CAB401 Parallelisation Project Report

Bailey Rossiter
Student ID: 11326158

Queensland University of Technology
Faculty of Engineering

October 26, 2025

Contents

1 Introduction

This project investigates the parallelisation of a sequential N-body simulation written in C. The N-body problem models the motion of multiple bodies under their mutual gravitational attraction, where each body exerts a force on every other body. The sequential implementation calculates these pairwise interactions using a direct $O(N^2)$ algorithm, which becomes increasingly expensive as the number of simulated bodies grows.

1.1 Overview

At a functional level, the simulation models gravitational motion over a series of discrete timesteps. Each timestep performs the following operations:

1. Compute gravitational forces between all pairs of bodies ($O(N^2)$).
2. Update each body's velocity and position using Euler integration.
3. Optionally record the current positions for visualisation after the run.

Runtime parameters such as the number of bodies, number of steps, timestep size, and thread count are provided through command-line arguments. If the thread count is greater than one, the program runs the parallel solver; otherwise, it defaults to the sequential version. When the `--noview` flag is not used, recorded simulation data is visualised using the SDL-based viewer once the simulation completes.

1.2 Software Architecture

The program is organised into modular C source files with a single entry point in `main.c`. The main function parses command-line arguments (via `cli_helpers.c`), initialises body data, and calls either the sequential solver (`nbody_seq.c`) or the parallel solver (`nbody_parallel.c`). Both versions share common definitions for the `Body` structure and constants in `nbody.h`.

The sequential solver performs all force calculations and integrations on a single thread using a direct $O(N^2)$ nested loop over body pairs. Each step updates velocities and positions with the explicit Euler method and stores snapshots at a user-defined interval for later playback. The viewer (`viewer.c`) runs independently, rendering these pre-recorded frames in 3D after the simulation.

Figure 1 shows the overall control flow of the sequential simulation.

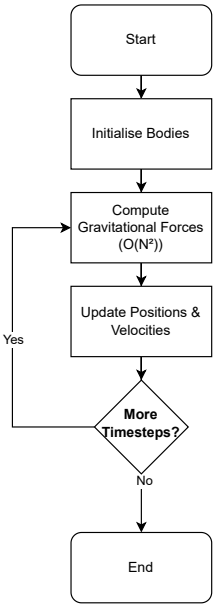

Figure 1: High-level flow of the sequential N-body simulation. The program initialises body states, computes all pairwise forces, applies Euler integration to update positions and velocities, and repeats this process for the configured number of timesteps.

1.3 Performance Context

The sequential solver produces correct results but scales poorly for large N. Each timestep performs $O(N^2)$ work on a single CPU core, causing runtime to increase quadratically while other cores remain unused. The goal of this project is to identify parallel regions in the code, restructure the force calculation phase, and implement a PThread-based approach that improves performance while maintaining numerical accuracy.

2 Analysis of the Original Application

The sequential simulation performs two main computational phases per timestep:

1. Force computation: Each body accumulates the net gravitational force from all other bodies ($O(N^2)$).

2. Integration: Each body's velocity and position are updated using Euler integration ($O(N)$).

2.1 Identification of Parallel Regions

Profiling and code inspection show that most of the runtime occurs within the nested loops of the force computation function, `compute_forces()`. Each iteration of the outer loop operates independently, as the force on body i depends only on the positions and masses of other bodies, not on results from other iterations. This makes the outer loop an ideal target for parallelisation.

In contrast, the integration phase (`update_bodies()`) performs simple, per-body updates that are linear in N and contribute very little to total runtime. Parallelising this phase would add synchronisation overhead without meaningful performance gain.

2.2 Data and Control Dependencies

The dominant computation in the sequential version is the nested loop in `compute_forces()`, which iterates over all body pairs (i, j). The outer loop index i represents the target body being updated, while the inner loop index j traverses all other bodies that exert forces on i.

Data Dependencies. Within a single timestep, each iteration of the outer loop in the sequential solver performs the following operations:

- Reads the positions and masses of all bodies: `b[j].x, b[j].y, b[j].z, b[j].mass`.
- Reads the position and mass of the current body: `b[i].x, b[i].y, b[i].z, b[i].mass`.
- Writes only to that body's accumulated force components: `b[i].fx, b[i].fy, b[i].fz`.

