

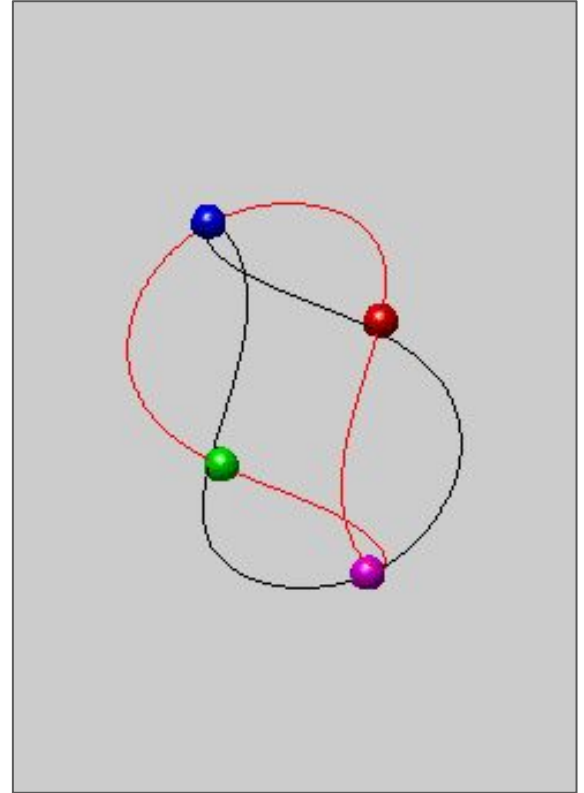
N Body Problem

Clara Almeida
David Guerrero-Pantoja

The N-Body Problem

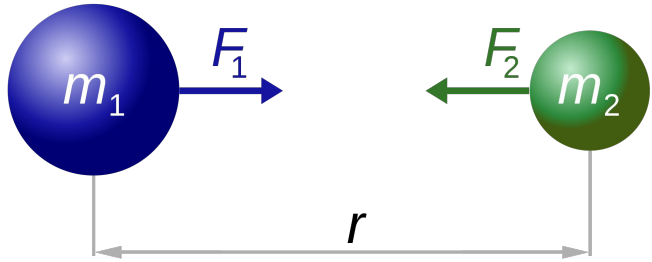
The N-Body problem in physics involves predicting the motion of celestial objects (bodies) that influence each other gravitationally.

- Each object exerts a force on every other object
- Repeated differential calculations must be done for each body in the system, which can be computationally intensive
- Parallelizing these calculations vastly improves efficiency



N-Body simulation

Gravitational force calculation formula
(Newton's law of gravitation):



$$F_1 = F_2 = G \frac{m_1 \times m_2}{r^2}$$

(Where G is the gravitational constant, m1 and m2 are the respective masses of each body)

This program will use a simplified version of Newton's law, by omitting G and assuming the mass of each body to be equal to 1. This simplifies the calculations and makes the simulation focus on the relative impacts of the bodies on each other rather than their absolute force values:

$$F = \frac{1}{r^2}$$

We will also add a softening constant (ϵ), which is a small floating point number that is used to pad the distance squared term to prevent potential division by 0 or extremely large force calculations if the distance between the bodies is very small:

$$F = \frac{1}{r^2 + \epsilon}$$

Calculations

Because we are treating force as vector in a 3-dimensional space, we measure each component using the following equations, where dx, dy, dz are the differences in positions of the bodies:

