COMP90025 project 2
Student Name: Yitong Chen
Login Name: frankiechan913

Parallelization and Implementation of the N-body Problem

Introduction

In physics, the n-body problem is the problem of simulating the movements of particles, and how they interact with each other gravitationally. In computer science, we just use the theory with some simplifications and algorithms to simulate the system.

In this project, we are trying to implement a parallelized version of the 2-dimensional *n*-body simulation.

Fundamental solutions

The force given by Newton's universal law of gravity, is given by

$$\vec{F}_{ij} = G \cdot \frac{m_i \cdot m_j}{r_{ij}^3} \cdot \vec{r}_{ij}$$

Where G is universal gravitational factor, i and j are two different particles.

There is a theory called Euler's method, which is cheap to compute the positions and velocities. However, the Euler's method is not very accuracy, that we could do some improvement called midpoint method.

We could predict the position at the middle of the timestep first, and the acceleration is already known because we have calculated the force. The i-th particle position and velocity is updated as [1]:

$$\vec{v}_i^{(t+1)} = \vec{v}_i^{(t)} + \vec{a}_i^{\left(t+\frac{1}{2}\right)} \cdot \Delta t$$
$$\vec{r}_i^{(t+1)} = \vec{r}_i^{(t)} + \left(\vec{v}_i^{(t)} + \vec{v}_i^{(t+1)}\right) \cdot \frac{\Delta t}{2}$$

The simulation will loop all particles, evolves by timestep (*dt*), each step we will first reset all forces (acceleration) to zero, and then recalculate the force to update the information of bodies.

Below is what we do in code:

```
for(int i = 0; i < nbodies; i++)
        {
            // leap frog
            bodies[i].x += (timestep/2)*bodies[i].vx; // pos += vel * dt/2
            bodies[i].y += (timestep/2)*bodies[i].vy;

            bodies[i].vx += bodies[i].ax * timestep; // vel += accel * dt
            bodies[i].vy += bodies[i].ay * timestep;

            bodies[i].x += (timestep/2)*bodies[i].vx; // pos += vel * dt/2
            bodies[i].y += (timestep/2)*bodies[i].vy;
        }
```

Figure 1. Leapfrog implementation

As for the computation, the complexity is unavoidable since each particle is forced by all the other particles, also the force is dependent on the distances, the forces should be recalculated as the particles move among the space in timestep. We need to set timestep small enough, like 0.1 to get more accuracy results in computer simulation.

We will initialize the system by generating random positions between -500 and 500 in coordinates, with velocity (-5, 5) in directions, but the first particle is set to be an origin in (0,0) with no velocities. We just set the mass for particles to the same value 1000 for simplification, whereas the origin should be a different particle with the large mass.

Barnes-Hut Tree

The brute force method which given above is time complex with $O(N^2)$, we can improve it by implementing a BH tree algorithm. The Barnes-Hut algorithm provides a systematic way to define "far-away" along with providing a method for approximating the force due to far-away bodies [2].

We see that BH algorithm uses clustering approximation, as it collects relatively close particles into a quadrant, and regards them as a single particle. We could construct a quadrant tree for implementing the 2-dimensional system. The subdivision is clear to visualize in two dimensions, as a quadtree.

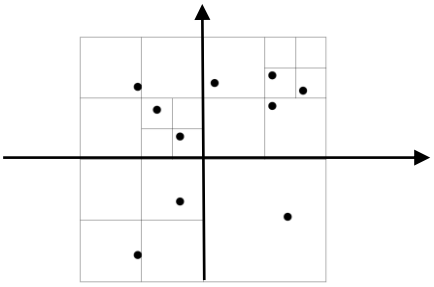

Figure 2. Quadtree construction

We will first construct the root node with the largest square, which has the range among minimum to maximum coordinates of all the particles, and then we can get which quadrant the particle should belongs to from comparing the space coordinates. When we

insert a new particle if the square already contains a particle, we need to subdivide it by inserting the leaf point into one quadrant, and then check for the inserted position, if it would be still place in the same square, we just recursively subdivide it until it is empty. We need to update center of mass after insertion, by implementing:

$\frac{1}{M}\sum_i c_i m_i$, where M is the total mass and m_i is mass for each child i. That is what we do in code:

```
void updatecenterofmass(struct node * nodep, struct body * bodyp)
//updates the center of mass after inserting a point into a branch
{
    nodep->centerx=(nodep->totalmass*nodep->centerx+bodyp->m*bodyp->x)/(nodep->totalmass + bodyp->m);
    nodep->centery=(nodep->totalmass*nodep->centery+bodyp->m*bodyp->y)/(nodep->totalmass + bodyp->m);
    nodep->totalmass += bodyp->m;
    return;
}
```

Figure 3. Update center of mass

The last thing is about summing up the force. I picked θ as 0.25, each particle is traversing from the root, so when the distance between the center of mass of the root and the particle, is greater than the length of the square's diagonal divided by θ, we will compute the force based on this node (cluster), neglecting individual bodies that are far away from the particle. Otherwise it will recursively sum up the forces deeper into the children. The choice of θ determines a compromise between accuracy and computational requirements. This could substantially reduce the computational [3].

