

Phil Pennington
Phil@UsefulEngines.com

January 2024

Subject: Space Simulation Technical Interview Project

The example code provided is a particle system simulation where each particle affects every other particle. This is an N-body problem, which the implemented algorithm solves with a run-time complexity of $O(n^2)$ as it uses a nested loop to compare each particle with every other. The solution is best optimized using the [Barnes-Hut](#) algorithm with a run-time complexity of $O(n \log n)$.

Note that the original code and iterative optimization approaches are archived via the following github repository.

UsefulEngines/NBodyDemos: Experimenting with C++ 20 standard library parallel constructs. (<https://github.com/UsefulEngines/NBodyDemos>)

An initial review of the solution indicates the following *semantic* observations:

Realism: A more realistic gravitational N-Body solution would account for additional variables including Particle Acceleration and perhaps Kinetic or Potential Energy. Here is example Particle struct with these member variables.

```
struct Particle {  
    double x, y, z;           // position  
  
    double mass;  
  
    double vx, vy, vz;  
  
    double ax, ay, az;       // adding Acceleration  
  
    double kinetic_energy;    // adding Kinetic Energy  
};
```

The classic N-Body solution demo leverages the acceleration inherent in Gravitational attraction (G) between large particles with Force calculations as follows.

```

struct Particle
{
    double x, y, z;           // position
    double vx, vy, vz;       // velocity
    double fx, fy;           // force
    double mass;              // mass
};

static double Distance(double x1, double y1, double x2, double y2) {
    return sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
}

static double Force(double mass1, double mass2, double r) {
    return G * mass1 * mass2 / (r * r + 1e-10);
}

```

With Newtonian mechanics, for every action, there is an equal and opposite reaction. This enables us to half the total number of *Force* calculations. If the force on particle q is due to particle k , F_{qk} , then the force on k due to q is $-F_{qk}$.

Initial Optimization Ideas

Use more efficient data structures: The Particles are stored in a `std::vector`. The vector provides contiguous memory, which is beneficial for cache locality and SIMD operations. It is usually the default choice for a collection of elements unless there is a specific reason to use another container type. Memory layout of the Particles data-structure requires consideration of CPU versus GPU architectures and potential struct packing and alignment. CPU cache lines are sensitive to data alignment with consideration of an “Array of Structures” versus a “Structure of Arrays” memory layout.

Different Algorithms: Consider using spatial partitioning data structures like [Octrees](#) for efficient force calculation. This can significantly reduce the number of particle interactions you need to compute. The Barnes-Hut algorithm leverages this approach.

Multi-threading: The current *update* function could easily be multi-threaded because each particle operation is independent. There should be one thread updating a particle's velocity and another updating its position. This way, while one thread is calculating the next position of a particle, the other is already updating the velocity of the next one. Force calculations between particles are independent and can be parallelized. Additionally, modern CPUs support SIMD (Single Instruction, Multiple Data) vectorization operations. Using libraries like Intel's MKL or OpenMP to distribute the computation across multiple cores. In the Update method, each particle's position is updated individually. This can be optimized using vectorized operations.

Optimize force calculation: The calculation of the force between particles is a bottleneck as it includes a square root operation in the distance calculation and division to determine the force. For the distance, you could calculate the squared distance and use it in the calculations. You could change the force formula to work with the squared distance, removing the cube root.

Use a simplified simulation (approximation): Accurately simulating the interaction of every particle with every other particle is computationally expensive. If 100% accuracy is not required for the use case, consider using approximation algorithms like the Barnes-Hut algorithm.

Reserve vector capacity: When you know how many particles you are going to simulate, you can use the reserve method to pre-allocate memory for the `std::vector`. This will avoid unnecessary reallocations as the vector grows.

Reducing function Calls: Within the *Update* function, avoid redundant calls to `std::sqrt` in the inner loop. Since the *force* calculation only requires the inverse square of the distance, you can eliminate the square root operation.

Improved Memory Access Patterns: Data structures may be reorganized to ensure better CPU cache utilization. Considerations include trade-offs between Structure of Arrays (SoA) versus Array of Structures (AoS) to improve cache locality. Additional considerations include accounting for CPU-group distribution across Non-Uniform Memory Access (NUMA) CPU sockets.

Avoid division by Zero: Within the original *Force* calculation, there's a potential for division by zero. This can be mitigated by adding a small epsilon value to the denominator.

Use a better random number generator: Use of `rand()` for particle initialization is not ideal. More recent pseudo-random number generators are available via the C++ `<random>` library.

