On the parallelization of a N-body simulation

ANDRÉ MENDES, Universidade do Minho, Portugal

BRUNO JARDIM, Universidade do Minho, Portugal

ROBERTO FIGUEIREDO, Universidade do Minho, Portugal

This report documents the parallelization techniques implemented on a N-body simulation. This was achieved through the OpenMP[1] API. After carefully analyzing and tested different optimization locations, the resulting program had a xxx% speedup compared to the original, sequential implementation. This result shows that physical, N-body simulations are an excellent target for parallel code optimizations. As even very simple optimization techniques are able to produce a huge speedup.

Additional Key Words and Phrases: Parallel, Parallelization, Code Optimization, Simulation, Physics Simulation, OpenMP

1 Introduction

With the gradual decline of Moore's Law[2], the time of swapping an older chip for a newer one and experiencing exponential speedups is coming to an end. This, in a sense, limits the computing power a certain problem can have to still be tractable. That is, if we were dealing with a single computing core on a machine.

Fortunately, that is not the case. Nowadays, even the simplest computers have multiple computing cores. Unfortunately, this parallelism is often misused or even neglected, making the device, essentially, a single-core computer.

As such, this demands a narrative shift to parallelism as design principle. In fact, this is precisely the direction in which hardware is going, with more cores instead of faster ones, as we are reaching the physical limits of silicon. Thus, performance gains are not dependent on how fast a given CPU is, but how the parallelism of a given CPU can be exploited.

This issue becomes quite significant when it comes to N-body problems. These problems are notorious for their computational complexity, this is due to the pairwise calculations between every body in the system. As such, this requires a $O(N^2)$ time complexity, meaning that even small increases in particle count could dramatically increase the runtime of such an algorithm. In real-world scenarios, these applications require millions, if not billions, of particles in order to simulate any interesting phenomena. Thus making serial implementations of these applications impractical. This necessitates parallelization for these kinds of problems. However, parallelism isn't a solution without any drawbacks, there are a lot of considerations when parallelizing any algorithm. Such as the synchronization overheads, load imbalances, cache coherence effects, and even, memory bandwidth

limitations. This article goes over the parallelization of a N-body simulation on multicore CPUs, focusing on memory behavior, scalability and algorithmic design choices.

2 Background

2.1 Overview of the N-body Problem

The N-body problem aims to simulate the time evolution of a given system with N particles, where all particles affect one another. Each time step necessitates the computation of the net forces that act on every single particle based on the states of all others and then updating their position and velocity. This results in an $O(N^2)$ complexity on every time step. Making the problem increasingly more expensive as N grows.

Due to this design the N-body problem is a well-suited candidate for parallelization. This is further reinforced by the fact that the force computations on individual particles are independent within that time step. However, most implementations on multicore CPUs are constrained by memory bottlenecks, cache coherence effects and even synchronization overheads. Thus, achieving linear speedups is not as trivial as it might seem.

Performance is heavily influenced by memory access patterns along with the in-memory representation of the given datastructures. As they can significantly improve cache utilization and throughput. All of these characteristics make the N-body problem a very useful benchmark for parallel performance on modern CPU architectures.

2.2 Parallel Computing on Multicore CPUs with OpenMP

OpenMP[1] is a commonly used programming model for shared-memory parallelism on multicore CPUs, providing a directive-based approach for defining parallel regions, work sharing, and synchronization. It allows existing sequential code to be parallelized incrementally, while maintaining a single shared address space. However, this abstraction does not remove the underlying costs associated with thread management, synchronization, and memory access, and as such performance still depends heavily on implementation details.

Parallel execution in OpenMP is most often achieved by distributing loop iterations or tasks across a team of threads. This maps well to data-parallel workloads such as N-body force calculations. However, scalability is influenced by scheduling choices such as the granularity of parallel work, and the amount of implicit synchronization introduced by constructs like barriers and reductions. Thus, suboptimal scheduling or excessive synchronization can limit performance improvements.

Memory behavior remains a central concern in OpenMP programs. Threads operate on cache-coherent shared memory, and frequent access to shared data can increase cache coherence traffic. This can lead to issues such as false sharing and reduced cache efficiency, which thus negatively affect scalability. As such, achieving good performance with OpenMP requires careful consideration of data layout, access patterns, and synchronization placement, in addition to exposing sufficient parallelism.

3 Sequential Implementation of the N-Body Simulation

The sequential implementation of the N-body simulation follows a straightforward time-stepping approach based on direct pairwise force computation. At each simulation step, the physics engine computes gravitational accelerations for all bodies and then updates their velocities and positions using explicit numerical integration. This implementation serves as the performance and correctness baseline against which parallel versions are evaluated.

The core computation is performed in the `integrateStep` function, which operates on a vector of bodies. For a system of N bodies, the function first allocates an auxiliary array to store the acceleration vectors for each body. The accelerations are initialized to zero at the beginning of each step, ensuring that force accumulation from the previous iteration does not persist. As such, force computation and state updates are cleanly separated.