$$F_x = dx \cdot \frac{1}{(dx^2 + dy^2 + dz^2 + \epsilon)^{\frac{3}{2}}}$$

$$F_y = dy \cdot \frac{1}{(dx^2 + dy^2 + dz^2 + \epsilon)^{\frac{3}{2}}}$$

$$F_z = dz \cdot \frac{1}{(dx^2 + dy^2 + dz^2 + \epsilon)^{\frac{3}{2}}}$$

Since the same calculations are done for each body, we will submit the following kernel to be executed on each body:

```
float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f;

for (int j = 0; j < nBodies; j++) {
    float dx = data[j].x - data[i].x;
    float dy = data[j].y - data[i].y;
    float dz = data[j].z - data[i].z;
    float distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;
    float invDist = 1.0f / sqrtf(distSqr);
    float invDist3 = invDist * invDist * invDist;

    Fx += dx * invDist3; Fy += dy * invDist3; Fz += dz * invDist3;
}

//updates velocities based on force
data[i].vx += dt*Fx; data[i].vy += dt*Fy; data[i].vz += dt*Fz;
```

Note: is an N² algorithm and is common for the number of Bodies to be above 40000

Step by Step Explanation of Previous Kernel:

First we initialize the force in all directions to 0:

```
float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f;
```

Next, we calculate the differences in positions for each direction between body i and j:

```
float dx = data[j].x - data[i].x;  
float dy = data[j].y - data[i].y;  
float dz = data[j].z - data[i].z;
```

Then we square the distances and add a softening constant:

```
float distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;
```

Now taking the square root of distSqr to get the magnitude of distance, we can calculate the inverse of the distance cubed:

```
float invDist = 1.0f / sqrtf(distSqr);  
float invDist3 = invDist * invDist * invDist;
```

Step by Step Explanation of Previous Kernel(cont.):

Now, we can use our previous component equations to find each force component:


```
Fx += dx * invDist3; Fy += dy * invDist3; Fz += dz * invDist3;
```

And use Euler integration to update the velocities with these new force components:

```
data[i].vx += dt*Fx; data[i].vy += dt*Fy; data[i].vz += dt*Fz;
```

Full C Code for Force on Bodies

```
void bodyForce(Body *p, float dt, int n) {  
    #pragma omp parallel for schedule(dynamic)  
    for (int i = 0; i < n; i++) {  
        float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f;  
  
        for (int j = 0; j < n; j++) {  
            float dx = p[j].x - p[i].x;  
            float dy = p[j].y - p[i].y;  
            float dz = p[j].z - p[i].z;  
            float distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;  
            float invDist = 1.0f / sqrtf(distSqr);  
            float invDist3 = invDist * invDist * invDist;  
  
            Fx += dx * invDist3; Fy += dy * invDist3; Fz += dz * invDist3;  
        }  
  
        p[i].vx += dt*Fx; p[i].vy += dt*Fy; p[i].vz += dt*Fz;  
    }  
}
```



Here we have the actual C code for computing the force of every particle onto every other particle.

Since f_x , f_y and f_z are specific to each i , there is no loop dependency so you can parallelize the i loop.

Because every loop cycle adds to f_x , f_y and f_z . The next cycle has a different value of f_x , f_y and f_z than it started with. Meaning there is a loop dependency so you can't parallelize the j loop.

SYCL Code for Force on Bodies

```
void bodyForce(Body *data, float dt, int nBodies, queue &q){
    q.submit([&](handler& h) { //compute forces
        h.parallel_for(range<1>(nBodies), [=](id<1> i) {
            float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f;

            for (int j = 0; j < nBodies; j++) {
                float dx = data[j].x - data[i].x;
                float dy = data[j].y - data[i].y;
                float dz = data[j].z - data[i].z;
                float distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;
                float invDist = 1.0f / sqrtf(distSqr);
                float invDist3 = invDist * invDist * invDist;

                Fx += dx * invDist3; Fy += dy * invDist3; Fz += dz * invDist3;
            }

            data[i].vx += dt*Fx; data[i].vy += dt*Fy; data[i].vz += dt*Fz;
        });
    }).wait(); //check syntax
}
```

As you can see the only change is with the first for loop now being a parallel for.

This makes a computationally intensive problem much faster.