Compile time optimizations: Utilize compiler optimizations when compiling code (e.g. *Whole Program Optimization, Favor Speed over Size*, etc.).

Profiling: Utilize the C++ Profiler to identify code segments that consume the most time and research optimizations accordingly.

Solution Scenarios Explored

The following solution scenarios have been explored corresponding to Part 1 of the exercise; that is, optimize the original serial solution using parallel algorithms. Note that the full source code with Visual Studio C++ 2022 project files is hosted via the github repository mentioned previously.

1. The original serial solution. See Appendix A.
2. Using the C++20 Standard Library. See Appendix B.
3. Using the OpenMP Library. See Appendix C.
4. Using C++20 and Structure of Arrays memory model. See Appendix D.

Runtime Summary

With *no optimization profiling*, the release builds of each project indicate the following runtimes for particle systems of sizes 100, 1000, and 10000. The time series execution step was maintained at 100 iterations. Note that, as of this writing, the C++ Struct of Arrays example has not been correctly compiled.

Immediate observations conclude that the C++ Standard Library example illustrates the most optimal runtime. And, as noted above, the example utilizing a Structure of Arrays instead of the Array of Structures memory model remains to be compiled.

See the following table that summarizes each runtime scenario.

Project	NumParticles	NumSteps	Runtime (microsecs)
Serial	100	100	2081
	1000	100	208210
	10000	100	20823451
C++20 Standard Lib	100	100	4540
	1000	100	117024
	10000	100	1301815
OpenMP	100	100	6356
	1000	100	41705
	10000	100	1484707
C++20 StructOfArrays	100	100	
	1000	100	


```

        particles.push_back(p);
    }
}

void update() {
    for (int i = 0; i < particles.size(); i++) {
        for (int j = i + 1; j < particles.size(); j++) {
            double dx = particles[j].x - particles[i].x;
            double dy = particles[j].y - particles[i].y;
            double dz = particles[j].z - particles[i].z;

            double dist = std::sqrt(dx * dx + dy * dy + dz * dz);
            double force = particles[i].mass * particles[j].mass / (dist * dist * dist);

            particles[i].vx += force * dx;
            particles[i].vy += force * dy;
            particles[i].vz += force * dz;

            particles[j].vx -= force * dx;
            particles[j].vy -= force * dy;
            particles[j].vz -= force * dz;
        }
    }

    for (auto& p : particles) {
        p.x += p.vx;
        p.y += p.vy;
        p.z += p.vz;
    }
}
};

```

E:\GIT\UsefulEngines\NBodyDemos\x64\Release>demo00.exe 100

Original nBody Demo without optimizations...

NumParticles	NumberOfSteps	Runtime(microseconds)
100	100	2081

E:\GIT\UsefulEngines\NBodyDemos\x64\Release>demo00.exe 1000

Original nBody Demo without optimizations...

NumParticles	NumberOfSteps	Runtime(microseconds)
1000	100	208210

E:\GIT\UsefulEngines\NBodyDemos\x64\Release>demo00.exe 10000

Original nBody Demo without optimizations...

NumParticles	NumberOfSteps	Runtime(microseconds)
10000	100	20823451

Appendix B: Solution Using C++20 Standard Library Parallelization

```

class ParticleSystem
{
public:
    std::vector<Particle> particles;

    ParticleSystem(int num_particles)

```

```

{
    //
    // Optimization: reserve space for all particles up front.
    //
    particles.reserve(num_particles);

    //
    // Thoughts on thread safety...
    //
    // The following code is not using a Parallel for loop because the std::vector::push_back() is not
    thread-safe.
    // And, the std::random_device is not thread-safe. We could rewrite this so that each thread has its own
    // std::random_device, but that would be inefficient too. Research further...
    //

    // Use a better random number generation scheme than rand()
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> dis(0.0, 1000.0);

    // Initialize particles... Thread-safe, but not parallelized.
    for (int i = 0; i < num_particles; ++i) {
        Particle p;
        // random position
        p.pos[0] = 1e18 * exp(-1.8) * (0.5 - dis(gen));
        p.pos[1] = 1e18 * exp(-1.8) * (0.5 - dis(gen));
        p.pos[2] = 1e18 * exp(-1.8) * (0.5 - dis(gen));
        // random velocity
        p.vel[0] = 1e18 * exp(-1.8) * (0.5 - dis(gen));
        p.vel[1] = 1e18 * exp(-1.8) * (0.5 - dis(gen));
        p.vel[2] = 1e18 * exp(-1.8) * (0.5 - dis(gen));
        // random mass
        p.mass = 1.98892e30 * dis(gen) * 10 + 1e20;
        // initial force
        p.acc[0] = 0.0;
        p.acc[1] = 0.0;
        p.acc[2] = 0.0;
        particles.push_back(p); // push_back() thread safety issues?
    }
}

void update()
{
    // Reset accelerations to zero
    for (auto& p : particles) {
        p.acc[0] = p.acc[1] = p.acc[2] = 0.0;
    }

    // Calculate forces in parallel using C++20 parallel algorithms
    std::for_each(std::execution::par, particles.begin(), particles.end(),
        [&](Particle& pi) {
            for (auto& pj : particles) {
                if (&pi != &pj) {
                    double dx = pj.pos[0] - pi.pos[0];
                    double dy = pj.pos[1] - pi.pos[1];
                    double dz = pj.pos[2] - pi.pos[2];

                    //
                    // Optimization: Avoid redundant calls to std::sqrt in this inner loop

