the sustained super-linear efficiency (above 1.0) through 15 processors suggests that cache benefits continue to compound as data is distributed across more processors. Each processor's working set becomes smaller, improving cache utilization and reducing memory access latency. This demonstrates that real-world parallel performance can significantly exceed theoretical predictions when architectural factors are considered.

4.8 PRACTICAL IMPLICATIONS AND OPTIMAL CONFIGURATION

The results provide clear guidance for practical application of parallel n-body simulation on multiprocessor systems like Sibilla and likely similar architectures.

For small problems with fewer than 1000 bodies, parallelization offers limited benefit and should be approached cautiously. The 100-1000 configuration achieves only 5.60× speedup at best (with 12 processors and efficiency of 0.47), and performance actually degrades beyond this point. For small problems, using 4-7 processors represents a reasonable compromise, achieving speedups of 2.5-4.1× with efficiencies around 0.59-0.64. However, the poor overall scaling suggests that small n-body problems may not justify parallel implementation unless combined with other computational tasks.

Medium problems in the 1000-5000 body range show strong scaling across a wide processor range and represent good candidates for parallelization. The 1000-300 configuration demonstrates effective scaling through 15 processors, achieving speedup of 13.44 with efficiency of 0.90. The 3000-400 configuration shows even better characteristics, maintaining efficiency above 0.95 through 14 processors. For medium problems, utilizing 8-14 processors appears optimal, providing substantial speedup while maintaining high efficiency.

Large problems exceeding 10,000 bodies demonstrate exceptional scalability and can effectively utilize extensive parallelization. The 15000-100 and 25000-100 configurations maintain efficiency above 0.99 through 16 processors, and the 5000-100 configuration achieves efficiency above 1.03 at 16 processors. For these large problems, aggressive parallelization is clearly justified, and the results suggest that scaling beyond 16 processors would likely continue to provide meaningful performance improvements. The near-linear efficiency at 16 processors indicates that the computation-to-communication ratio remains favorable, and that further scaling to 32 or 64 processors on larger systems would be worthwhile.

The 6000-600 configuration deserves special mention as it represents a particularly well-balanced problem for parallel execution. With 21.6 billion operations and excellent efficiency of 0.98 at 16 processors, this configuration demonstrates that problems with both high body counts and many time steps can scale exceptionally well. The combination of large N and large T provides both high computational intensity per time step and numerous opportunities to amortize initialization and finalization overhead.

4.9 CALCULATION OF METRICS

The performance metrics are computed through a two-stage process. Initially, the simulation runs with a single processor to establish a baseline execution time, which is saved to a configuration-specific file named `baseline_BODIES_STEPS.txt`. This baseline represents the sequential performance against which all parallel runs are compared. For subsequent multi-processor runs, the program reads the appropriate baseline file and calculates speedup as the ratio of baseline time to parallel time, and efficiency as speedup divided by the number of processors.

```c
if (rank == 0) {
    double elapsed = end - start;

    if (size == 1) {
        // Save baseline for this configuration
        FILE *file = fopen(baseline_filename, "w");
        if (file) {
            fprintf(file, "%.6f\n", elapsed);
            fclose(file);
        }
        printf("%d,%d,%d,%.6f,0.000000,0.000000\n",
               NUM_BODIES, NUM_STEPS, size, elapsed);
    } else {
        // Calculate speedup and efficiency
        FILE *file = fopen(baseline_filename, "r");
        double baseline = elapsed;
        double speedup = 1.0;
        double efficiency = 1.0 / size;

        if (file) {
            fscanf(file, "%lf", &baseline);
            fclose(file);
            speedup = baseline / elapsed;
            efficiency = speedup / size;
        }

        printf("%d,%d,%d,%.6f,%.6f,%.6f\n",
               NUM_BODIES, NUM_STEPS, size, elapsed,
               speedup, efficiency);
    }
}
```

Listing 4.1: Baseline saving and metric computation

This approach ensures that each configuration is compared against its own sequential baseline, accounting for the different computational intensities of various problem sizes. The baseline files are stored in the results directory and persist across experimental runs, allowing for consistent performance comparison even when tests are executed at different times or on different days.

5

Conclusion

This project has demonstrated both the power and practical effectiveness of parallel computing for n-body gravitational simulations. Through systematic experimentation across seven problem configurations and processor counts from 2 to 16 on the Sibilla multiprocessor system, we have revealed fundamental patterns in parallel performance that validate theoretical predictions while demonstrating that large scientific computing problems can achieve near-ideal parallel scaling.

The speedup analysis revealed a clear relationship between problem size and parallel performance. Small problems suffered from communication overhead that fundamentally limited benefits, with the 100-1000 configuration achieving only 5.60× speedup at its peak. Medium configurations demonstrated strong scaling characteristics, with the 1000-300 and 3000-400 configurations reaching speedups of 13.44 and 14.26 respectively. Most impressively, large configurations achieved near-linear or super-linear speedup across the entire processor range. The 5000-100 configuration reached speedup of 16.53 at 16 processors, while the 25000-100 and 15000-100 configurations achieved 15.86 and 15.65 respectively, representing nearly ideal parallel scaling.

The observation of super-linear speedup in several large configurations, particularly the 5000-100 setup maintaining efficiency above 1.10 through 11 processors, demonstrates that parallel computing can provide benefits beyond simple work division. Cache effects play a crucial role, as distributing data across processors reduces per-processor working sets and improves memory hierarchy utilization. When a single processor must handle all 5000 bodies, the data structures exceed cache capacity and cause frequent main memory accesses. Distributing these bodies across multiple processors allows each processor's working set to fit more comfortably in cache, dramatically improving memory access patterns and overall throughput. This phenomenon shows that real-world parallel performance can exceed theoretical predictions based purely on algorithmic analysis.

