N-Body Problem

- Explanation
- Approach
- Parallelisation using MPI
- Algorithm-Simulation
- Conclusion

18-10-20
CED17I047
Ritvik Raj Singh

Problem Formulation:

The N-body problem is one of famous problems in classical physics for predicting the motion of n celestial body that interacts gravitationally in free space.It is a problem for predicting individual motion of bodies starting from a quasi state.

The problem has been motivation to understand motions of sun ,planets and other celestial bodies in global clusters . Consider general relativity the problem is difficult to solve and still is an open problem .See two-body problem and restricted three-body problem which have been solved .

The below problem solution is based on problem of simulation of random 5040 masses ranging from 34000,25000000 kg on **2d** plane on (-1000,1000) on both x and y coordinates where initial states of bodies are in quasi state (initial velocity and acceleration are 0 in both x axis and y axis) and initial position are ((-500,500)|(-400,600)) in x and y axis respectively .

Assumption:

For simulation purpose and ease of calculation classical newton laws are used for computing velocity and acceleration of individual bodies .

$$fi,j(t) = -Gmimj||ri(t)-rj(t)||3(ri(t)-rj(t))$$

Where i and j the body are applying f(i,j) force on each other. (Newton 3rd law) where G =6.67 × 10^{-11} Newtons kg^{-2} m^2

NOTE: While calculating position and velocity of actual planetary motion Newton Laws are no longer valid.(https://en.wikipedia.org/wiki/N-body_problem)

Approach

Assuming Classical Newton laws to hold true following formulas are used for calculating for velocity and acceleration .

$$v = u+at \qquad s = ut+(0.5)a(t)*t$$

u	:	initial velocity
v	:	final velocity
a	:	acceleration
t	:	Time
s	:	Distance

Assuming Classical Newton laws to hold true following formulas are used for calculating for velocity and acceleration.
The simulation is based on updating the vector and velocity information at delta t timestep instead of continuous time simulation(deltaT≈0) .To make the simulation smooth lower the deltaT value.

The input for the given problem is given by coordinates of mass(ranging in 3400-2500000kg) given in file text.
The algorithm steps are:
For each time step

Compute force computation on ith body by all n-1 body :

Compute Vector position of j th body wrt to ith body in both x and y axis
Computer rvector = (sqrt(dx*dx+dy*dy))^(1.5)

$F = (Gm_im_j * vec) /(rvector)$
Compute acceleration on each i,j body and thus compute i,j velocity value of both i and j thus new r vector position of i with new velocity.

Once the value of force is computed for each body then update the new position of each body simultaneously .

Repeat until time finished

Parametres:

Time step incremented by deltaT
NOTE:Varying deltaT and making it small(1-5) will make simulation smooth but computation time taken will be extremely large for number of bodies

Number_of_particles: Increasing number of bodies (maximum 111002) will increase computation but make simulation uniform .

Total_Time_Run : The number of timestep until which the simulation will run .
NOTE:Increasing the Total_Time_Run will increase the runtime of simulation.

counter_velocities :A single dimension array of length 2n containing Velocity-x and Velocity-y of each n body at contiguous memory location .

OpenGL is used for simulating the bodies (look into simulation.cpp)
NOTE: The simulation is attempted only when all the computation is done for all specific bodies.It is not done simultaneously when calculating individual velocity and position of bodies at each timestamp.

DEBUG_MODE : variable to set 1 for light information display and 2 for all information displayed at each time interval .

Parallelisation with MPI:

Although using similar methodology used in openmp parallelisation for the same problem might work but will incur much communication overhead,thus block partitioning methodology is used .

Block partitioning methodology is used for MPI thus utilising less memory and improving load balancing. Cache misses are still prominent but load balance improvement makes up for it.

Strategy :

Parallelisation of computing of force is done by distributing the array of mass vector to p nodes reducing the workload to n/p .

Initial vector position and mass of n bodies information is sent to each worker (because computation of i th requires all rest n-1 bodies force).

The velocities of the bodies are distributed among p workers each given their velocity for usage of updation of individual rvector and velocity value.

