N-Body Problem(OpenMP)

- 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 34000,25000000 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NOTE: While calculating position and velocity of actual planetary motion Newton Laws are no longer valid ,due to fixation of barycenter.(https://en.wikipedia.org/wiki/N-body_problem)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 .

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

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 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 .
Fi = ΣFj (j=1:n j!=i)

For each particle i do:
    foreach particlej > i do
        ompsetlock(& locks[i])
        Fi(t)+=fi,j(t).
        ompunsetlock(& locks[i])
        ompsetlock(& locks[j])
        Fj(t)−=fi,j(t).
        ompunsetlock(& 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))

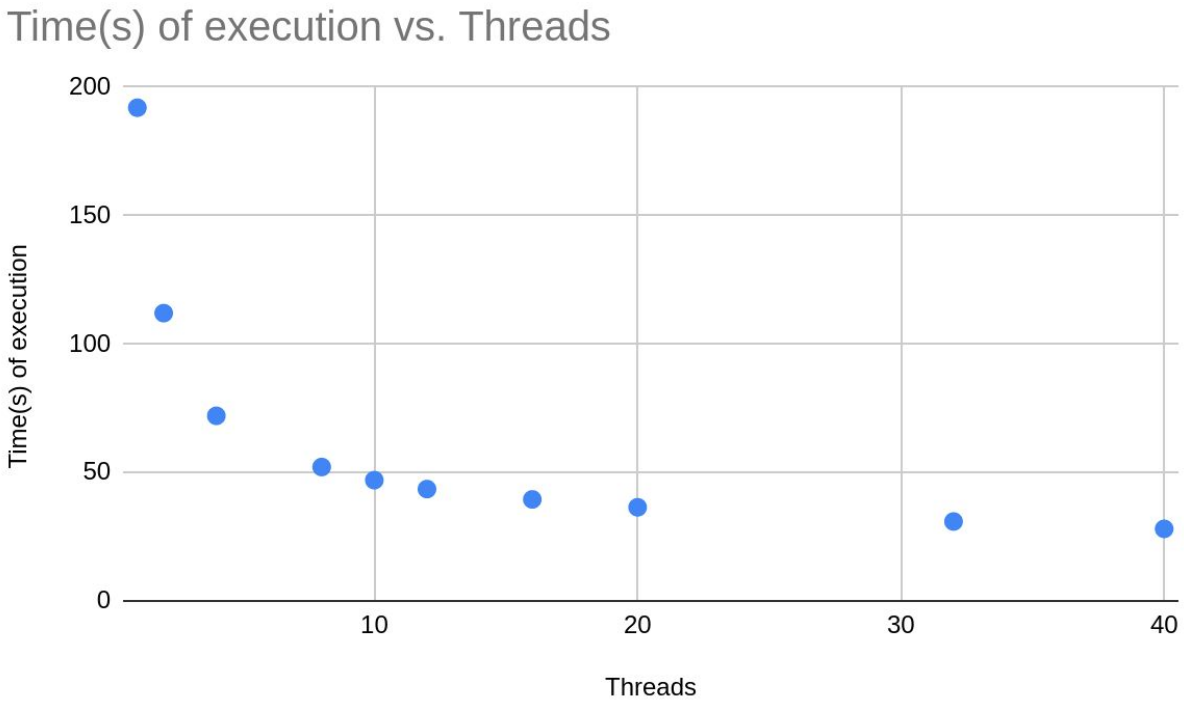
Average parallelization factor=0.73616512

- 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.

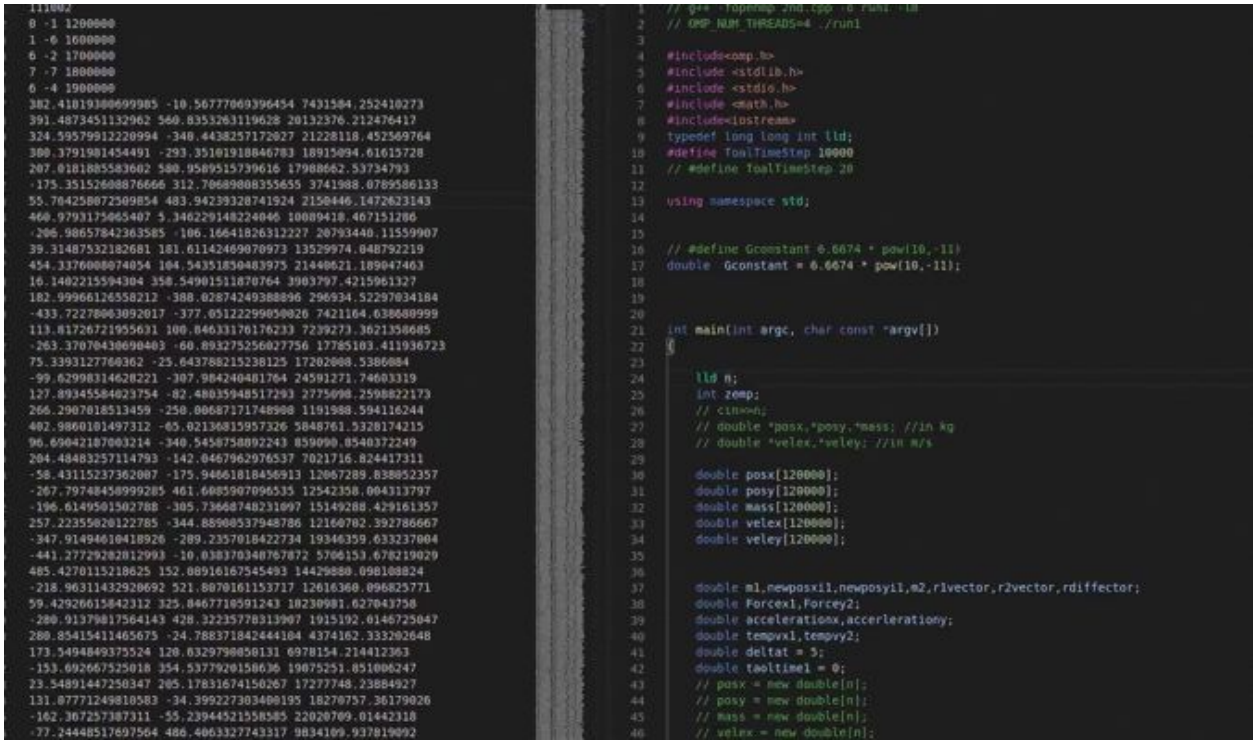
Threads	Time(s) of execution	Parallelisation Factor
1	191.981	
2	112.019	.82650996
4	72.0463	.83296622
8	52.1042	.83268240
10	46.9954	.83912007
12	43.5288	.84356188
16	39.5014	.84719275
20	36.4251	.85291280
32	30.9002	.86611115
40	28.041	.87583453

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

Average parallelization factor=0.8463213


Time(s) of execution vs. Threads
200
150
100
50
0
Time(s) of execution
10
20
30
40
Threads

