Mini-Paper 3: N-Body Problem

Ishaan Mathur

Abstract

In this paper, I discuss the optimization of an N-body simulation program. Through rounds of profiling using `gperftools` and `pprof`, I was able to decrease the runtime of 1) many bodies (1000) for a short time horizon (5000) by 6.9x, 2) few bodies (5) for a long time horizon (1000000000) by 2.6x, and 3) many bodies (1000) for a long time horizon (100000) by 6.9x. The main optimizations came using additional compiler flags, exploiting symmetry native to the problem, and applying parallelism.

1 Methodology

To standardize the development environment, I worked in the virtual machines provided by the course, rather than through a Docker container. To profile, I modified the Makefile to take a constant `PROFILE` to indicate whether to profile the code or not. Building the program with `make PROFILE=1` results in an executable that writes the profiling results to `my_profile.prof` when executed. Using `pprof`'s web interface by running `make view-profile`, I was able to view the profile results in different forms, whether as text or as flamegraph.

I present the optimizations individually with their corresponding Git commit hash. Some optimizations are combined by similarity. All optimizations are benchmarked using the minimum wall-clock execution time over 3 runs. Each optimization is evaluated against a baseline to ensure correctness. I noticed the output of the N-body program was quite sensitive to small differences in precision (for example by re-ordering operations in a mathematically equivalent manner), which accumulated and resulted in drastic differences over long timesteps. To accurately evaluate correctness, I measured the divergence of the last planet's `x`, `y`, `vx`, and `vy` over the first N = 200 timesteps against the baseline and ensured it was within an epsilon of 1e-5.

My code can be found at https://github.com/imathur1/598APE-HW3. The overall summary of speedups is in 3 and 1.

2 Optimizations

2.1 Baseline (f595e61)

Running the baseline without any optimizations is incredibly slow, especially on the many bodies (1000), long time horizon (100000) configuration.

2.2 Clang++ (af3779)

As in previous homeworks, I found switching from `g++` to `clang++` improved runtime across all 3 settings, likely because it tends to more aggressively optimize at `-O3` than GCC.

2.3 Memcpy, Mass caching, exploiting symmetry (6f42b89)

At the beginning of every `next` function call, the data from `planets` is copied over to `nextplanets`.

```
for (int i=0; i<nplanets; i++) {
   nextplanets[i].vx = planets[i].vx;
   nextplanets[i].vy = planets[i].vy;
   nextplanets[i].mass = planets[i].mass;
   nextplanets[i].x = planets[i].x;
   nextplanets[i].y = planets[i].y;
}
```

This can be replaced with a simple `memcpy` which is more optimized by the CPU and offers a better memory access pattern. It avoids multiple loads and stores and instead moves a contiguous memory block from one place to another.

```
memcpy(nextplanets, planets, nplanets * sizeof(
    Planet));
```

I also noticed the program repeatedly calculated the product of two planets' mass, for every combination of planets (N^2), for every timestep (T).

```
for (int i = 0; i < nplanets; i++) {
    for (int j = 0; j < nplanets; j++) {
        double invDist = planets[i].mass * planets
            [j].mass / sqrt(distSqr);
    }
}
```

Since the mass of a planet does not change, these mass products can be computed once and cached, resulting in $O(N^2T)$ less calculations.

```
for (int i = 0; i < nplanets; i++) {
    for (int j = 0; j < nplanets; j++) {
        double invDist = massProducts[i * nplanets
             + j] / sqrt(distSqr);
    }
}
```

Finally, to reduce redundant computation in the pairwise force calculation of the n-body simulation, I exploit the symmetry of gravitational interactions. The force exerted by planet i on planet j is equal in magnitude and opposite in direction to the force exerted by j on i. In the naive implementation, the force between every pair (i, j) is computed twice—once for (i, j) and once for (j, i)—leading to $O(N^2)$ work.

```
for (int i=0; i<nplanets; i++) {
    for (int j=0; j<nplanets; j++) {
      double dx = planets[j].x - planets[i].x;
      double dy = planets[j].y - planets[i].y;
      ...
      nextplanets[i].vx += dt * dx * invDist3;
      nextplanets[i].vy += dt * dy * invDist3;
   }
}
```

Instead, I iterate only over pairs where $j > i$, to ensure that each interaction is computed exactly once. The resulting force is then applied to both bodies with appropriate sign adjustments. This reduces the number of distance and force calculations by half, significantly improving performance without sacrificing accuracy.

```
for (int i=0; i<nplanets; i++) {
   for (int j=i+1; j<nplanets; j++) {
      double dx = planets[j].x - planets[i].x;
      double dy = planets[j].y - planets[i].y;
      ...
      nextplanets[i].vx += dt * dx * invDist3;
      nextplanets[i].vy += dt * dy * invDist3;
      nextplanets[j].vx += dt * -dx * invDist3;
      nextplanets[j].vy += dt * -dy * invDist3;
   }
}
```

2.4 Inline next, compiler flags (944292d)

I inlined the next function directly into the main loop, resulting in modest performance improvements. This change eliminates the overhead associated with function calls (stack frame setup and passing parameters), which can impact performance over long time horizons as in the 1000000000 timesteps configuration.

