HPC Report ( N – Body Problem Solution )

Introduction:

The goal of this project is to tackle the classic n-body problem, which involves the interaction of multiple bodies through gravitational forces. This problem arises in the field of astrophysics to predict the movement and gravitational interactions of planets. While the problem can be analytically solved for two bodies, it becomes unsolvable for more than two bodies. However, this problem is useful for simulating the trajectory and motion of bodies in space, as demonstrated by a 2009 study that modeled the behavior of the entire solar system over the next millennium. To solve this problem, many factors such as particle size and velocity need to be considered. However, due to its O(N2) time complexity, it is a computationally expensive algorithm. In this project, we will start by using a sequential approach to analyze its performance, followed by parallelizing it with OpenMP to improve performance. Additionally, we will evaluate its performance using OpenMP + MPI (hybrid) and CUDA.

**The Gravitational N -Body Problem**

The gravitational N-body problem deals with the forces affecting multiple bodies under the influence of gravitational force between them, where each body affects every other body. This problem is widely experimented with and associated with the creation of galaxies, black hole effects, and the search for dark matter.

We have used the All Pairs algorithm to solve this problem, although there are many more algorithms which are computationally less expensive for example Barnes Hut algorithm.

The case where there are only two bodies (N = 2) was completely and analytically solved by Bernoulli, not by Newton. Several attempts have been done to obtain analytical solutions to special combinations of initial conditions for the case N = 3 and N = 4. Mathematicians like Euler and Lagrange give several contributions there, probably you have heard or learned before about "The Lagrange Points" which is a special solution for the 3-Body problem (under special initial conditions).

In this case, we are going to solve the challenging N-Body problem numerically with a simple C++ code. Numerically means that we are computing an approximation of the solution which accuracy relies on the numerical algorithm and the floating-point precision used. Logically these parameters can be tuned to get a very acceptable solution to the problem.

All Pair Algorithm:

It is an exhaustive brute force algorithm which takes a particle (i) and calculate the force on it due to all other particles in the system. The gravitational force is given by Newton's law of gravitation, which states that the force between two bodies is proportional to their masses and inversely proportional to the square of the distance between them.

Any two bodies in the system are attracted to each other with a force Fij which is inversely proportional to the square of distance between them.

F = Gmimj/rij2

Since there is a force, the body also experience a net acceleration on it due to all the forces acting on it by the remaing N-1 bodies, which can be given by

Fi  = miai

We simply used these equation to solve this problem, we used one for loop to select each particle, we select particle (i) and then in the inner loop, we calculate its force with all other bodies in the system.

The total force on particle i is thus given by

A picture containing schematic

Description automatically generated

**Function calculate force() is**

**foreach i: body do**

**find force(i, particles)**

Function find force(i: body, particles)

is foreach j in particles do

if j̸=i then  
d sq = distance(i, j)  
force[i].x += d x \* mass(i) / d sqˆ3

force[i].y += d y \* mass(i) / d sqˆ3

The Particle-Particle method is the simplest. The method is to:

Accumulate forces by finding the force *F(i,j)* of particle *j* on particle *i,*

Integrate the equations of motion, and

Update time counter

For example, in a gravitational N-body simulation, a particle of mass *M* attracts another particle of mass *m* with a force: *-(GMm/r^3)\*r*. You have N particles, computing the force (N-1) times. Separate the equation into two first-order differential equations involving acceleration and involving velocity. Then, use an integration scheme like Euler or Runge-Kutta to get the positions and velocities. (Simple, eh?)

As particles approach each other, the forces get much bigger, and so do the velocities. If a constant time-step is used, the particles will travel too far and the errors can get unacceptably large when two particles come close together. To avoid this, you will want to consider a numerical integration scheme that uses variable time-steps. These schemes automatically cut the time step down when the particles are near each other, and increase the time step when the particles are far away.

The Particle-Particle direct integration approach is flexible but has a high computational cost: O(N^2) operations are required to evaluate the forces on all N particles.

If you have less than about N=1000 particles, and are interested in the close-range dynamics of the particles, then this method is the most straight-forward.

In the for loop, we calculate the velocity by integrating it. Here we are simply integrating this for each velocity component, vel[0] is Velocity along x axis

**mParticles[pi].vel[0] += force[0] \* mInverseMass \* mDt;**

dv / dt = a

a = F / m => 1/m = **mInverseMass**

**dv = F \* mInverseMass \* dt**

A picture containing text

Description automatically generated

**Sequential Code with basic optimisation :**

We initialised the problem with all the particles at a random position in a circle with random velocities, so as the simulation starts and time passes by we can see that the particles first come close to each other, due to the gravitational force between them. As they all speed up and collide and then move apart from each other and decelerate due to acceleration due to gravitational attraction. For simplicity, we keep mass for all the bodies same. This way, we don’t bring another variable into the problem.

This is perfectly a simulation of how it should happen in reality.

Results:

Picture of our system at different time steps.

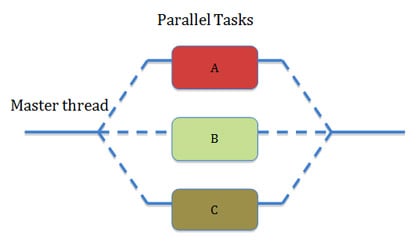
Time taken for different number of bodies in system:

**OpenMP –**

**What is openMP.**

OpenMP (Open Multi-Processing) is an application programming inter- face that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran. The main feature is that allows you to use many threads in parallel, in a single process, for increasing the execution performance of a program, exploting the single program multiple data parallelism (SPMD).

OpenMP is a programming model which allows us to use multiple threads to run a program, a programming model is not a programming language which means that it provides us methods to implement to our existing code in the form of pragma and decorators.



OpenMP consists of a set of:

compiler directives, responsible of defining which part of the code will run in parallel.

Library routines, a set of user level routines like, for instance, the one used for retrieving the number of threads in a parallel region.

Environmental variables, for adjusting the runtime behavior of parallel aplica- tion.

OpenMP is a shared memory architecture, meaning that all the multiple threads, that we decide to use, can access to the same whole main memory. A single process, in fact, has its own memory, file etc. . . , and it can create a certain number of threads that share some process state information, like the memory. For this reason, threads are considered very lightweight tasks, since they can reduce memory overhead. This API is also very diffused in parallel programming because it is available with almost all the compilers, like GCC or ICC, and also is always updated for dealing with new programming methods and technologies.

**Parallelisation with openMP, how we parallelise it.**

First we identify the hotspots of our code, namely the region which is using the most computational resources. As it is clear from above, the for loop in the integrate method uses the most resources as it is an embedded loop. And for each iteration, It computes force on it.

So if we have 1000 particles and we run on 4 threads, then each thread will have 250 particles and each of them is going to compute the solution for a subgroup of the total number of particles.

Text

Description automatically generated

Results with OpenMP

When we tried with only 100 bodies, we saw that it took x time, but when we increased the number of threads, there was a 2x speed up. This was good, but after we increased the threads to 4 and 8 from 2, we did not see a significant speed up as the overhead of launching those threads was higher.

So, to check if paralellisation was working correctly, we increased the number of particle in the system to 10,000 and tried to study the speed up. It was scaling up perfectly, as the time changed from 126 seconds to 44 seconds.

1 Threads

2 Threads

4 Threads

8 Threads

16 Threads

32 Threads

64 Threads

128 Threads

256 Threads

Write a little note about these results here, seeing the result with different number of particles.