Because each outer-loop iteration updates a distinct body index `i` and only reads shared data, there are no loop-carried dependencies between iterations:

- Read-after-Write (RAW): Absent — positions and masses remain constant during each timestep.
- Write-after-Read (WAR): Absent — each iteration writes only to its own force accumulator.
- Write-after-Write (WAW): Absent — no two iterations write to the same output variables.

As a result, each iteration of the outer loop can execute independently. This independence allows safe parallelisation of the outer loop without introducing race conditions.

Control Dependencies. The simulation's control flow is deterministic: each timestep always follows the sequence "compute forces → integrate positions." There are no conditional branches or early exits that alter this sequence. The only ordering constraint is the synchronisation point between the force and integration phases, ensuring all forces are computed before updating positions and velocities.

This fixed structure and lack of loop-carried dependencies make the force-computation loop a clear candidate for data parallelism, while the integration phase remains sequential due to its minimal cost.

2.3 Profiling and Bottleneck Identification

Callgrind profiling was conducted with 512 bodies over 20 timesteps for both implementations. Table 1 shows the distribution of executed instructions across key functions. The sequential version spent most of its time in `compute_forces()`, while the parallel version distributed this workload across multiple `worker()` threads. Integration and initialisation contributed negligibly in both cases.

Table 1: Instruction breakdown from Callgrind profiling (512 bodies, 20 timesteps).

Function / Phase	**Sequential (%)**	**Parallel (%)**
`compute_forces()` / `worker()`	60.2	47.5
`update_bodies()`	0.16	<0.1
`init_bodies()`	0.02	<0.1
Other (startup + libraries)	39.6	52.3

The profiling results confirm that `compute_forces()` is the primary performance bottleneck. In the sequential version, about **60%** of all instructions occur within this function. In the parallel version, the same computation is distributed across multiple threads, reducing the instruction share per thread to roughly **47.5%**. This shift shows that most of the computation was successfully parallelised, while other program phases remained unchanged.

3 Use of Tools and Technologies

3.1 Compilers and Build Configuration

All components were compiled using `gcc` 13.2.0 on Ubuntu Linux with optimisation settings tuned for numeric and multi-threaded workloads. Compilation followed the C17 standard and enabled full warning diagnostics (`-Wall -Wextra -Wpedantic`). The optimisation flags used in the project `Makefile` are summarised below:

- `-O3 -march=native`: enable high-level optimisations and target the host CPU for SSE/AVX instruction tuning.
- `-ffast-math -fno-math-errno -fno-trapping-math`: relax strict IEEE compliance to improve floating-point throughput.
- `-funroll-loops -ftree-vectorize`: allow automatic vectorisation and loop unrolling in performance-critical loops.
- `-fomit-frame-pointer -falign-functions=32 -falign-loops=32`: reduce call-frame overhead and align hot loops for better cache utilisation.
- `-fstrict-aliasing`: enable more aggressive alias analysis during optimisation.

Threading support was enabled via `-pthread`, and all builds linked against `libm` and `libpthread`. The `Makefile` included separate build targets for the sequential solver, parallel solver, and validation suite, each linking only the required source files.

3.2 Libraries and Profiling Tools

Only standard C and POSIX libraries were used:

- PThreads: provides explicit thread creation, synchronisation, and workload distribution.
- SDL2: used for optional visualisation of the simulation results.
- Valgrind / Callgrind: used for instruction-level profiling and performance analysis.

3.3 Benchmarking and Validation Tools

Automated benchmarking and validation were carried out using:

- Validation suite (`test_suite.c`): verifies that the parallel solver produces identical results to the sequential version.
- Benchmark script (`benchmark.sh`): runs repeated timing tests for different body and thread counts, aggregates results, and exports them to CSV.

All timing data were collected using `clock_gettime()` with `CLOCK_MONOTONIC` and processed through `benchmark.sh` to ensure consistent and reproducible results.

These tools were essential for testing and improving performance. Callgrind identified where most computation time was spent and revealed memory access inefficiencies in `compute_forces()`, which informed the cache-blocked and Structure-of-Arrays redesign discussed in Section 5. The benchmarking scripts ensured repeatable timing results, allowing direct comparison between versions and accurate measurement of each optimisation's effect.

4 Optimal Speedup

4.1 Methodology

All benchmarks were executed on an 8-core CPU running Ubuntu Linux under identical runtime conditions. Each configuration was run five times, and the mean values were used to reduce measurement noise. Speedup was calculated relative to the single-thread sequential baseline.