Up until this point I avoided adding compiler flags because they caused small numerical differences in floating point precision which resulted in drastic differences over long timesteps. However, after defining correctness as nondivergence within the first N = 200 timesteps (as mentioned in Methodology) I was able to introduce several compiler flags that improved performance while maintaining correctness. Namely, I added `-march=native`, `-ffast-math`, and `-flto` as I found them beneficial in previous homeworks.

2.5 `Reducing mallocs/frees (c015ba2)`

The current approach, for every timestep, does the following:

1. Allocate a new `nextplanets` buffer using `malloc`.
2. Compute all pairwise interactions using the current `planets` buffer and update the `nextplanets` buffer accordingly.
3. Free the memory used by the current `planets` buffer.
4. Set the `planets` pointer to point to the newly updated `nextplanets` buffer.

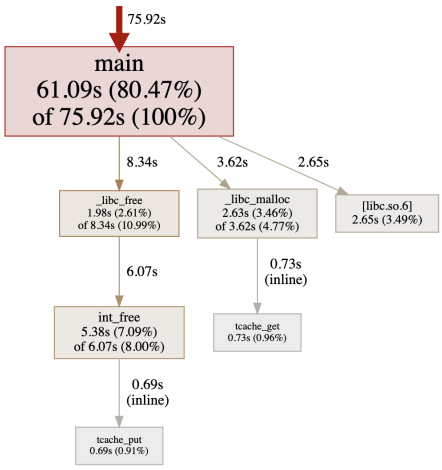

Figure 1. perf graph before malloc/free optimizations

This approach results in $O(T)$ mallocs and $O(T)$ frees. I replaced it with a double buffering technique which involves using two pre-allocated memory buffers to alternate between the current and next planet states, eliminating the need to allocate and free memory dynamically at each timestep. After each timestep, the roles of the two buffers are swapped, effectively reusing the same memory blocks throughout the simulation. This approach results in $O(1)$ mallocs and $O(1)$ frees.

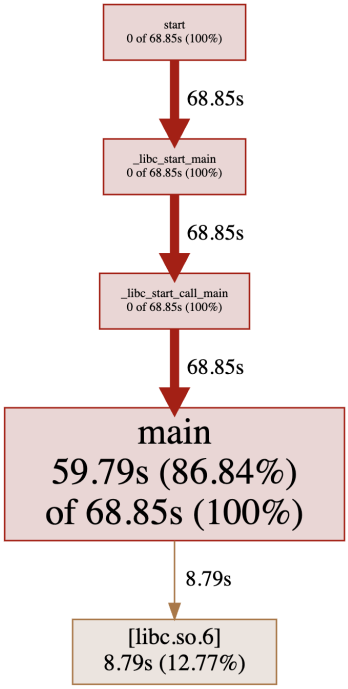

Figure 2. perf graph after malloc/free optimizations

2.6 OpenMP Parallelization (575700d)

The `nplanets` nested loop that captures interactions between each pair of planets is parallelizable. However, it is not trivial because the same iteration can update both `nextplanets[i]` and `nextplanets[j]` due to my logic for exploiting symmetry, leading to potential race conditions. To address this, I used thread-local accumulators (`thread_VX` and `thread_VY`, which are 2D buffers of shape `(nthreads, nplanets)`) to store the velocity updates for each thread independently. After the parallel loop completes, the results from all thread-local accumulators are combined to compute the final velocity changes for each planet. This approach ensures thread safety while allowing parallel computation of the gravitational interactions, significantly improving performance for a large number of planets. On the VM with 4 cores, we would expect a maximum of 4x speedup, but in reality achieve 2.5x speedup. This is likely due to the overhead of maintaining the accumulators.

For configurations with few bodies (for example 5), the overhead incurred by OpenMP actually slows down the program. Thus, I maintain a serial and parallel version of the simulation, determined by a threshold of planets I set to be 50.

2.7 `Misc changes (8cb15b5)`

Previously, I had defined the thread-local accumulators inside the timestep loop. I modified them to be defined outside the loop and set to 0 in every timestep. I also defined a chunk size for the parallelization, which I set to be $\sqrt{\texttt{nplanets}}+1$ due to improved load balancing and cache locality. I also removed the double buffering change because I realized I only needed one buffer to maintain the current state and I could update it in-place. This resulted in reduced memory overhead of maintaining two buffers, and eliminated all `memcpy`'s.

3 Review

In this section, I summarize the performance improvements achieved, describe optimizations that failed to produce tangible speedups, and highlight potential avenues for further enhancements.