Algorithm Simulation:



```
111002
0 -1 1200000
1 -6 1600000
6 -2 1700000
7 -7 1800000
6 -4 1900000
382.41019300699985 -10.567770693964545 7431584.252410273
391.4873451132962 560.8353263119628 20132376.212476417
324.59579912220994 -340.4438257172027 21228118.452569764
380.3791981454491 -293.35101918846783 18915094.61615728
207.0181885583602 580.9589515739616 17988662.53734793
-175.35152608876666 312.70689808355655 3741988.0789586133
55.704258072509854 483.94239328741924 2150446.1472623143
460.9793175065407 5.346229148224046 10089418.467151286
-206.98657842363585 -106.16641826312227 20793440.11559907
39.314875321826 81.61142469070973 13529974.048792219
454.3376008074054 104.54351850483975 21440621.189047463
16.1402215594304 358.54901511870764 3903797.4215961327
182.99966126558212 -388.02874249388896 296934.52297034184
-433.72278003092017 -377.05122259050026 7421164.638680999
113.81726721955631 100.84633176176233 7239273.3621350685
-263.37070430690403 -60.893275256027756 17785103.411936723
75.33931277602 -25.643788215238125 17202008.5386084
-99.62998314628221 -307.9842404817 24591271.74603319
127.89345584023754 -82.48035948517293 2775098.2598822173
266.2907018513459 -258.00687171748998 1191988.594116244
402.9860101497312 -65.02136815957326 5848761.5328174215
96.69042187003214 -340.5458758892243 859090.8540372249
204.48483257114793 -142.04679629765 37 7021716.824417311
-58.43115237362087 -175.94661818456913 12067289.838052357
-267.79748458999285 461.6085907096535 12542358.004313797
-196.61495015027888 -305.7366874823109 7 15149288.429161357
257.22355820122785 -344.88900537948786 12160702.392786667
-347.91494610418926 -289.2357018422734 19346359.633237004
-441.27729282812993 -10.038370340767872 5706153.678219029
485.42701152180625 152.08916167545493 14429880.098108824
-218.96311432920692 521.8070161153717 12616360.096825771
59.42926615842312 325.8467710591243 10230981.627043758
-280.91379817564143 428.32235778313907 1915192.0146725047
280.85415411465675 -24.788371842444104 4374162.333202648
173.54948493775524 120.6329790850131 6978154.214412363
-153.6926675250 18 354.5377920158636 19075251.851006247
23.54891447250347 205.17831674150267 17277748.23884927
131.07771249810583 -34.399227303400195 18270757.36179026
-102.367257387311 -55.23944521558585 22020709.01442318
-77.24448517697564 486.4063327743317 9834109.937819092
```

```
// g++ -fopenmp 2nd.cpp -o run1 -lm
// OMP_NUM_THREADS=4 ./run1

#include<omp.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include<iostream>
typedef long long int lld;
#define TotalTimeStep 10000
// #define TotalTimeStep 20

using namespace std;


// #define Gconstant 6.6674 * pow(10,-11)
double  Gconstant = 6.6674 * pow(10,-11);



int main(int argc, char const *argv[])
{

    lld n;
    int zemp;
    // cin>>n;
    // double *posx,*posy,*mass; //in kg
    // double *velex,*veley; //in m/s

    double posx[120000];
    double posy[120000];
    double mass[120000];
    double velex[120000];
    double veley[120000];


    double m1,newposxi1,newposyi1,m2,r1vector,r2vector,rdiffector;
    double Forcex1,Forcey2;
    double accelerationx,accerleration y;
    double tempvx1,tempvy2;
    double deltat = 5;
    double taoltime1 = 0;
    // posx = new double[n];
    // posy = new double[n];
    // mass = new double[n];
    // velex = new double[n];
```

Check the video at
https://drive.google.com/file/d/1LtMBHbnq3cejyO7lR0vORSjGo6JRp3tB/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 .

N-Body Problem(MPI)

- 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 .

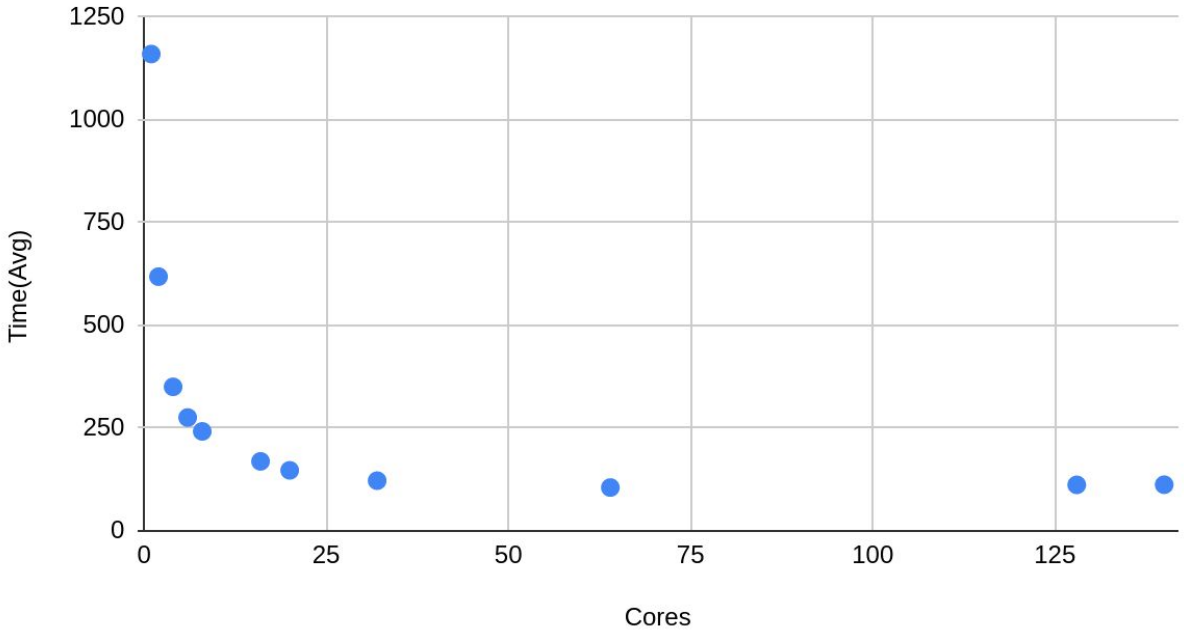
Processor Info (10 cores system supporting hyperthreading each with min clock boost 1.2 GHZ max 3.0 GHZ) Intel® Xeon® Processor E5-2640 v4

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

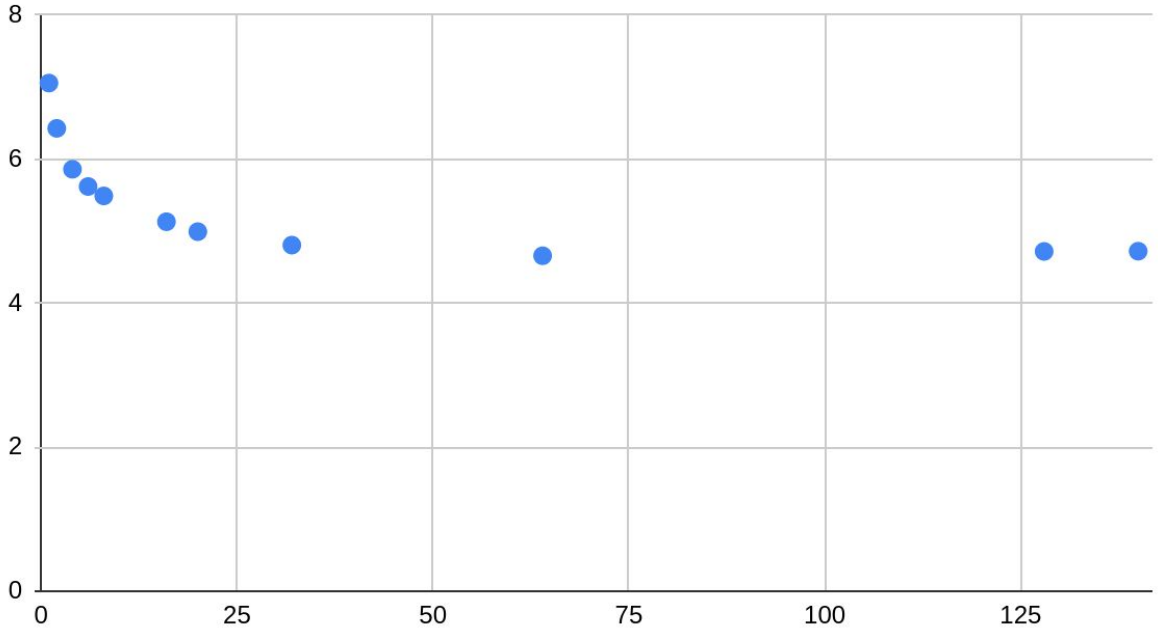
F = p/(1-p)*((T(p)-T(1))/T(1))

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.