To avoid race condition rvector ,velocity ,acceleration(force) is stored in individual array data type and updation is done once force is computed for rest n-1 bodies .

```
MPI_Bcast(masses, NUMBER_OF_PARTICLES, MPI_DOUBLE,
MASTER_PROCESSOR_RANK, MPI_COMM_WORLD);//send same information

MPI_Bcast(positions, 2 * num_of_processors * particles_per_processor,
MPI_DOUBLE, MASTER_PROCESSOR_RANK, MPI_COMM_WORLD);
```

Note velocities are scatter (equally distributed)

```
MPI_Scatter(velocities, 2 * particles_per_processor, MPI_DOUBLE,
curr_proc_velocities, 2 * particles_per_processor, MPI_DOUBLE,
MASTER_PROCESSOR_RANK, MPI_COMM_WORLD);
```

For each worker force is calculated and once force(acceleration)is known those specific bodies velocities are updated.

Specific distributed velocities(current proc_velcoties) are collected again MPI_GATHER.

Then all arrays of each quantities are freed.

```
//READ DATA

Initialize MPI and call n workers
//Share rvector and masses to all workers .
//Distribute velocity

Parallelise p workers
        For each time step:
                For each body data of worker :
                        Compute force from rest bodies under that specific
        worker and update it

                For each body in worker:
                        Compute and update Velocity
                        Compute and update rvector

                //Wait until all positions are updated
                //MPI_Allgather

        Gather velocities of each worker after the individual worker is
        done with the task .


        //Free Mass
        //Free(Velocity)
        //Free(Curr_proc_velocity)
        //Free(Forces)

End MPI_routine
```

Speed up improvement with MPI:

- For n = 5040 bodies simulation over 6000s total time sampled at 5 second on 20 core system equivalent 7 nodes are the following .

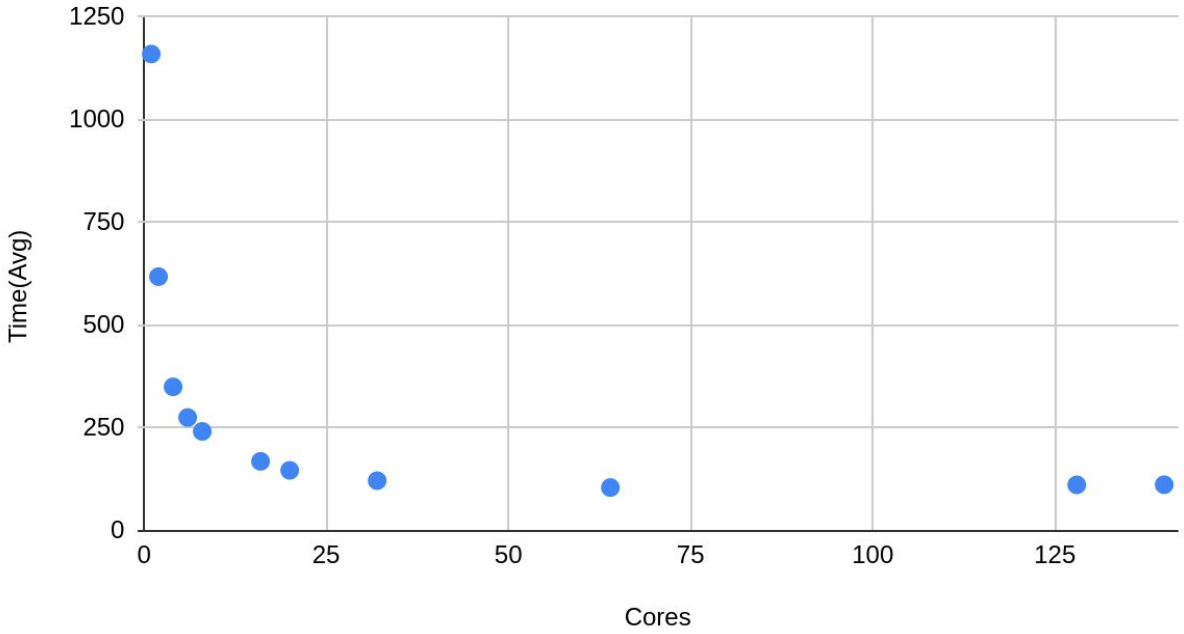
Processor Info (10 cores system supporting hyperthreading each with min clock boost 1.2 GHZ max 3.0 GHZ) Intel® Xeon® Processor E5-2640 v4