This is an N algorithm. But running on N nodes in parallel :-)

C Code for computing New Position

```
for (int iter = 1; iter <= nIters; iter++) {  
    //start timer here  
    start = omp_get_wtime();  
  
    bodyForce(p, dt, nBodies); // compute interbody forces  
  
    for (int i = 0 ; i < nBodies; i++) { // integrate position  
        p[i].x += p[i].vx*dt;  
        p[i].y += p[i].vy*dt;  
        p[i].z += p[i].vz*dt;  
    }  
  
    //end timer here  
    end = omp_get_wtime();  
  
    double timerElapsed = end - start;  
    //get value of the elapsed time  
    if (iter > 1) { // First iter is warm up  
        totalTime += timerElapsed;  
    }  
}
```

The first for simulates time, it runs the code as many times as you want. This cannot be parallelized because you need the previous output as inputs for the next iteration.

The second for loop you can parallelize because every i is a different particle, so you can compute the new positions for every particle, at the same time.

SYCL Code for Computing New Position

```
for (int iter = 1; iter <= nIters; iter++) {  
    StartTimer();  
  
    bodyForce(data, dt, nBodies, q);  
  
    q.submit([&](handler& h) { //integrate positions  
        h.parallel_for(range<1>(nBodies), [=](id<1> i) {  
            data[i].x += data[i].vx*dt;  
            data[i].y += data[i].vy*dt;  
            data[i].z += data[i].vz*dt;  
        });  
    }).wait(); //wait also  
  
    const double tElapsed = GetTimer() / 1000.0;  
    if (iter > 1) { // First iter is warm up  
        totalTime += tElapsed;  
    }  
}
```

First for is unchanged.

Second for is now a parallel for.

Wait at the end is to synchronize the results and to make sure that all the calculations have finished so that the time Elapsed is correct.

Random in Parallel

This is how you randomize in parallel. You can't use `rand()` because it is not thread safe. You can parallelize this for because no loop dependencies between each `i`.

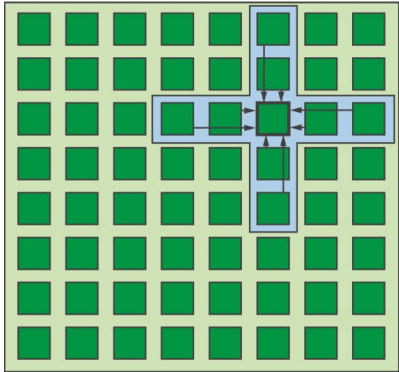
The code for randomizing in parallel is included in the oneMKL library

For these two problems all initialization can be done in parallel.

```
void randomizeBodies(Body *data, int n, queue &q) {
    q.submit([&](handler& h) {
        h.parallel_for(range<1>(n), [=](item<1> idx) {
            auto offset = idx.get_linear_id();
            oneapi::dpl::minstd_rand engine(seed, offset);
            oneapi::dpl::uniform_real_distribution<float> distr(-1.0f, 1.0f);
            data[offset].x = distr(engine);
            data[offset].y = distr(engine);
            data[offset].z = distr(engine);
            data[offset].vx = distr(engine);
            data[offset].vy = distr(engine);
            data[offset].vz = distr(engine);
        });
    }).wait();
}

void randomizeBodies(Body *data, int n) {
    for (int i = 0; i < n; i++) {
        data[i].x = 2.0f * (rand() / (float)RAND_MAX) - 1.0f;
        data[i].y = 2.0f * (rand() / (float)RAND_MAX) - 1.0f;
        data[i].z = 2.0f * (rand() / (float)RAND_MAX) - 1.0f;
        data[i].vx = 2.0f * (rand() / (float)RAND_MAX) - 1.0f;
        data[i].vy = 2.0f * (rand() / (float)RAND_MAX) - 1.0f;
        data[i].vz = 2.0f * (rand() / (float)RAND_MAX) - 1.0f;
    }
}
```

Stencil Code in 2D (2D-Jacobi)



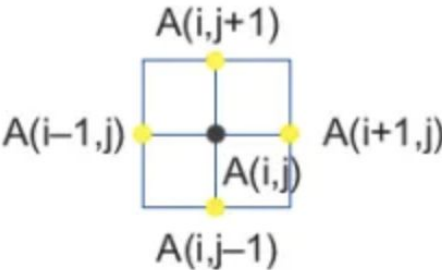

$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

Figure 1. Mathematical representation of the Laplace Equation — as it is possible to observe, it is a simple stencil equation where the solution is the average of the heat value of its immediate neighbors.