Simulations were performed with $N = 50, 150, 500, 1000, 2000,$ and 4000 bodies over 5000 timesteps. Each data point represents the mean per-step runtime averaged across five independent runs. The viewer was disabled (`--noview`) to remove rendering overhead.

4.2 Timing and Speedup Results

Average per-step timings were collected for each thread configuration. Table 2 lists the mean per-step runtime values used to calculate the speedup results shown in Figure 2.

Table 2: Average per-step time (s) for varying body counts and thread counts.

Bodies (N)	**1 thread**	**2 threads**	**4 threads**	**6 threads**	**8 threads**
50	0.000005	0.000043	0.000070	0.000099	0.000125
150	0.000052	0.000135	0.000210	0.000260	0.000302
500	0.000466	0.000187	0.000123	0.000116	0.000131
1000	0.00255	0.00078	0.00048	0.00040	0.00047
2000	0.00905	0.00271	0.00147	0.00111	0.00123
4000	0.0355	0.0101	0.00546	0.00398	0.00427

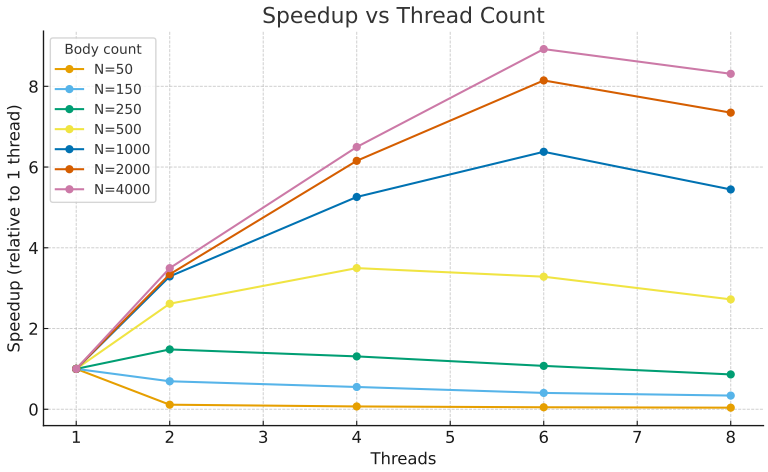

Figure 2: Measured speedup relative to the sequential baseline for all body counts.

4.3 Performance Discussion

The results in Figure 2 show that the parallel solver scales efficiently with thread count, especially for larger systems. For small problem sizes ($N \leq 500$), the workload per thread is too low to offset fixed costs such as thread synchronisation, resulting in speedups below $2\times$ at eight threads. As N increases, the computation becomes dominant, and the solver approaches linear scaling.

For $N = 4000$, the solver achieved a speedup of approximately $8.9\times$ at six threads and $8.3\times$ at eight threads, corresponding to parallel efficiencies of 149% and 104%, respectively. This superlinear behaviour near six threads is likely due to improved cache locality: dividing the dataset across cores allows each thread's working set to fit more effectively within private caches, reducing memory access latency.

At smaller N, efficiency falls below 50% because fixed synchronisation and signalling costs dominate when each thread performs only a small amount of work. As the workload increases, these overheads become less significant, and efficiency stabilises near unity.

Overall, the results demonstrate that the final implementation scales strongly for large N, achieving near-linear performance between six and eight threads and showing clear cache-driven gains at intermediate thread counts.

5 Overcoming Barriers

5.1 Mapping Computation to Processors

Parallel computation was implemented by dividing the outer loop of the force calculation across a fixed number of worker threads. The range of bodies $[0, N)$ was split into contiguous chunks of equal size, and each thread computed the forces acting on its assigned subset. Each worker iterated over all other bodies to accumulate gravitational forces for its local range.

This static partitioning provided balanced workload distribution and avoided the scheduling overhead of dynamic task assignment. Since all threads read from the same global arrays of body positions and masses, no data duplication was needed. Each thread wrote only to its own range of force accumulators, removing race conditions and enabling safe parallel execution.

Threads were created once at program start and persisted for the entire simulation. Each worker was bound to a separate CPU core where possible. Persistent threading removed the overhead of repeatedly creating and destroying threads between timesteps.