3.1 Summary of Optimizations

Overall, the n-body simulation underwent a series of targeted optimizations that significantly improved performance across various configurations. Starting with compiler-level improvements, switching from `g++` to `clang++` and enabling flags like `-march=native`, `-ffast-math`, and `-flto` provided substantial runtime benefits. Memory operations were streamlined by eliminating unnecessary malloc/free calls through in-place updates. Computational efficiency was enhanced by caching mass products and exploiting the symmetry of gravitational interactions, reducing redundant calculations. To further reduce overhead, the main simulation loop was inlined to avoid function call costs. The most substantial speedups came from OpenMP parallelization using thread-local accumulators to safely handle concurrent force updates, coupled with a dynamic chunk size of $\sqrt{\texttt{nplanets}}+1$ to improve load balancing and cache efficiency. These combined changes yielded substantial performance gains, especially in large-body scenarios where parallelization could effectively be used.

3.2 Failed Optimizations

Several attempted optimizations did not lead to performance improvements and were ultimately discarded. First, I experimented with fast approximations for the square root calculation (Newton-Raphson iterations), hoping to reduce the overhead of computing distances. However, these approximations failed to outperform the highly optimized sqrt implementation provided by modern compilers and hardware. I also explored various chunk sizes for OpenMP scheduling, but found that most values either caused load imbalance or increased overhead. I ultimately empirically chose the value of $\sqrt{\texttt{nplanets}}+1$. Finally, I attempted a 2D parallelization approach using `#pragma omp parallel for collapse(2)` to simultaneously parallelize both outer and inner loops over planet pairs. However, this led to worse performance than the 1D parallelization.

3.3 Areas for Improvement

Although our optimizations yielded impressive results, there are several areas for further optimizations. Exploring vectorized instructions (such as AVX), using Profile-guided optimization (PGO), or exploring further optimizations on specialized hardware. From an algorithmic perspective, I could also explore using the Barnes-Hut Tree Algorithm for approximating an nbody simulation with $O(n\log n)$ calculations instead of $O(n^2)$ using a quadtree data structure. I could further use the Fast Multipole Method (FMM) to reduce the complexity down to $O(n)$ in certain situations.

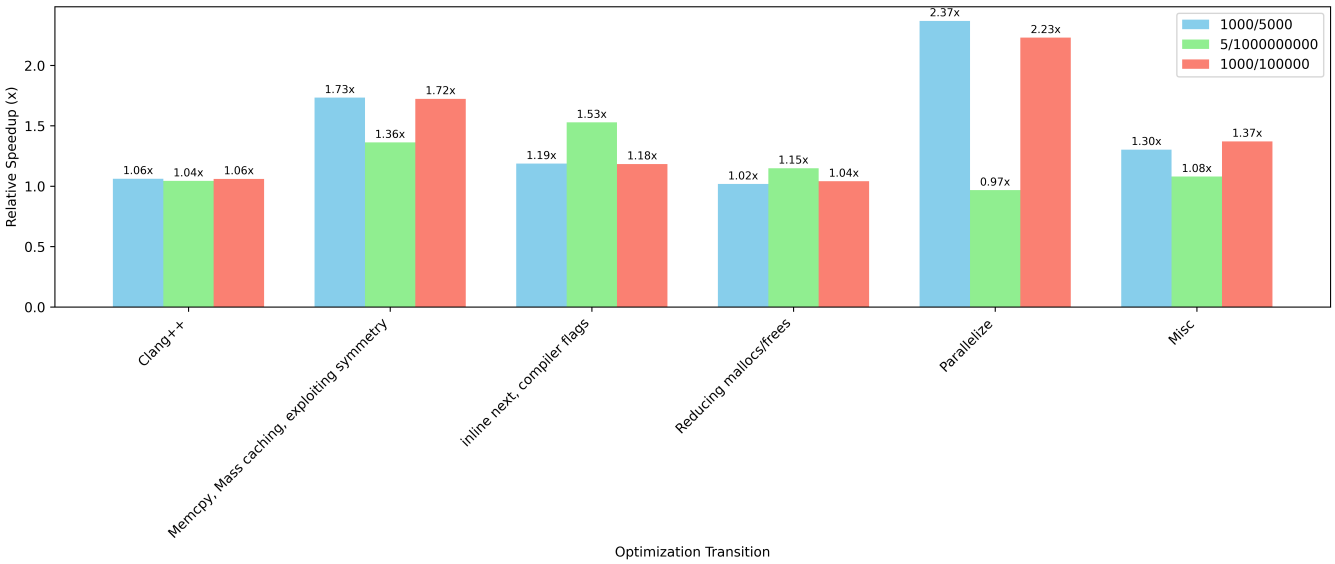

Figure 3. N-Body Simulation Optimizations

Optimization	**Time (s)**			**Commit Hash**
	1000/5000	**5/1000000000**	**1000/100000**	
Baseline	27.154503	168.426620	540.893071	f595e61
Clang++	25.581055	161.376495	510.129103	af3779
Memcpy, Mass caching, exploiting symmetry	14.756897	118.436707	296.005399	6f42b89
inline next, compiler flags	12.434470	77.496124	250.097184	944292d
Reducing mallocs/frees	12.209402	67.463959	240.309101	c015ba2
Parallelize	5.155138	69.751678	107.741165	575700d
Misc	3.957794	64.583389	78.569160	8cb15b5

Table 1. Minimum execution times for each optimization and simulation configuration.