Gravitational forces are computed using a double-nested loop that iterates over all pairs of bodies. For each body *i*, the contribution from every other body *j* is accumulated, excluding self-interactions. The force calculation is based on the classical inverse-square law, with a small softening term added to the distance computation to avoid numerical instability when bodies are very close. This approach results in a computational complexity of $O(N^2)$ per simulation step, which quickly becomes the dominant cost as the number of bodies increases.

Once all accelerations have been computed, a second loop updates the velocities and positions of each body using a simple explicit integration scheme. Velocities are updated based on the computed accelerations and the time step size, and positions are then advanced using the updated velocities. This integration method is computationally inexpensive but conditionally stable, and thus appropriate for short time steps and as a baseline reference.

Overall, the sequential implementation emphasizes clarity and correctness rather than performance optimization. It makes no attempt to reduce asymptotic complexity or exploit hardware parallelism. However, its simple structure and deterministic execution order make it well suited as a reference point for evaluating the effects of parallelization, memory behavior, and scalability in subsequent implementations.

4 ILP Optimizations

The first step when optimizing the sequential implementation is exploring the inherent capabilities of modern CPUs to execute multiple instructions concurrently. Even without introducing thread-level parallelism, the baseline N-body implementation exposes opportunities for improved instruction-level parallelism through careful ordering of arithmetic operations and memory accesses. As such, changes such as reducing control dependencies, increasing the number of independent operations, and improving data locality can significantly increase single-core performance. These optimizations preserve the sequential execution model while providing a more efficient foundation for subsequent shared-memory parallelization.

```cpp
void PhysicsEngine::integrateStep(SoABodies& bodies, double dt) {
    const double G = 6.67430e-11;
    size_t n = bodies.size();

    // Reset accelerations
    std::fill(bodies.acc_x.begin(), bodies.acc_x.end(), 0.0);
    std::fill(bodies.acc_y.begin(), bodies.acc_y.end(), 0.0);
    std::fill(bodies.acc_z.begin(), bodies.acc_z.end(), 0.0);

    for (size_t i = 0; i < n; ++i) {
        double ax = 0.0, ay = 0.0, az = 0.0;

        double px = bodies.pos_x[i];
        double py = bodies.pos_y[i];
        double pz = bodies.pos_z[i];

        for (size_t j = 0; j < n; ++j) {
            // if (i == j) continue; removed to optimize performance

            double rx = bodies.pos_x[j] - px;
```

```
            double ry = bodies.pos_y[j] - py;
            double rz = bodies.pos_z[j] - pz;

            double distSq = rx * rx + ry * ry + rz * rz + 1e-9;
            double inv_dist = 1.0 / std::sqrt(distSq);
            double inv_dist3 = inv_dist * inv_dist * inv_dist;
            double factor = G * bodies.mass[j] * inv_dist3;
            ax += factor * rx;
            ay += factor * ry;
            az += factor * rz;

        }
        bodies.acc_x[i] += ax;
        bodies.acc_y[i] += ay;
        bodies.acc_z[i] += az;
    }

    for (size_t i = 0; i < n; ++i) {
        bodies.vel_x[i] += bodies.acc_x[i] * dt;
        bodies.vel_y[i] += bodies.acc_y[i] * dt;
        bodies.vel_z[i] += bodies.acc_z[i] * dt;

        bodies.pos_x[i] += bodies.vel_x[i] * dt;
        bodies.pos_y[i] += bodies.vel_y[i] * dt;
        bodies.pos_z[i] += bodies.vel_z[i] * dt;
    }
}
```

The implementation exposes a significant amount of instruction-level parallelism through careful ordering of arithmetic operations and data access patterns. In the inner force computation loop, independent floating-point operations are structured in a way that allows the CPU to overlap execution and hide instruction latencies. In particular, the accumulation of acceleration components along each axis is performed using separate scalar variables, enabling the processor to execute these operations concurrently where possible.

Several code-level decisions further improve ILP. Particle position values for the outer-loop body are loaded once into local variables before entering the inner loop, reducing redundant memory accesses and allowing the compiler to better schedule dependent instructions. Additionally, the use of a structure-of-arrays data layout results in contiguous memory accesses, which improves cache behavior and allows the processor to issue multiple independent load instructions in parallel.

The removal of the conditional branch that excludes self-interactions eliminates a potential control dependency inside the inner loop. This reduces branch misprediction penalties and allows the instruction pipeline to remain fully utilized. As such, the inner loop consists primarily of straight-line code with predictable control flow, which is well suited for both static compiler scheduling and dynamic out-of-order execution. While these changes do not alter the algorithmic complexity, they improve single-core efficiency by increasing effective instruction throughput.

5 SIMD Optimizations

References

[1] Openmp application programming interface. https://www.openmp.org, 2026. Accessed 2026.
[2] G.E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.