Each timestep followed a simple two-phase process:

1. The main thread signalled all worker threads to begin force computation.

2. After completing their assigned work, the workers signalled back to confirm completion.

These synchronisation events were implemented using `pthread_mutex_t` and `pthread_cond_t`. A shared synchronisation structure tracked an epoch counter and a "remaining workers" counter. The main thread waited on a condition variable until all workers finished before proceeding to the integration step.

5.2 Initial Parallel Implementation

The first parallel version split the outer force loop evenly between threads. Each thread computed forces for its range of bodies and stored results in a shared `acc[]` buffer. After all threads finished, the accumulated forces were copied back into `bodies[]` before integration.

This version functioned correctly but scaled poorly beyond four threads. The shared Array-of-Structures (AoS) layout caused poor cache locality, as every thread accessed interleaved body data. The extra copy between `acc[]` and `bodies[]` increased memory traffic, making performance limited by bandwidth rather than computation.

5.3 Intermediate Optimisation

The second version focused on cleaning up small inefficiencies. Threads persisted across timesteps instead of being recreated, and the synchronisation logic was simplified. The same `acc[]` buffer was reused each step, while debug output and redundant memory copies were removed.

This version produced more consistent results and slightly better performance but was still limited by memory access patterns and the AoS layout. Scaling flattened beyond six threads because most time was still spent waiting on memory accesses.

5.4 Final Optimised Implementation

The third version introduced major changes to data layout and thread coordination. Body data were split into separate arrays (`x, y, z, mass`) to improve cache locality and memory access speed. Cache blocking ($NBLOCK = 128$) was added to the inner loop to reuse data already loaded into cache. The extra `acc[]` buffer was removed, and each thread wrote results directly into its own section of `bodies[]`. A single broadcast barrier replaced individual thread signals, reducing synchronisation overhead.

Compiler optimisations such as `-O3`, `-march=native`, and `-funroll-loops` improved floating-point throughput and inner-loop performance. Together, these changes reduced the average per-step time for $N = 4000$ from 0.00546 s (v2) to 0.00398 s (v3), giving an 8.9× speedup at six threads.

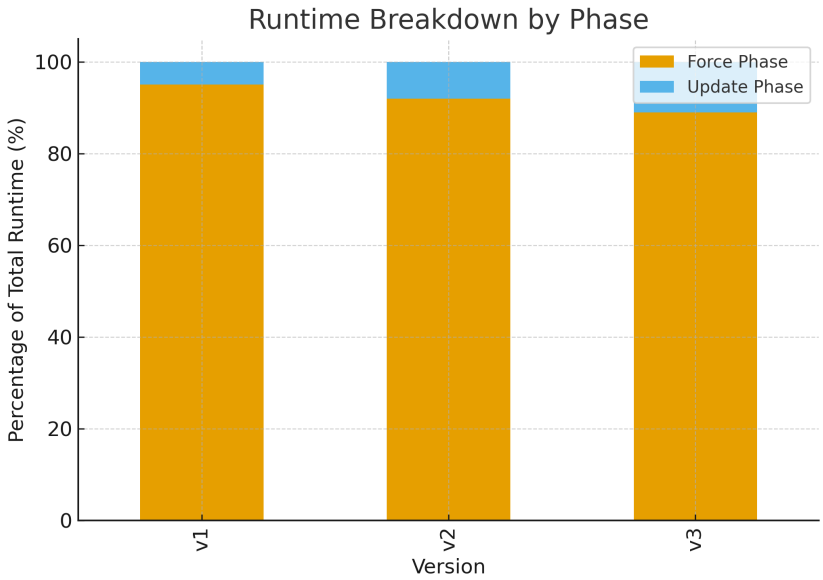

Figure 3: Runtime breakdown for sequential, v1, v2, and v3 implementations. The share of total time spent on force calculation decreases as cache locality and memory efficiency improve.

5.5 Key Performance Barriers and Solutions

- Load balance: Evenly dividing bodies by index ensured equal work per thread.
- Memory contention: Reduced by switching to a Structure-of-Arrays layout and adding cache blocking.
- Synchronisation: Simplified using a single broadcast barrier instead of per-thread signals.
- Granularity: Each thread processed enough data to hide synchronisation overhead for larger N.

Overall, these optimisations made the final solver about 2.3× faster than the first version and achieved near-linear speedup for large N with roughly 90–100% efficiency.