N-Body Problem(CUDA)

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

4-12-20
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 CUDA:

Parallelisation is achieved by making each body force computation parallel by calling kernel

Although calling kernel multiple times causes overhead ,dividing all bodies force computation requires less kernel call but overloads memory limitations of kernel .

Initial vector position and mass of n bodies information is sent to the kernel (because computation of i th requires all rest n-1 bodies force).
To avoid race condition rvector ,velocity ,acceleration(force) is stored in individual array data type and updation is done once force is computed from rest n-1 bodies .

// each body computation of force is parallelised .
//copy position vector , velocity vector, mass vector to kernel
Call kernel
//copy force vector back
//update new velocity and position of the ith body

Strategy:

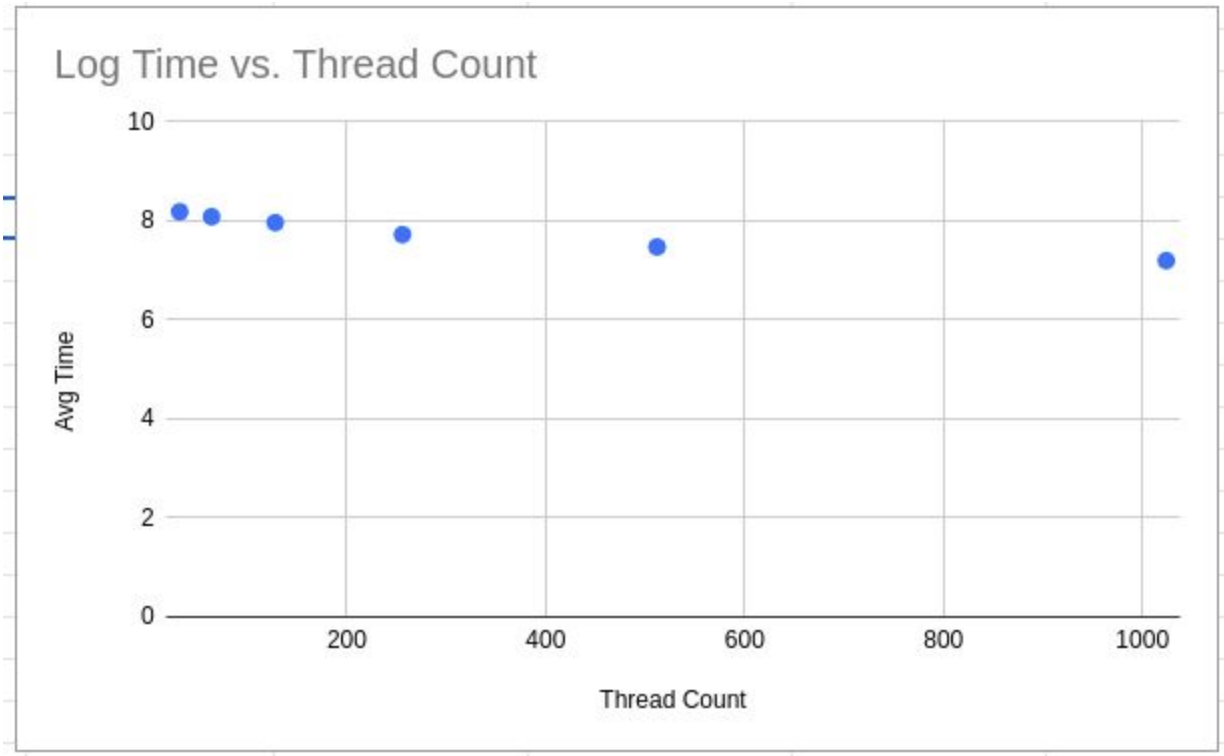
```
cudaMemcpy(dposx,posx,array_bytes,cudaMemcpyHostToDevice);
cudaMemcpy(dposy,posy,array_bytes,cudaMemcpyHostToDevice);
cudaMemcpy(dmass,mass,array_bytes,cudaMemcpyHostToDevice);
cudaMemcpy(dvelex,velex,array_bytes,cudaMemcpyHostToDevice);
cudaMemcpy(dveley,veley,array_bytes,cudaMemcpyHostToDevice);
kernelcomputenewvel<<<1,threads_in_block>>>(dmass,dvelex,dveley,dposx,dposy,forece
arrayx,forecearrayy,i1,threads_in_block,nobj);
// __global__ void kernelcomputenewvel(double idx ,double masslist[], double
velocitylistx[],double velocitylisty[], double postionx[],double postiony[],lld threadcunt,double
Forcex1[],double Forcey2[] )
cudaMemcpy(hoforecearrayx,forecearrayx,threadsize,cudaMemcpyDeviceToHost);
```

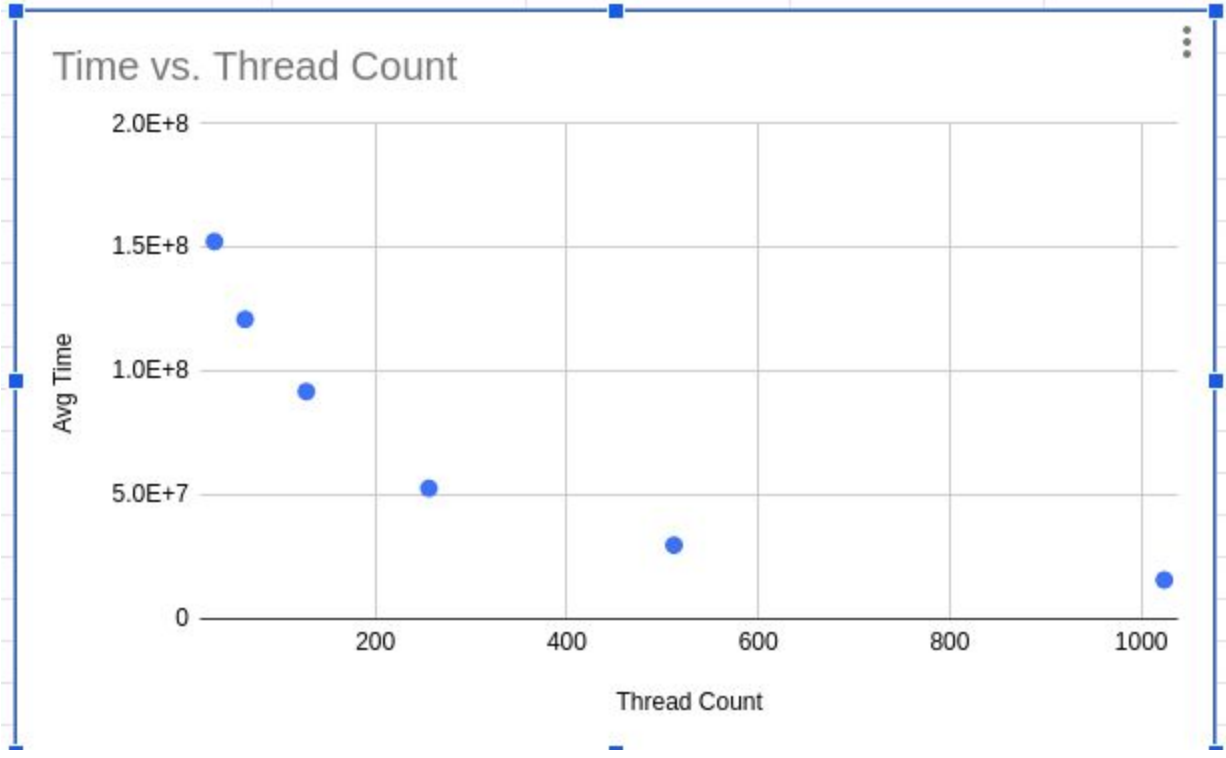
Speed up improvement with CUDA:

GPU usage (Tesla P100-PCIE-16GB,memory available 16 G.B.(distributed on server))
//devicequery output

Device 0: "Tesla P100-PCIE-16GB"
CUDA Driver Version / Runtime Version 10.1 / 10.1
CUDA Capability Major/Minor version number: 6.0
Total amount of global memory: 16281 MBytes (17071734784 bytes)
(56) Multiprocessors, (64) CUDA Cores/MP: 3584 CUDA Cores
GPU Max Clock rate: 1329 MHz (1.33 GHz)
Memory Clock rate: 715 Mhz
Memory Bus Width: 4096-bit
L2 Cache Size: 4194304 bytes
Maximum Texture Dimension Size (x,y,z) 1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
Maximum Layered 1D Texture Size, (num) layers 1D=(32768), 2048 layers
Maximum Layered 2D Texture Size, (num) layers 2D=(32768, 32768), 2048 layers
Total amount of constant memory: 65536 bytes
Total amount of shared memory per block: 49152 bytes
Total number of registers available per block: 65536
Warp size: 32
Maximum number of threads per multiprocessor: 2048
Maximum number of threads per block: 1024
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)

Threads	Time(s) of execution
32	152456424
64	121030088
128	91820974
256	52735786
512	29772088
1024	15715570


Log Time vs. Thread Count
10
8
6
4
2
0
Avg Time
200
400
600
800
1000
Thread Count


Time vs. Thread Count
2.0E+8
1.5E+8
1.0E+8
5.0E+7
0
Avg Time
200
400
600
800
1000
Thread Count

Conclusion

The device host methodology clearly benefits from large thread calls. Large calls of thread(per block)depends on the GPU architecture model and memory .

Usage of system bus ensures fast memory transfer and allows multiple GPU usage(if available).
The above methodology underutilised the given hardware by not utilising blocks and multidimensional gtids in streaming multiprocessor .

//CODE

MPI

```
// mpic++ alphav2.cpp -o run1
// mpirun -np 4 ./run1

#include <mpi.h>
#include<iostream>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define DEBUG_MODE 0
#define MASTER_PROCESSOR_RANK 0
#define DELTA_T 10
#define INPUT_FILE_DATA "inputfile.txt"




using namespace std;


double GRAVITATIONAL_CONSTANT = 6.6674 * pow(10,-11);
// FILE *fp4;



int rank_of_processor,num_of_processors;

MPI_Datatype aggregat_Type;
```

```
double * velocities = NULL;
long long int NUMBER_OF_PARTICLES = 2000;
long long int TOTAL_TIME_RUN = 3000;


// Function prototypes for calculating forces, updating position &
velocity
void compute_force(double masses[], double positions[], double
forces_each_proc[], int rank_of_processor, int BODIES_per_proc);
void update_positions_velocities(double positions[], double
forces_each_proc[], double velocities_per_proc[], int rank_of_processor,
int BODIES_per_proc);

// Main function.
int main(int argc, char * argv[]) {

 // Array for all positions and velocities & forces for particles
belonging to current processors.
 double * masses;
 double * positions;
 double * velocities_per_proc;
 double * forces_each_proc;

 MPI_Init( & argc, & argv);
 MPI_Comm_size(MPI_COMM_WORLD, & num_of_processors);
 MPI_Comm_rank(MPI_COMM_WORLD, & rank_of_processor);

 // Get number of particles per processor (fancy calculation is to get the
ceiling).
 int BODIES_per_proc = (NUMBER_OF_PARTICLES + num_of_processors - 1) /
num_of_processors;

 if (rank_of_processor == MASTER_PROCESSOR_RANK) {
   velocities = (double *)malloc(2 * num_of_processors * BODIES_per_proc *
sizeof(double));
 }

 masses = (double *)malloc(NUMBER_OF_PARTICLES * sizeof(double));
```

```
 positions = (double *)calloc(2 * num_of_processors * BODIES_per_proc,
sizeof(double));
 velocities_per_proc = (double *)malloc(2 * BODIES_per_proc *
sizeof(double));
 forces_each_proc = (double *)malloc(2 * BODIES_per_proc *
sizeof(double));



 // This is to communicate positions and velocity of each chunk (so 4
total doubles for each particle in chunk).
 MPI_Type_contiguous(2 * BODIES_per_proc, MPI_DOUBLE, & aggregat_Type);
 MPI_Type_commit( & aggregat_Type);

 if (rank_of_processor == MASTER_PROCESSOR_RANK) {
   FILE * fp = fopen(INPUT_FILE_DATA, "r");
   if (!fp) {
     printf("Error opening input file.\n");
     exit(1);
   }
   long long int number_offile_list;
   int counter_masses = 0;
   int counter_positions = 0;
   int counter_velocities = 0;
   int line = 0;
     fscanf(fp, "%lld", &number_offile_list);
     cout<<"MAXIMUM NUMBER of objects"<<number_offile_list<<endl;
       FILE *fp4 = fopen("justcoordinates.txt","w+");

       fprintf(fp4,"%lld",NUMBER_OF_PARTICLES);

       fprintf(fp4,"\n");
       fprintf(fp4,"%lld",TOTAL_TIME_RUN);
       fprintf(fp4,"\n");

       fclose(fp4);

   for (line = 0; line < NUMBER_OF_PARTICLES; line++) {
     fscanf(fp, "%lf %lf %lf", & positions[counter_positions], &
positions[counter_positions + 1], & masses[counter_masses] );
```

```
   //   printf("%lf %lf %lf\n ",  positions[counter_positions],
positions[counter_positions + 1], masses[counter_masses]);

     velocities[counter_velocities] =0;
     velocities[counter_velocities + 1] =0;




     counter_masses += 1;
     counter_positions += 2;
     counter_velocities += 2;
   }



   if (DEBUG_MODE>=2)
           {
                   counter_masses = 0;
                   counter_positions = 0;
                   counter_velocities = 0;

                       for (line = 0; line < NUMBER_OF_PARTICLES; line++)
{

                               // fscanf(fp, "%lf %lf %lf ",  &
positions[counter_positions], & positions[counter_positions + 1], &
masses[counter_masses]);
                               printf("%lf %lf %lf\n ",
positions[counter_positions], positions[counter_positions + 1],
masses[counter_masses]);


                           counter_masses += 1;
                           counter_positions += 2;
                           counter_velocities += 2;


                           }
```

```
           }




   fclose(fp);
 }

 MPI_Bcast(masses, NUMBER_OF_PARTICLES, MPI_DOUBLE, MASTER_PROCESSOR_RANK,
MPI_COMM_WORLD);//send same information
 MPI_Bcast(positions, 2 * num_of_processors * BODIES_per_proc, MPI_DOUBLE,
MASTER_PROCESSOR_RANK, MPI_COMM_WORLD);
 MPI_Scatter(velocities, 2 * BODIES_per_proc, MPI_DOUBLE,
velocities_per_proc, 2 * BODIES_per_proc, MPI_DOUBLE,
MASTER_PROCESSOR_RANK, MPI_COMM_WORLD);
   //send chink of information
 double start_time = MPI_Wtime();

 // We simulate for the specified number of steps.
 int steps = 1;
 for (steps = 1; steps <= TOTAL_TIME_RUN; steps++) {
   compute_force(masses, positions, forces_each_proc, rank_of_processor,
BODIES_per_proc);
   update_positions_velocities(positions, forces_each_proc,
velocities_per_proc, rank_of_processor, BODIES_per_proc);
   MPI_Allgather(MPI_IN_PLACE, 1, aggregat_Type, positions, 1,
aggregat_Type, MPI_COMM_WORLD);




   //just coordinates written in file for simualtion purpose
       if(rank_of_processor == MASTER_PROCESSOR_RANK)
       {

                  int     counter_masses1 = 0;
                   int counter_positions1 = 0;
```

```
                    int counter_velocities1 = 0;



       FILE *fp4 = fopen("justcoordinates.txt","a");

                        for (long long int line = 0; line <
NUMBER_OF_PARTICLES; line++)
                            {


                            // fscanf(fp, "%lf %lf %lf ",  &
positions[counter_positions], & positions[counter_positions + 1], &
masses[counter_masses]);
                            // printf("%lf %lf %lf\n ",
positions[counter_positions], positions[counter_positions + 1],
masses[counter_masses]);
                            fprintf(fp4 ,"%lf ",
positions[counter_positions1]);

                            counter_masses1 += 1;
                            counter_positions1 += 2;
                            counter_velocities1 += 2;

                            }

                            fprintf(fp4,"\n");
                    counter_masses1 = 0;
                    counter_positions1 = 0;
                    counter_velocities1 = 0;

                        for (long long int line = 0; line <
NUMBER_OF_PARTICLES; line++)
                            {


                            // fscanf(fp, "%lf %lf %lf ",  &
positions[counter_positions], & positions[counter_positions + 1], &
masses[counter_masses]);
```