2D stencil in C

Both for loops can be parallelized because every combination of i and j is a different calculation, none of which are dependent on each other.

```
void compute_average_host(int nrows, int ncols, double * average, double * T) {  
    int i,j;  
    for(i=1;i<nrows-1;i++) {  
        for(j=1;j<ncols-1;j++) {  
            average[i*ncols+j] = (1.0/5.0)*(T[(i-1)*ncols+j]+  
                                           T[i*ncols+j]+  
                                           T[(i+1)*ncols+j]+  
                                           T[i*ncols+j+1]+  
                                           T[i*ncols+j-1]  
                                           );  
        }  
    }  
}
```

The math looks like this because in reality it is a 1D array pretending to be a 2D array. So every time you go the amount of columns in the array it acts as a new row. Then adding j gets you to the correct row column value.

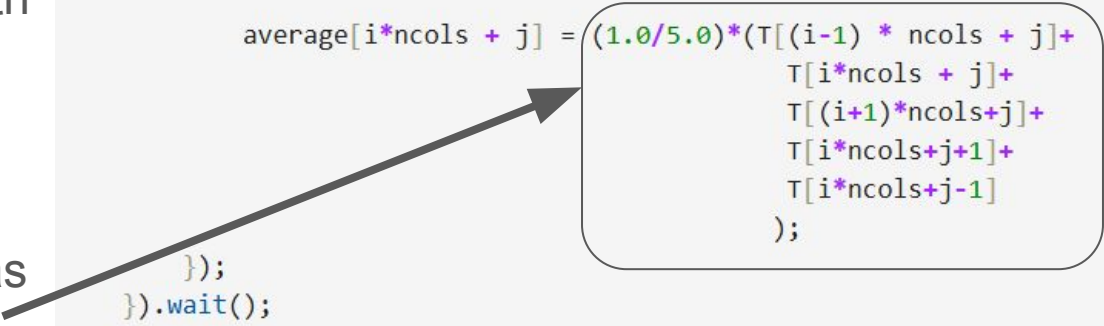
$$T[i][j] = T[i * ncols + j]$$

2D Stencil in SYCL

You skip the first and last row and column (Halo, can also be padded with 0's or by coy)

The actual math is exactly the same as on previous equation

```
void compute_average_sycl_2D(int nrows, int ncols, double *average, double *T, queue &q) {  
    q.submit([&](handler &h) {  
        h.parallel_for(range<2>(nrows-2, ncols-2), [=](id<2> idx) {  
            int i = idx[0] + 1;  
            int j = idx[1] + 1;  
  
            average[i*ncols + j] = (1.0/5.0)*(T[(i-1) * ncols + j] +  
                T[i*ncols + j] +  
                T[(i+1)*ncols+j] +  
                T[i*ncols+j+1] +  
                T[i*ncols+j-1]  
                );  
        });  
    }).wait();  
}
```



Simulating Time

You cannot parallelize this for loop because it simulates time, meaning that every output is the new input, a loop dependency.

```
start = clock();  
for(iter=0; iter<itmax; iter+=2) {  
    compute_average_sycl_2D(nrows, ncols, average, T, q);  
    compute_average_sycl_2D(nrows, ncols, T, average, q);  
}  
end = clock();  
  
cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
```

Instead of copying average into T for the next iteration it does two iterations in one. This works because after the first iteration the output is the new input. We take advantage of the fact that compute average rewrites to save us a from having to copy