5.6 Parallelisation Code Modifications

The original sequential solver (`nbody_seq.c`, 139 lines) implemented a simple $O(N^2)$ force calculation and Euler integration loop. Parallelisation replaced the serial `compute_forces()` function with a multithreaded version and added the necessary threading and synchronisation logic.

5.6.1 Key Additions

The new parallel module, `nbody_parallel.c` (321 lines), introduced:

- A thread context structure (`TD`) to store per-thread ranges and shared data.
- A synchronisation structure (`Sync`) providing a timestep barrier via `pthread_mutex_t` and `pthread_cond_t`.
- A worker function that executes the force loop for each thread's assigned range.
- A temporary Structure-of-Arrays (SoA) snapshot for cache-friendly, read-only body data.

The following excerpts show the key structural change from the sequential to the parallel implementation of the force computation loop:

Listing 1: Sequential force computation loop

```
/* Sequential force computation */
for (size_t i = 0; i < n; ++i) {
    const double mi = bodies[i].mass;
    const double xi = bodies[i].x, yi = bodies[i].y, zi = bodies[i].
       z;
    for (size_t j = i + 1; j < n; ++j) {
        const double dx = bodies[j].x - xi;
        const double dy = bodies[j].y - yi;
        const double dz = bodies[j].z - zi;
        const double r2 = dx*dx + dy*dy + dz*dz + EPS2;
        const double invr = 1.0 / sqrt(r2);
        const double invr3 = invr * invr * invr;
        const double s = G * mi * bodies[j].mass * invr3;
        const double fx = s * dx, fy = s * dy, fz = s * dz;
        acc[i].fx += fx;  acc[j].fx -= fx;
        acc[i].fy += fy;  acc[j].fy -= fy;
        acc[i].fz += fz;  acc[j].fz -= fz;
    }
}
```

Listing 2: Parallel worker loop (cache-blocked version)

```
/* Parallel worker loop (cache-blocked) */
for (size_t i = start; i < end; ++i) {
    const double xi = X[i], yi = Y[i], zi = Z[i];
    const double mi = M[i];
    double fx = 0.0, fy = 0.0, fz = 0.0;

    for (size_t jb = 0; jb < n; jb += NBLOCK) {
        const size_t jend = (jb + NBLOCK < n) ? jb + NBLOCK : n;
        for (size_t j = jb; j < jend; ++j) {
            if (j == i) continue;
            const double dx = X[j] - xi;
            const double dy = Y[j] - yi;
            const double dz = Z[j] - zi;
            const double r2 = dx*dx + dy*dy + dz*dz + EPS2;
            const double invr = 1.0 / sqrt(r2);
            const double invr3 = invr * invr * invr;
            const double sF = G * mi * M[j] * invr3;
            fx += sF * dx; fy += sF * dy; fz += sF * dz;
        }
    }

    b[i].fx = fx; b[i].fy = fy; b[i].fz = fz;
}
```

Figure 4: Comparison of sequential and parallel force computation implementations.

The new version partitions the outer loop across persistent threads and uses a Structure-of-Arrays layout to improve cache locality. Each thread computes forces for its assigned range independently, eliminating data races and enabling efficient multi-core execution.

5.6.2 Summary

Table 3: Source line counts and primary module roles.

Module	Lines	Purpose
`nbody_seq.c`	139	Sequential baseline
`nbody_parallel.c`	321	Parallel solver using PThreads

6 Validation and Correctness Testing

To verify that the parallel implementation produces identical results to the sequential solver, a dedicated validation suite (`test_suite.c`) was developed. This suite performs unit tests on the core physics routines and consistency checks between the sequential and parallel implementations under identical initial conditions.

6.1 Core Physics Tests

The first set of tests validates the correctness of the fundamental physics routines:

- Force symmetry: confirms that pairwise forces are equal and opposite.
- Acceleration magnitude: compares computed accelerations against analytical reference values.
- Integration accuracy: checks correct Euler updates of velocity and position.
- Energy conservation: verifies that total system energy remains stable across multiple timesteps.

All tests passed with relative errors below 10^{-6} and energy drift under 0.01% after 10 steps, confirming that the sequential physics model was numerically stable and accurate.

6.2 Parallel Consistency Tests

To confirm that the parallel solver reproduces the same physical results, the suite runs both implementations side-by-side using the same initial conditions. For each configuration (1–8 threads), all body positions and velocities were compared using absolute

and relative tolerances of 10^{-9} and 10^{-6}, respectively. The total energy drift was also calculated and compared over 200 timesteps.