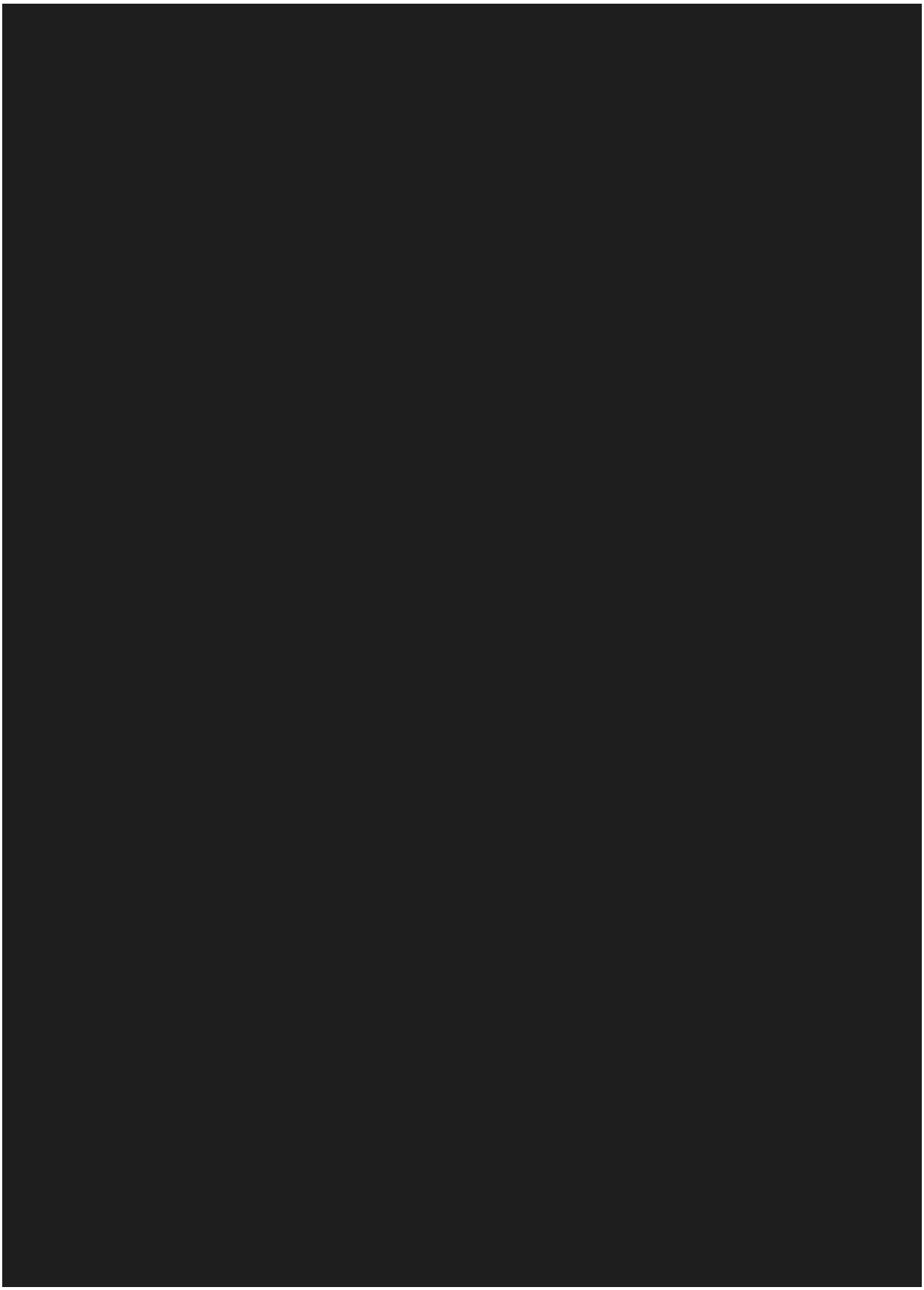
```
                        // printf("%lf %lf %lf\n ",
positions[counter_positions], positions[counter_positions + 1],
masses[counter_masses]);
                        fprintf(fp4 ,"%lf ",
positions[counter_positions1+1]);

                        counter_masses1 += 1;
                        counter_positions1 += 2;
                        counter_velocities1 += 2;

                        }

                        fprintf(fp4,"\n");
                counter_masses1 = 0;
                counter_positions1 = 0;
                counter_velocities1 = 0;




      fclose(fp4);
   }

 }
//inverse of MPI_GAther at upper line
 MPI_Gather(velocities_per_proc, 1, aggregat_Type, velocities, 1,
aggregat_Type, MASTER_PROCESSOR_RANK, MPI_COMM_WORLD);

 if (DEBUG_MODE >= 1 && rank_of_processor == MASTER_PROCESSOR_RANK  )
 {
     FILE * final_state = fopen("final_state.txt", "w+");
     if (!final_state) {
       printf("Error creating output file.\n");
       exit(1);
     }

     int particle = 0;
     for (particle = 0; particle < NUMBER_OF_PARTICLES; particle++) {
```

```
      fprintf(final_state, "%lf %lf %lf %lf %lf\n", masses[particle],
positions[2 * particle], positions[2 * particle + 1], velocities[2 *
particle], velocities[2 * particle + 1]);
    }

    fclose(final_state);
 }

 double end_time = MPI_Wtime();
 if (rank_of_processor == MASTER_PROCESSOR_RANK)
 {
   printf("Time take = %lf s.\n", end_time - start_time);
 }

 MPI_Type_free( & aggregat_Type);
 free(masses);
 free(positions);
 free(velocities_per_proc);
 free(forces_each_proc);
 if (rank_of_processor == MASTER_PROCESSOR_RANK) {
   free(velocities);
 }

 MPI_Finalize();
 return 0;
}









void compute_force(double masses[], double positions[], double
forces_each_proc[], int rank_of_processor, int BODIES_per_proc)
{

 // Starting and ending particle for the current processor.
```

```
 int starting_index = rank_of_processor * BODIES_per_proc;
 int ending_index = starting_index + BODIES_per_proc - 1;

 if (starting_index >= NUMBER_OF_PARTICLES)
 {
   return;
 }
 else if (ending_index >= NUMBER_OF_PARTICLES)
 {
   ending_index = NUMBER_OF_PARTICLES - 1;
 }


 int particle = starting_index;


 for (particle = starting_index; particle <= ending_index; particle++)
 {
     double force_x = 0;
     double force_y = 0;
     int i = 0;
     for (i = 0; i < NUMBER_OF_PARTICLES; i++)
     {
       if (particle == i)
       {
          continue;
       }
       double x_diff = positions[2 * i] - positions[2 * particle];
       double y_diff = positions[2 * i + 1] - positions[2 * particle + 1];
       double distance = sqrt(x_diff * x_diff + y_diff * y_diff);
       double distance_cubed = distance * distance * distance;

       double force_total = GRAVITATIONAL_CONSTANT * masses[i] / distance;
       force_x += GRAVITATIONAL_CONSTANT * masses[i] * x_diff /
distance_cubed;
       force_y += GRAVITATIONAL_CONSTANT * masses[i] * y_diff /
distance_cubed;
     }

     forces_each_proc[2 * (particle - starting_index)] = force_x;
```

```
    forces_each_proc[2 * (particle - starting_index) + 1] = force_y;

    if (DEBUG_MODE >= 2)
    {
        printf("Force on particle %i = %.3f  %.3f\n", particle, force_x,
force_y);
    }

 }



}

void update_positions_velocities(double positions[], double
forces_each_proc[], double velocities_per_proc[], int rank_of_processor,
int BODIES_per_proc)
{

 // Starting and ending particle for the current processor.
 int starting_index = rank_of_processor * BODIES_per_proc;
 int ending_index = starting_index + BODIES_per_proc - 1;

 if (starting_index >= NUMBER_OF_PARTICLES) {
   return;
 } else if (ending_index >= NUMBER_OF_PARTICLES) {
   ending_index = NUMBER_OF_PARTICLES - 1;
 }

 int particle = starting_index;
 for (particle = starting_index; particle <= ending_index; particle++)
 {
    //   s = ut+1at^2
    positions[2 * particle] += velocities_per_proc[2 * (particle -
starting_index)] * DELTA_T + (forces_each_proc[2 * (particle -
starting_index)] * DELTA_T * DELTA_T / 2);
    positions[2 * particle + 1] += velocities_per_proc[2 * (particle -
starting_index) + 1] * DELTA_T + (forces_each_proc[2 * (particle -
starting_index) + 1] * DELTA_T * DELTA_T / 2);
```

```
    // v = u+at
    velocities_per_proc[2 * (particle - starting_index)] +=
forces_each_proc[2 * (particle - starting_index)] * DELTA_T;
    velocities_per_proc[2 * (particle - starting_index) + 1] +=
forces_each_proc[2 * (particle - starting_index) + 1] * DELTA_T;


    if (DEBUG_MODE >= 2)
    {
        printf("Position of particle %i = %.3f  %.3f\n", particle,
positions[2*particle], positions[2*particle + 1]);
    }
 }


}
```