Cores	Time1	Time2	Min1
1	1162.656	1160.188	1160.188
2	618.5245	621.1817	618.5245
4	358.6218	350.4052	350.4052
6	278.305	275.813	275.813
8	242.584	242.067	242.067
16	169.0924	172.8312	169.0924
20	147.308	148.2056	147.308
32	122.1699	124.01	122.1699
64	107.77	105.435	105.435
128	111.9598	114.9491	111.9598
140	112.3102	125.303	112.3102

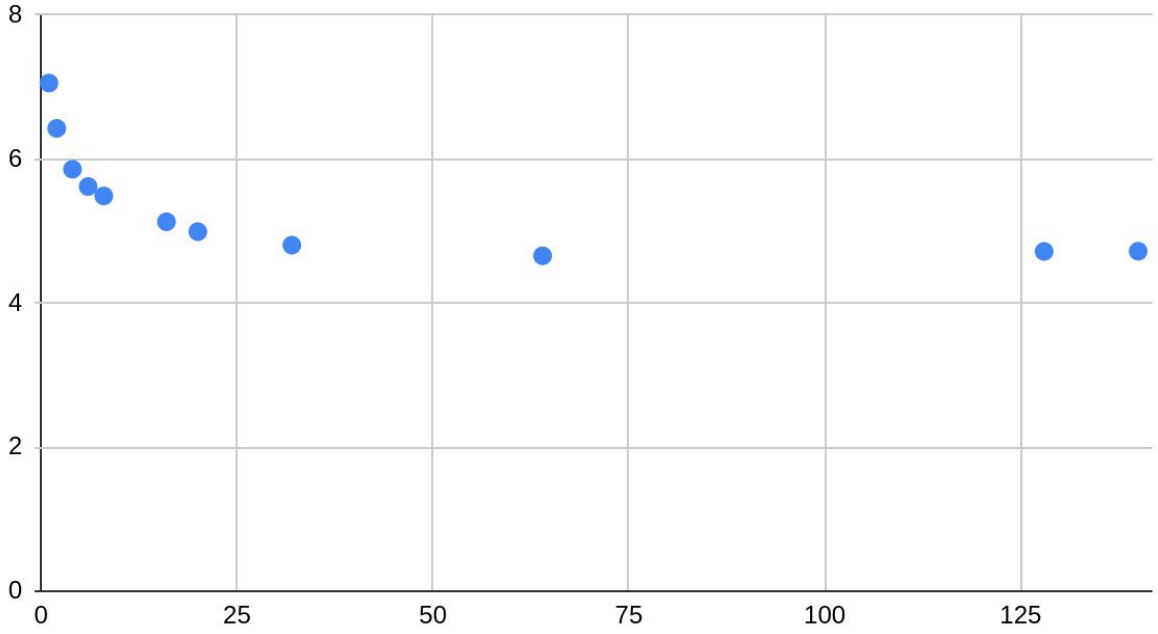
F = p/(1-p)*((T(p)-T(1))/T(1))

AVERAGE Parallelise factor = 91.811 %

Time(Min) vs. Cores



Log(Time) vs Cores



Algorithm Simulation:

Check video at : https://drive.google.com/file/d/1zpL6n2CtIua31vD5PkKOP-b5d_KRnyjT/view?usp=sharing

Algorithm is simulated using freeglutdev opengl in ubuntu

Conclusion:

On average the parallel section is 91 % of code .

Race Condition is avoided by ensuring different array variables to different workers.

Communication overhead is minimal until the number of workers are less than 64 ,after that communication overhead takes more time than reduced time by multiple workers.

Increasing the number of bodies decreases communication overhead time as compared to individual worker time.

The number of bodies are assumed to be equal divisible by n workers.