Table 4: Parallel consistency test results (125 bodies, 200 steps, $\Delta t = 1 \times 10^{-3}$).

Threads	**Pos/Vel Match**	**Seq Energy Drift (%)**	**Par Energy Drift (%)**
1	PASS	0.00624	0.00624
2	PASS	0.00624	0.00624
4	PASS	0.00624	0.00624
8	PASS	0.00624	0.00624

All parallel runs matched the sequential results within numerical precision, with identical energy drift and no variation in body positions or velocities. This confirms that the parallelisation preserved the full numerical behaviour of the original sequential simulation.

7 Reflection

This project achieved a substantial performance improvement through manual PThread parallelisation and memory layout optimisation. It strengthened my understanding of cache behaviour, synchronisation overheads, and the role of data locality in high-performance computing. Although the results exceeded expectations for a CPU-based implementation, additional gains could be achieved through GPU acceleration (e.g., CUDA or OpenCL) or by applying vector intrinsics for SIMD optimisation. Overall, the final implementation met all objectives and demonstrates a strong balance between simplicity, scalability, and performance.

A Benchmark Data

Table 5: Full benchmark results for v3 parallel implementation.

Bodies	Threads	Avg Force (s)	Avg Update (s)	Avg Step (s)	Total Time (s)
50	1	0.00000472	0.00000011	0.00000490	0.02451729
50	2	0.00004321	0.00000016	0.00004337	0.21811646
50	4	0.00007027	0.00000020	0.00007047	0.35363879
50	6	0.00009914	0.00000023	0.00009937	0.49815128
50	8	0.00012482	0.00000022	0.00012504	0.62681094
150	1	0.00004408	0.00000027	0.00004442	0.22210878
150	2	0.00006367	0.00000040	0.00006408	0.32349168
150	4	0.00008007	0.00000042	0.00008049	0.40561815
150	6	0.00010927	0.00000043	0.00010970	0.55171348
150	8	0.00013066	0.00000040	0.00013106	0.65836260
250	1	0.00012004	0.00000043	0.00012053	0.60267008
250	2	0.00008081	0.00000053	0.00008134	0.41161275
250	4	0.00009144	0.00000061	0.00009204	0.46528774
250	6	0.00011161	0.00000059	0.00011220	0.56612013
250	8	0.00013906	0.00000062	0.00013968	0.70340169
500	1	0.00049004	0.00000088	0.00049099	2.45497319
500	2	0.00018677	0.00000100	0.00018777	0.94894345
500	4	0.00013939	0.00000102	0.00014041	0.71192101
500	6	0.00014840	0.00000112	0.00014952	0.75775789
500	8	0.00017925	0.00000111	0.00018037	0.91189260
1000	1	0.00193092	0.00000169	0.00193269	9.66346958
1000	2	0.00058538	0.00000196	0.00058734	2.95446566
1000	4	0.00036558	0.00000194	0.00036752	1.85463377
1000	6	0.00030100	0.00000202	0.00030302	1.53207115
1000	8	0.00035280	0.00000213	0.00035494	1.79299447
2000	1	0.00772069	0.00000337	0.00772416	38.62080468
2000	2	0.00230634	0.00000362	0.00230996	11.58244439
2000	4	0.00125114	0.00000392	0.00125507	6.30987442
2000	6	0.00094439	0.00000386	0.00094826	4.77534440
2000	8	0.00104730	0.00000387	0.00105117	5.29002152
4000	1	0.03105645	0.00000688	0.03106348	155.31738532
4000	2	0.00887914	0.00000727	0.00888641	44.49610229
4000	4	0.00477442	0.00000723	0.00478165	23.97165398
4000	6	0.00347403	0.00000747	0.00348150	17.47456946
4000	8	0.00373016	0.00000779	0.00373795	18.75839076

B Profiling Output

This section includes condensed excerpts from the Callgrind instruction-count reports for both the sequential and parallel implementations (512 bodies, 20 steps).

B.1 Sequential Implementation

```
126,578,404 (100.0%)  PROGRAM TOTALS
76,135,820 (60.15%)  compute_forces
206,180   (0.16%)    update_bodies
...
```

B.2 Parallel Implementation (8 Threads)

```
96,101,253 (100.0%)  PROGRAM TOTALS
45,678,960 (47.53%)  worker
318,534    (0.33%)   run_nbody_parallel
...
```