OPENMP

```cpp
// g++ -fopenmp 2nd.cpp -o run1 -lm
// OMP_NUM_THREADS=4 ./run1

#include<omp.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include<iostream>
typedef long long int lld;
#define ToalTimeStep 10000
// #define ToalTimeStep 20

using namespace std;


// #define Gconstant 6.6674 * pow(10,-11)
double  Gconstant = 6.6674 * pow(10,-11);



int main(int argc, char const *argv[])
{

   lld n;
   int zemp;
```

```
// cin>>n;
// double *posx,*posy,*mass; //in kg
// double *velex,*veley; //in m/s

double posx[120000];
double posy[120000];
double mass[120000];
double velex[120000];
double veley[120000];


double m1,newposxi1,newposyi1,m2,r1vector,r2vector,rdiffector;
double Forcex1,Forcey2;
double accelerationx,accerlerationy;
double tempvx1,tempvy2;
double deltat = 5;
double taoltime1 = 0;
// posx = new double[n];
// posy = new double[n];
// mass = new double[n];
// velex = new double[n];
// veley = new double[n];

FILE *fp1,*fp2,*fp3,*fp4;

fp1 = fopen("inputfile.txt","r");
fp4 = fopen("justcoordinates.txt","w");
fp2 = fopen("outputtxt.txt","w");
fp3 = fopen("outputtonlytimedone.txt","w");


fscanf(fp1,"%lld",&n);
n = 4000 ;//only 2000 objects taken
cout<<n;
lld numoflines = ToalTimeStep/deltat;
fprintf(fp4,"%lld",n);

fprintf(fp4,"\n");
fprintf(fp4,"%lld",numoflines);
fprintf(fp4,"\n");
```

