

N-Body Problem

- Explanation
- Approach
- Parallelisation
- Algorithm-Simulation
- Conclusion

10-10-20
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 10000 masses ranging from 34000,250000000 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 .

$$f_{i,j}(t) = -G m_i m_j / |r_i(t) - r_j(t)|^3 (r_i(t) - r_j(t))$$

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

NOTE: While calculating position and velocity of actual planetary motion Newton Laws are no longer valid ,due to fixation of barycenter . (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 .

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

| | | |
|---|---|------------------|
| 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($\Delta t \approx 0$) .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 = $(\sqrt{dx^2 + dy^2})^{1.5}$

$F = (Gm_i m_j * \text{vec}) / (r\text{vector})$

 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 ΔT

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

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

Total Time : The number of timestep until which the simulation will run .

NOTE: Increasing the total time will increase the runtime of simulation.

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

Parallelisation with OpenMP:

The algorithm uses block distribution as symmetry of Force , $F(i,j) = F(j,i)$ and cyclic distribution for force calculations ,since all bodies force value computation is calculated and then updated at end simultaneously .

Since OpenMP uses shared memory for parallelisation thus cyclic distribution computation parallelisation is easy.

Thus `#pragma omp parallel` for directive is added for force computation from all n-1 bodies and workload is distributed on static scheduling where chunks of data are equally distributed to the number of threads that are available .

To avoid race conditions each data is saved in individual rvector position ,velocity data in vector data type in c++ so that each thread in parallel constructs ensures access of one element at index .

Parallelization is done on force of n-1 body computation is independent of each other thus no memory race condition is introduced .

$$F_i = \sum_{j=1:n, j \neq i} F_j$$

For each particle i do:

 foreach particle j > i do

 ompsetlock(& locks[i])

$F_i(t) += f_{i,j}(t)$.

 ompunlock(& locks[i])

 ompsetlock(& locks[j])

$F_j(t) -= f_{i,j}(t)$.

 ompunlock(& locks[j])

 end for

end for

Speed up improvement with OpenMP:

- For n = 2000 bodies simulation over 10000s total time sampled at 5 second on 4 core system taken are the following .

| Threads | 1 | 2 | 4 |
|------------------------|---------|-------------|--------------|
| Time(s) of execution | 249.406 | 155.456 | 114.925 |
| Parallelisation Factor | | 0.753390054 | 0.7189402019 |

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

Average parallelization factor=0.73616512

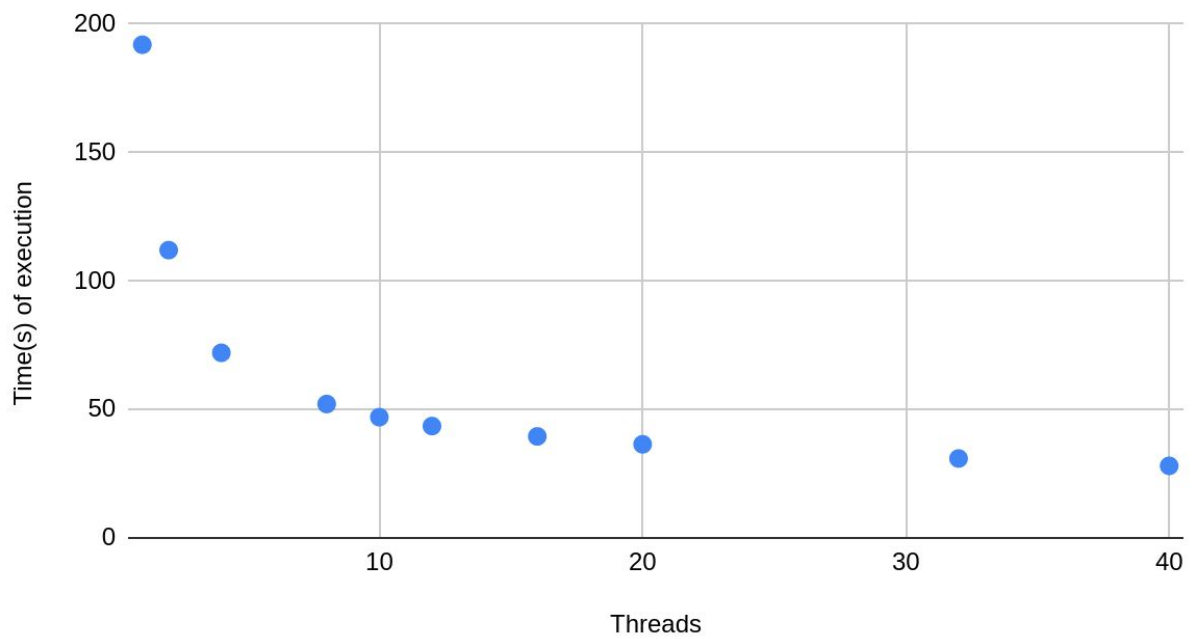
- For n = 2000 bodies simulation over 2000s total time sampled at 5 second on 20 core system(max 3.00 GHZ time) taken are the following.