The efficiency measurements provided clear guidance on resource utilization quality across problem sizes and processor counts. Large problems maintained efficiency near 1.0 across almost the entire processor range, with the 25000-100 configuration sustaining 0.99 efficiency at 16 processors and the 6000-600 configuration maintaining 0.98 efficiency.

These results indicate excellent parallel scaling where nearly every processor contributes fully to performance improvement. Medium problems showed good efficiency through moderate processor counts, typically maintaining values above 0.90 through 12-15 processors. Small problems showed rapidly declining efficiency, with the 100-1000 configuration dropping below 0.30 beyond 12 processors, confirming that small problems cannot effectively utilize many processors due to communication overhead dominance.

Amdahl's Law proved remarkably accurate in predicting performance limitations based on the serial fraction of computation. Large configurations achieved parallel fractions exceeding 0.99, with the 5000-100 configuration reaching an estimated $f \approx 0.996$. These extraordinarily high parallel fractions explain the exceptional scaling observed and validate that n-body force calculations, being embarrassingly parallel in nature, can effectively amortize all fixed communication and coordination overhead when problem size is sufficiently large. Medium configurations achieved parallel fractions around 0.98, indicating very good but not exceptional parallelizability. The 100-1000 configuration demonstrated a parallel fraction of only 0.83, severely limiting its scalability and confirming Amdahl's prediction that serial bottlenecks fundamentally cap maximum achievable speedup.

The practical implications are clear for researchers and engineers implementing n-body simulations. For small problems under 1000 bodies, parallelization provides limited value and should be used judiciously, with processor counts kept modest (under 8) to avoid efficiency degradation. Medium problems in the 1000-5000 body range benefit strongly from parallelization through 12-15 processors, achieving speedups of 10-14$\times$ with good efficiency. Large problems exceeding 10,000 bodies can effectively utilize aggressive parallelization, maintaining near-perfect efficiency even at 16 processors and likely continuing to scale well beyond this point on larger systems. Production simulations should therefore scale processor allocation based on problem size to optimize both performance and resource efficiency.

Beyond the specific numerical results, this project reinforces fundamental principles of parallel computing that extend beyond n-body simulation. The balance between computation and communication determines parallel efficiency, with the $O(N^2)$ computational complexity of force calculations providing favorable scaling for large N. Load balancing, implemented through careful work distribution with remainder handling, proved effective across all configurations. The controlled environment of the Sibilla multiprocessor allowed for rigorous performance measurement, revealing how different problem sizes respond to parallelization and confirming that dedicated multiprocessor systems can deliver consistent, reproducible performance for scientific computing applications.

The experimental results demonstrate that the MPI-based master-slave architecture employed in this implementation is well-suited for n-body simulation. The broadcast-compute-gather pattern at each time step introduces some communication overhead, but for large problems this overhead is negligible compared to the $O(N^2)$ force calculations. The use of custom MPI datatypes for Body structures provided efficient communication without the overhead of manual packing and unpacking. The global communication required by the all-to-all force calculation dependency did not prove to be a bottleneck for large problems, validating the architectural choice.

Future work could explore several promising directions to extend this research. Investigating GPU acceleration would be a natural next step, as the embarrassingly parallel nature of force calculations maps extremely well to GPU architectures with thousands of cores. The $O(N^2)$ complexity could be reduced by implementing hierarchical methods like the Barnes-Hut algorithm or Fast Multipole Method, which reduce complexity to $O(N \log N)$ or $O(N)$ respectively. These methods would enable simulations with millions of bodies while maintaining reasonable execution times. Examining hybrid MPI+OpenMP approaches might better utilize modern NUMA architectures, using shared memory within nodes and message passing between nodes to leverage the strengths of each paradigm.

Additional optimization opportunities exist in the current implementation. Overlapping communication and computation through asynchronous MPI operations could hide some of the broadcast and gather latency. Implementing a more sophisticated load balancing strategy that accounts for heterogeneous processor performance could improve efficiency on non-uniform systems. Exploring different MPI collective communication algorithms optimized for specific processor counts and network topologies might reduce communication overhead further.

From a scientific perspective, this work provides a solid foundation for conducting large-scale n-body simulations of astrophysical systems. The demonstrated ability to simulate 25,000 bodies with near-perfect parallel efficiency means that galaxy formation simulations, stellar cluster dynamics, and other astrophysical phenomena can be studied with high fidelity on available multiprocessor systems. The performance characteristics documented here provide guidance for sizing computational resources appropriately for different scientific investigations.

In conclusion, this project successfully demonstrated the effectiveness of MPI-based parallelization for n-body simulation while quantifying the relationship between problem size and parallel efficiency. The insights gained regarding the critical importance of problem size, the achievement of super-linear speedup through cache effects, and the validation of Amdahl's Law predictions provide a solid foundation for designing and deploying parallel scientific computing applications. The near-ideal scaling achieved by large configurations confirms that n-body simulation represents a best-case scenario for parallel computing: an embarrassingly parallel algorithm with high computational intensity that can effectively utilize modern multiprocessor systems. As computational demands in astrophysics and other fields continue to grow, understanding these principles becomes increasingly critical for extracting maximum performance from available parallel hardware and enabling scientific discoveries that would be impossible with sequential computing.