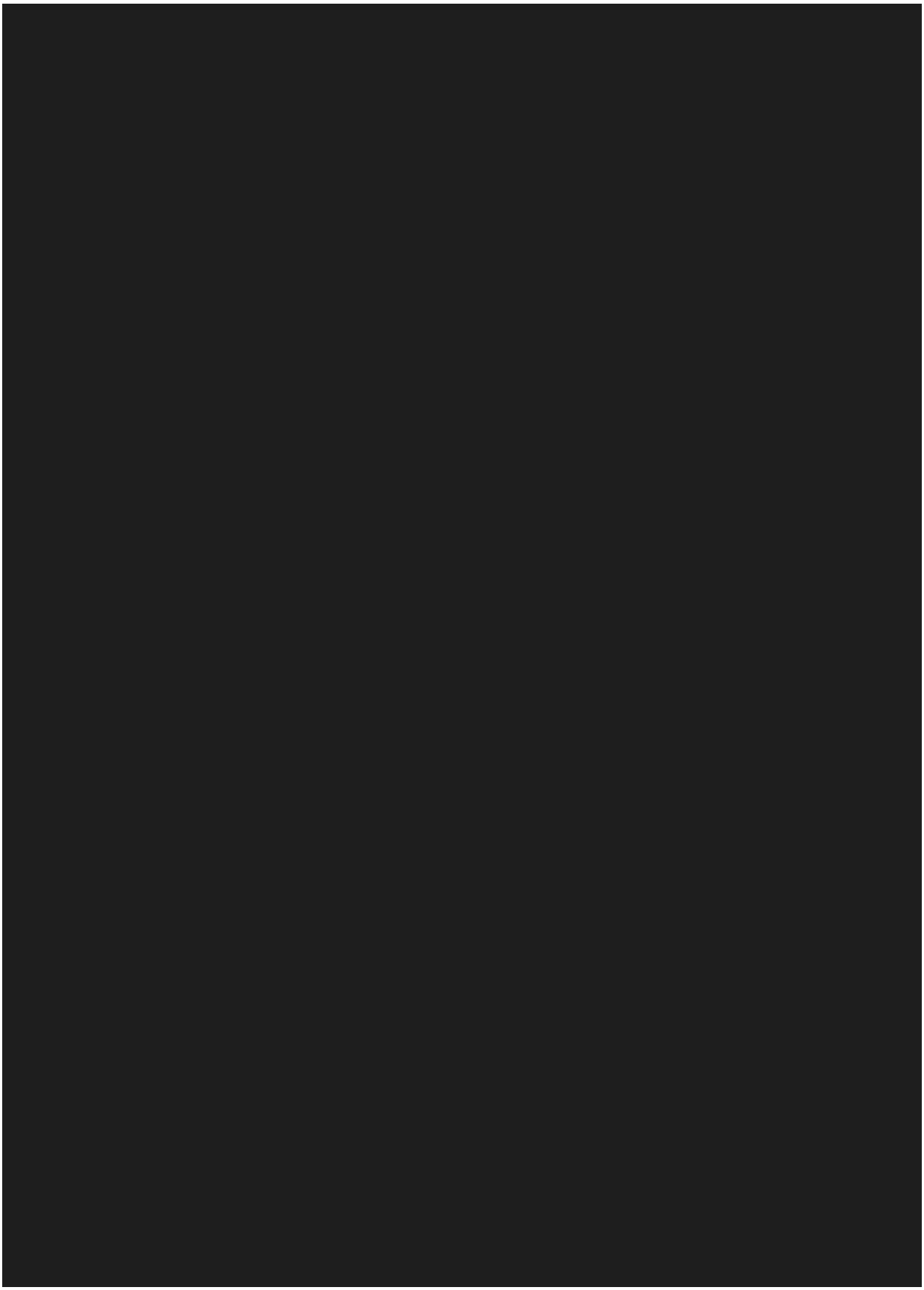
```
   for (lld rep = 0; rep < n; rep++)
       {
           // cin>>posx[rep]>>posy[rep]>>mass[rep];

           fscanf(fp1, "%lf %lf %lf", &posx[rep],&posy[rep],&mass[rep]);
           velex[rep] = veley[rep] = 0;
       }
   // for (int rep = 0; rep < n; rep++)
   //     {
   //         // cout<<posx[rep]<<" "<<posy[rep]<<" "<<mass[rep]<<endl;;
   //             fprintf(fp2,"%lf %lf ",posx[rep],posy[rep]);
   //             fprintf(fp2,"\n");
   //     }









    for (lld timestep = 0; timestep*deltat < ToalTimeStep ; timestep++)
   {
       fprintf(fp2,"%lf",timestep*deltat);
       fprintf(fp2,"\n");
       lld j1;

       double tbegin = omp_get_wtime();

       // #pragma omp for
       #pragma omp parallel for schedule(static)
private(j1,r1vector,r2vector,m1,m2,Forcex1,Forcey2,rdiffector,acceleration
x,accerlerationy,velex,veley)
       for (lld i1 = 0; i1 < n; i1++)
       {
           //each body
           // current status then update each body wrt static position of
```

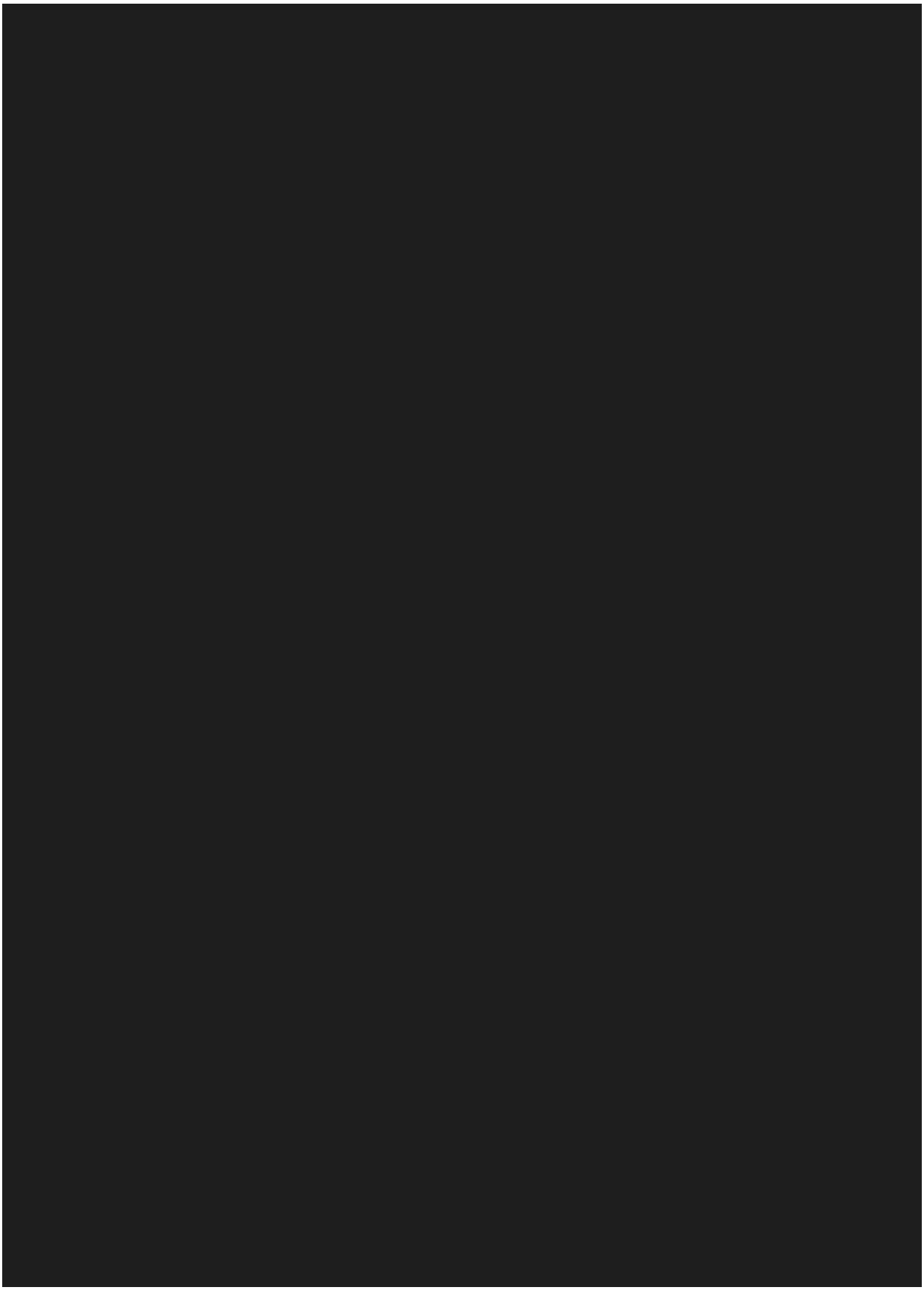
```
           m1 = mass[i1];
           newposxi1 = posx[i1];newposyi1 = posy[i1];
           Forcex1 = 0;
           Forcey2 = 0;
           //consider all bodies except i1 th to effect n(i1)
           for (j1 = 0; j1 < n ; j1++)
           {
               if (j1!=i1)
               {
                   m2 = mass[j1];
                   r1vector = posx[j1]-posx[i1] ; //wrt to current body
                   r2vector = posy[j1]-posy[i1] ; //wrt to current body
                   rdiffector  = r1vector*r1vector+r2vector*r2vector;
                   rdiffector = pow(rdiffector,1.5);

                   Forcex1 = Forcex1+
(((Gconstant*m1*m2)/rdiffector)*r1vector) ;
                   Forcey2 = Forcey2+
(((Gconstant*m1*m2)/rdiffector)*r2vector) ;



               }

           }
           // time difference delatat delta

           accelerationx = Forcex1/m1;
           tempvx1 = velex[i1] + accelerationx*deltat;
           posx[i1] += (velex[i1]*deltat+(0.5*accelerationx*deltat*deltat)
);
           velex[i1] = tempvx1;

   //moement in y direction

           accerlerationy = Forcey2/m1;
           tempvy2 = veley[i1] + accerlerationy*deltat;
           posy[i1] +=
(veley[i1]*deltat+(0.5*accerlerationy*deltat*deltat) );
           veley[i1] = tempvy2;
```

```
}

// //debug mode
// cin>>zemp;
// cout<<"FORCE1"<<Forcex1;
// cout<<"FORCE2"<<Forcey2;
// cin>>zemp;

  double wtime = omp_get_wtime() - tbegin;
  fprintf( fp3,"%lf", wtime );
  fprintf(fp3,"\n");
  taoltime1 +=wtime;




//print data at each time step
for (lld xdata = 0; xdata < n; xdata++)
{

    // printf("%lf,",posx[xdata]);
    fprintf(fp2,"%lf ",posx[xdata]);
    fprintf(fp4,"%lf ",posx[xdata]);

}
fprintf(fp2,"\n");
fprintf(fp4,"\n");


for (lld ydata = 0; ydata < n; ydata++)
{
    // printf("%lf,",posx[ydata]);
```

```
            fprintf(fp2,"%lf ",posy[ydata]);
            fprintf(fp4,"%lf ",posy[ydata]);

        }

        fprintf(fp2,"\n");
        fprintf(fp4,"\n");


        fprintf(fp2,"\n");




    }

    fclose(fp1);
    fclose(fp2);
    fclose(fp3);
    fclose(fp4);
```

```
        cout<<"\nTime "<<taoltime1<<endl;




    // delete []posx;
    // delete []posy;
    // delete []mass;
    // delete []velex;
    // delete []veley;


    return 0;
}
```