| Threads | Time(s) of execution | Parallelisation Factor |
|---------|----------------------|------------------------|
| 1 | 191.981 | |
| 2 | 112.019 | .82650996 |
| 4 | 72.0463 | .83296622 |
| 8 | 52.1042 | .83268240 |
| 10 | 46.9954 | .83912007 |
| 12 | 43.5288 | .84356188 |
| 16 | 39.5014 | .84719275 |
| 20 | 36.4251 | .85291280 |
| 32 | 30.9002 | .86611115 |
| 40 | 28.041 | .87583453 |

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

Average parallelization factor=0.8463213

Time(s) of execution vs. Threads



Algorithm Simulation:

```
111002
0 -1 1200000
1 -6 1600000
6 -2 1700000
7 -7 1800000
6 -4 1900000
382.41019308699985 -10.56777060396454 7431584.252410273
391.4873451132962 560.8353263119628 20132376.212476417
324.59579912220994 -340.4438257172027 21228118.452569764
300.3791981454491 -293.35101918846783 18915094.61615728
207.018105583602 580.9589515739616 17988662.53734793
175.35152608876666 312.70689808355655 3741988.0789586133
55.704258072509854 483.94239320741924 2150446.1472623143
460.9793175065407 5.346229148224046 10089418.467151206
206.98657842363585 -106.16641826312227 20793440.11559907
39.31487532182681 181.61142469070973 13529974.048792219
454.3376088074054 104.54351850483975 21440621.189947463
16.1402215594304 350.54901511870764 3903797.4215961327
102.99966126558212 -388.02874249388896 296934.52297034184
-433.72278063092017 -377.05122299050026 7421164.638680999
113.81726721955631 100.04633170176233 7239273.3621350685
-263.37070430690403 -60.893275256027756 17785103.411936723
75.3393127760362 -25.643788215238125 17202008.5380804
-99.62998314628221 -307.984240481764 24591271.74603319
127.89345584623754 -82.48035948517293 2775990.2598822173
266.2907018513459 -250.00687171748908 1191988.594116244
402.9860101497312 -65.02136815957326 5048761.5320174215
96.69042187003214 -340.5458758092243 859090.0540372249
204.48483257114793 -142.0467962976537 7021716.824417311
-58.43115237362007 -175.94661818456913 12067289.838852357
-267.79748458999285 461.6085907096535 12542358.004313797
-196.6149501502788 -305.73668748231097 15149288.429161357
257.22355020122785 -344.88908537948786 12160782.392786667
-347.91494610418926 -289.2357018422734 19346359.633237004
-441.27729280129903 -10.038370340767872 5706153.678219029
485.4270115218625 152.08916167545493 14479888.090108824
-218.96311432920692 521.8070161153717 12616360.096825771
59.42926615842312 325.0467710591243 10230981.627043758
-280.91379017564143 428.32235778313907 1915192.0146725047
280.85415411465675 -24.788371842444104 4374162.333202648
173.5494849375524 120.8329790850131 6978154.214412363
-153.692667525018 354.5377920158636 19075251.051006247
23.54891447250347 205.17831674150267 17277748.23884927
131.07771249810583 -34.399227303400195 18270757.36179026
-162.307257307311 -55.23944521558585 22020709.01442318
-77.24448517697564 486.4063327743317 9834109.937019092
1 // g++ Topnump_vml.cpp -o run1 -ld
2 // GMP_NUM_THREADS=4 ./run1
3
4 #include <omp.h>
5 #include <stdlib.h>
6 #include <stdio.h>
7 #include <math.h>
8 #include <iostream>
9 typedef long long int lld;
10 #define TotalTimeStep 10000
11 // #define TotalTimeStep 20
12
13 using namespace std;
14
15
16 // #define Gconstant 0.6674 * pow(10,-11)
17 double Gconstant = 0.6674 * pow(10,-11);
18
19
20
21 int main(int argc, char const *argv[])
22 {
23     lld n;
24     int temp;
25     // cin>>n;
26     // double *posx,*posy,*mass; //in kg
27     // double *vex,*vey; //in m/s
28
29     double posx[120000];
30     double posy[120000];
31     double mass[120000];
32     double vex[120000];
33     double vey[120000];
34
35
36     double m1,newposx1,newposy1,m2,r1vector,r2vector,rddiffector;
37     double Forcex1,Forcey2;
38     double accelerationx,accelerationy;
39     double tempvx1,tempvy2;
40     double deltat = 5;
41     double taotime1 = 0;
42     // posx = new double[n];
43     // posy = new double[n];
44     // mass = new double[n];
45     // vex = new double[n];
46 }
```

Check the video at

<https://drive.google.com/file/d/1LTMBHbnq3ceiyO7IR0vORSjGo6JRp3tB/view?usp=sharing>

Run the source code .For simulation run opengl freegludev is required in ubuntu .

Conclusion:

The distributed workload decreases the time taken by almost 45 % workload .

On average the parallel section is 75 % of code .

Even though shared memory is used no memory race condition is encountered .

Equally proc load is distributed by n threads specified during runtime .