requirements.

```
//sum the forces on body bodyp from points in tree with root node nodep
void sumforce(struct node * nodep, struct body * bodyp, double G, double theta)
{
    double dx, dy, r, rsqr; //x distance, y distance, distance, distance^2
    double accel;
    double a_over_r;

    dx = nodep->centerx - bodyp->x;
    dy = nodep->centery - bodyp->y;

    rsqr = pow(dx,2) + pow(dy,2);
    r = sqrt(rsqr);

    // r > diag/theta, where 0 < theta < 1 is an opening angle,
    if( (((r/nodep->diag) > 1/theta) || (nodep->bodyp))&&(nodep->bodyp!=bodyp) )
    {
        accel = (G * nodep->totalmass) / rsqr;    //acceleration
        a_over_r = accel/r;

        bodyp->ax += a_over_r*dx;
        bodyp->ay += a_over_r*dy;

    } else {
        //If not then the algorithm is recursively applied to the children
        //summing up the forces obtained to obtain a net force.
        if(nodep->q1) { sumforce(nodep->q1, bodyp, G, theta); }
        if(nodep->q2) { sumforce(nodep->q2, bodyp, G, theta); }
        if(nodep->q3) { sumforce(nodep->q3, bodyp, G, theta); }
        if(nodep->q4) { sumforce(nodep->q4, bodyp, G, theta); }
    }
    return;
}
```

Figure 4. Sum up the force

After improving the algorithm, we could construct the tree by setting the origin particle as the root, and insert bodies to it, updating the center of mass each time.

Implementation with evaluation of OpenMP

We can first look at the basic structure of the pseudocode:

```
1: for each (constant) timestep do
2:    for each particle i do
3:        Compute Fi(t).
4:    end for
5:    for each particle i do
6:        Update Pos_i(t) (and Vel_i(t)).
7:    end for
8: end for
```

The two inner loops are both iterating over particles. So, we could use *parallel for* on them:

```
#pragma omp parallel
{
    #pragma omp for
    for(int i = 0; i < nbodies; i++)   //sum accel
    {
        sumforce;
    }

    #pragma omp for
    for(int i = 0; i < nbodies; i++)
    {
        leapfrog update;
    }
}
```

Figure 5. OpenMP structure

we need to check for race conditions resulting from loop-carried dependencies. The thread in first loop will just update acceleration for each particle i, other things are read only or nature, so there are no race conditions. The second is more obvious that each thread only update particle i's position and velocity, with no race conditions either [4].

We did some testings on Snowy, the step number is controlled as 500, with timestep 0.1:

Thread\N	1000	2000	4000
1	14.09	35.73	94.83
4	6.36	16.38	31.64
16	4.13	8.82	15.6
32	3.67	8.66	16.16

Figure 6. OpenMP timetable

The performance is satisfying, especially when the data is large enough (N = 4000), see the performance from 4 threads to 16 of 1000 bodies is just 1.5x speedup, whereas increasing 1 to 4 causes 3 times speedup of 4000 bodies, and most speedup is nearly linear (2 times) with the increase of threads. As the

threads come to 32, the speedup is less obvious because of the overhead of threads.

Implementation with evaluation of MPI

The MPI implementation has each of p processors responsible for N/p particles. Each processor will then have to send N/p distinct units of data to each of the other p – 1 processors after each round, which is a gather/broadcast operation.

It will first broadcast the bodies information to all processes, the original first for loop in Figure 5 should be changed, and then broadcast the force (acceleration) for each particle.

```
for(int i = rank; i < nbodies; i+=numtasks)  //sum accel, split up n bodies into p processes
    {
        sumforce(rootnode, bodies+i, G, treeratio);
    }
for(int i = 0; i < nbodies; i++)
    {
        broadcastaccel(bodies+i,i%numtasks);
    }
for(int i = 0; i < nbodies; i++)
    {
        leapfrog
    }
```

Figure 7. MPI structure

We can see the results below, based on the parameters (step number = 500, timestep = 0.1):

Process\N	1000	2000	4000
1	8.08	19.69	53.17
4	4.62	10.15	25.34
16	4.07	6.88	12.96
32	9.89	7.78	13.73

Figure 6. MPI timetable

We can see the performance is pretty good, the original time for large system like 4000 bodies is long, but it got improvement after MPI implementation. When the process taken from 1 to 4, the performance got 2 times better through benefits of communications between process, however see the performance from 16 to 32 processes, the time increased a little, which might because the consumption of communication is bigger than the benefits. We need to take care of this point and choose the appropriate number of processes in practice.

There is another idea that we might just update the currently forces on rank 0, and then broadcast the position and velocity to all processes, we need to reduce the forces at the end of *sumforce* function at this point, which could be tried in the future to make a comparison.

The improvement of Hybrid OpenMP and MPI is not obvious compared to MPI.

Conclusion

As with any parallel program, there is an overhead associated with the amount of time threads spend communicating with and waiting for each other to finish, which means that parallel programs are often less efficient than serial programs. I believe the best use of parallel implementation is for large data in N-body simulation, if the data is not big enough, the brute force is better as we did BH tree decreasing the accuracy of forces, but much faster for computation.

We can say that N-body problem is often suitable to make parallelization because of its big data with complex calculation feature, and it could be a research topic to dig deeper and deeper, not only the mathematical and physics derivation, but the importance of algorithms of parallelization, we need to consider trade-off between the overhead and speedup of parallelization in practice.

We might try to improve the hybrid version with more dimensions implementation. Besides, the quadtree presented earlier does not lead to a balanced computation since some children will contain no particles. An orthogonal recursive bisection attempts to provide a more balanced division of space, with the others like Ring Termination Algorithm could also be implemented in the future [3].

References

1. GRAVITATIONAL SIMULATION
https://kof.zcu.cz/st/dis/schwarzmeier/gravitational_simulation.html

2. N-body Simulation
http://physics.princeton.edu/~fpretori/Nbody/intro.htm

3. Parallel algorithm techniques
Aaron Harwood with contributions from Tuck Leong Ngun

4. PART V - The n-body problem
https://www.cs.usask.ca/~spiteri/CMPT851/notes/nBody.pdf