```

```

        //
        double dist_squared = dx * dx + dy * dy + dz * dz;
        double dist_sixth = dist_squared * dist_squared * dist_squared;
        double force = pi.mass * pj.mass / (dist_sixth + 1e-12); // Small epsilon to avoid division
by zero

        pi.acc[0] += force * dx;
        pi.acc[1] += force * dy;
        pi.acc[2] += force * dz;
    }
}
};

// Update velocity and position. Parallelize this loop too?
for (auto& p : particles) {
    for (int i = 0; i < 3; ++i) {
        p.vel[i] += p.acc[i];
        p.pos[i] += p.vel[i];
    }
}
};

```

E:\GIT\UsefulEngines\NBodyDemos\x64\Release>NBodyCpp20.exe 100

```

Modified nBody Demo with C++20 standard library optimizations...
NumParticles      NumberOfSteps      Runtime(microseconds)
100               100               4540

```

E:\GIT\UsefulEngines\NBodyDemos\x64\Release>NBodyCpp20.exe 1000

```

Modified nBody Demo with C++20 standard library optimizations...
NumParticles      NumberOfSteps      Runtime(microseconds)
1000              100               117024

```

E:\GIT\UsefulEngines\NBodyDemos\x64\Release>NBodyCpp20.exe 10000

```

Modified nBody Demo with C++20 standard library optimizations...
NumParticles      NumberOfSteps      Runtime(microseconds)
10000             100               1301815

```

Appendix C: Solution Using OpenMP Library Parallelization

```

class ParticleSystem
{
public:
    std::vector<Particle> particles;

    ParticleSystem(int num_particles)
    {
        //
        // Optimization: reserve space for all particles up front.
        //
        particles.reserve(num_particles);

        //
        // Thoughts on thread safety...
    }
};

```



```

//
// The following code is not using a Parallel for loop because the std::vector::push_back() is not thread-
safe.
// And, the std::random_device is not thread-safe. We could rewrite this so that each thread has its own
// std::random_device, but that would be inefficient too. Research further...
//

// Use a better random number generation scheme than rand()
std::random_device rd;
std::mt19937 gen(rd());
std::uniform_real_distribution<> dis(0.0, 1000.0);

// Initialize particles...
for (int i = 0; i < num_particles; ++i) {
    Particle p;
    // random position
    p.pos[0] = 1e18 * exp(-1.8) * (0.5 - dis(gen));
    p.pos[1] = 1e18 * exp(-1.8) * (0.5 - dis(gen));
    p.pos[2] = 1e18 * exp(-1.8) * (0.5 - dis(gen));
    // random velocity
    p.vel[0] = 1e18 * exp(-1.8) * (0.5 - dis(gen));
    p.vel[1] = 1e18 * exp(-1.8) * (0.5 - dis(gen));
    p.vel[2] = 1e18 * exp(-1.8) * (0.5 - dis(gen));
    // random mass
    p.mass = 1.98892e30 * dis(gen) * 10 + 1e20;
    // initial force
    p.acc[0] = 0.0;
    p.acc[1] = 0.0;
    p.acc[2] = 0.0;
    particles.push_back(p);
}

void update()
{
    int n = particles.size();

    // Reset accelerations to zero
    #pragma omp parallel for
    for (int i = 0; i < n; ++i) {
        particles[i].acc[0] = particles[i].acc[1] = particles[i].acc[2] = 0.0;
    }

    // Calculate forces in parallel using OpenMP
    #pragma omp parallel for
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (i != j) {
                double dx = particles[j].pos[0] - particles[i].pos[0];
                double dy = particles[j].pos[1] - particles[i].pos[1];
                double dz = particles[j].pos[2] - particles[i].pos[2];

                double dist_squared = dx * dx + dy * dy + dz * dz;
                double dist_sixth = dist_squared * dist_squared * dist_squared;
                double force = particles[i].mass * particles[j].mass / (dist_sixth + 1e-12);

                particles[i].acc[0] += force * dx;
                particles[i].acc[1] += force * dy;
                particles[i].acc[2] += force * dz;
            }
        }
    }
}

```

```

}
}
}

```

```

// Update velocity and position
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
for (int k = 0; k < 3; ++k) {
particles[i].vel[k] += particles[i].acc[k];
particles[i].pos[k] += particles[i].vel[k];
}
}
}
};

```

E:\GIT\UsefulEngines\NBodyDemos\x64\Release>nbodyopenmp.exe 100

Modified nBody Demo with OpenMP library optimizations...

NumParticles	NumberOfSteps	Runtime(microseconds)
100	100	2643

E:\GIT\UsefulEngines\NBodyDemos\x64\Release>nbodyopenmp.exe 1000

Modified nBody Demo with OpenMP library optimizations...

NumParticles	NumberOfSteps	Runtime(microseconds)
1000	100	267168

E:\GIT\UsefulEngines\NBodyDemos\x64\Release>nbodyopenmp.exe 10000

Modified nBody Demo with OpenMP library optimizations...

NumParticles	NumberOfSteps	Runtime(microseconds)
10000	100	27066009

Appendix D: Solution Using C++20 and a Structure of Arrays Memory Model

```

class ParticleSystem
{
public:
    std::vector<std::array<double, 3>> positions;
    std::vector<std::array<double, 3>> velocities;
    std::vector<std::array<double, 3>> accelerations;
    std::vector<double> masses;

    ParticleSystem(int num_particles)
    {
        // Reserve space for all particles up front
        positions.reserve(num_particles);
        velocities.reserve(num_particles);
        accelerations.reserve(num_particles);
        masses.reserve(num_particles);

        // Use a better random number generation scheme
        std::random_device rd;
        std::mt19937 gen(rd());
        std::uniform_real_distribution<> dis(0.0, 1000.0);

        // Initialize particles...
        for (int i = 0; i < num_particles; ++i) {
            std::array<double, 3> pos = { { 1e18 * exp(-1.8) * (0.5 - dis(gen)),

```

```

        1e18 * exp(-1.8) * (0.5 - dis(gen)),
        1e18 * exp(-1.8) * (0.5 - dis(gen))} };

std::array<double, 3> vel = { { 1e18 * exp(-1.8) * (0.5 - dis(gen)),
        1e18 * exp(-1.8) * (0.5 - dis(gen)),
        1e18 * exp(-1.8) * (0.5 - dis(gen))} };

double mass = 1.98892e30 * dis(gen) * 10 + 1e20;

positions.push_back(pos);
velocities.push_back(vel);
accelerations.push_back({ 0.0, 0.0, 0.0 });
masses.push_back(mass);
}
}

void update()
{
    // Reset accelerations to zero
    for (auto& acc : accelerations) {
        acc.fill(0.0);
    }

    //
    // NOTE: The std::execution::par algorithm below doesn't yet work... debugging...
    //

    // Calculate forces in parallel using C++20 parallel algorithms
    std::for_each(std::execution::par, positions.begin(), positions.end(),
        [&](std::array<double, 3>& pi_pos, size_t i) {
            for (size_t j = 0; j < positions.size(); ++j) {
                if (i != j) {
                    std::array<double, 3>& pj_pos = positions[j];
                    double dx = pj_pos[0] - pi_pos[0];
                    double dy = pj_pos[1] - pi_pos[1];
                    double dz = pj_pos[2] - pi_pos[2];

                    double dist_squared = dx * dx + dy * dy + dz * dz;
                    double dist_sixth = dist_squared * dist_squared * dist_squared;
                    double force = masses[i] * masses[j] / (dist_sixth + 1e-12);

                    accelerations[i][0] += force * dx;
                    accelerations[i][1] += force * dy;
                    accelerations[i][2] += force * dz;
                }
            }
        });

    // Update velocity and position
    for (size_t i = 0; i < positions.size(); ++i) {
        for (int j = 0; j < 3; ++j) {
            velocities[i][j] += accelerations[i][j];
            positions[i][j] += velocities[i][j];
        }
    }
};

```