CUDA

```
#include <stdio.h>
#include <stdlib.h>
// #include <cutil.h>
#include<time.h>
#include <iostream>
#include <string>
#include <vector>

#define DEBUG_MODE 0
#define MASTER_PROCESSOR_RANK 0
#define DELTA_T 10
#define INPUT_FILE_DATA "inputfile.txt"
```

```
#define ToalTimeStep 10000
const float DEG2RAD = 3.14159/180;

using namespace std;
double  Gconstant = 6.6674 * pow(10,-11);
// #define Gconstant 6.6674 * pow(10,-11)


typedef long long int lld;

lld numeroflines ;


// double *arrx1 *arry2;

// __global__ void kernelcomputenewvel(double* idx ,double *masslist,
double *velocitylistx,double *velocitylisty, double
*newvelocitylistx,double *newvelocitylisty,double *postionx,double
*postiony,double *newpostionx,double *newpostiony ,lld threadcunt )
__global__ void kernelcomputenewvel(double masslist[], double
velocitylistx[],double velocitylisty[], double postionx[],double
postiony[],double Forcex1[],double Forcey2[] ,lld idx ,lld threadcunt,lld
n )
{
   // lld threadid = blockDim.x * blockIdx.x + threadIdx.x;
   lld threadid =  threadIdx.x;

   double m2 ,r1vector,r2vector,rdiffector;


   Forcex1[threadid] = 0;
   Forcey2[threadid] = 0 ;


   for(lld i1 = threadid;i1<n;i1+=threadcunt)
   {
       // c1temp[i1] = a1[i1]+b1[i1];
       if(i1!=idx)
       {
           m2 = masslist[i1] ;
```

```
           r1vector = postionx[i1] - postionx[idx];
           r2vector = postiony[i1] - postiony[idx];

           rdiffector = r1vector*r1vector+r2vector*r2vector;
           rdiffector = pow(rdiffector,1.5);
           Forcey2[threadid] = Forcey2[threadid] +
((m2/rdiffector)*r2vector) ;
           Forcex1[threadid] = Forcex1[threadid] +
((m2/rdiffector)*r1vector) ;

       }
   }

   // __syncthreads(); // barrier point


}









int main(int argc, char **argv)
{
   int x1,x2;
   int deltavartobeused;
   // cout<<argc;
   double deltat = DELTA_T;

   if (argc-1==2)
   {
       //argv[1] number of threads ,number of bodies
        x1 = atoi(argv[1]);
        x2 = atoi(argv[2]);
       // cout<<x1;
```

```
}



FILE * fp = fopen(INPUT_FILE_DATA, "r");
if (!fp) {
  printf("Error opening input file.\n");
  exit(1);
}

FILE *fp2,*fp3,*fp4;

// fp4 = fopen("justcoordinates.txt","w+");
fp2 = fopen("outputtxt.txt","w");
fp3 = fopen("outputtonlytimedone.txt","w");
long long int nobj;
 double m1,newposxi1,newposyi1,m2,r1vector,r2vector,rdiffector;
double Forcex1,Forcey2;

double accelerationx,accerlerationy;
double tempvx1,tempvy2;
double average_kernel_time = 0;



//host particle storage
double posx[120000];
double posy[120000];
double mass[120000];
double velex[120000];
double veley[120000];
double *forecearrayx,*forecearrayy;
double *hoforecearrayx,*hoforecearrayy;



//device particle storage
double *dposx;
```

```
double *dposy;
double *dmass;
double *dvelex;
double *dveley;

lld maxnumberofobjs ;

// lld threads_in_block = 1024;
lld threads_in_block = x1;

long long int array_siez,numer_of_lines;

numer_of_lines = ToalTimeStep/deltat;

  fscanf(fp, "%lld", &maxnumberofobjs);
  cout<<"MAXIMUM NUMBER of objects "<<maxnumberofobjs<<endl;
nobj = x2 ;

  cout<<"Selected number of objects "<<nobj<<endl;
  cout<<"Number of Threads "<<threads_in_block<<endl;
  cout<<"Deltat "<<deltat<<endl;


    fp4 = fopen("justcoordinates.txt","w+");

    fprintf(fp4,"%lld",nobj);

    fprintf(fp4,"\n");
    fprintf(fp4,"%lld",numer_of_lines);
    fprintf(fp4,"\n");

    // fclose(fp4);
for (lld rep = 0; rep < nobj; rep++)
{

    fscanf(fp, "%lf %lf %lf", &posx[rep],&posy[rep],&mass[rep]);
    velex[rep] = veley[rep] = 0;
}

if(DEBUG_MODE>=2)
```

```
{
    for (lld rep = 0; rep < nobj; rep++)
    {

        fprintf(fp3, "%lf %lf %lf\n", posx[rep],posy[rep],mass[rep]);
    }

}



float et;
cudaEvent_t start, stop;
 clock_t startc, end;
startc = clock();



// fclose(fp);
array_siez = nobj;
 lld array_bytes = array_siez*sizeof(double);
 lld threadsize = threads_in_block*sizeof(double);
hoforecearrayx = (double*)malloc(threadsize);
hoforecearrayy = (double*)malloc(threadsize);



cudaMalloc((void **)&dposx,array_bytes );
cudaMalloc((void **)&dposy,array_bytes );
cudaMalloc((void **)&dmass,array_bytes );
cudaMalloc((void **)&dvelex,array_bytes );
cudaMalloc((void **)&dveley,array_bytes );
cudaMalloc((void **)&forecearrayx,threadsize );
cudaMalloc((void **)&forecearrayy,threadsize );


for (lld timestep = 0; timestep*deltat < ToalTimeStep ; timestep++)
    {
        // fprintf(fp2,"%lf",timestep*deltat);
        // fprintf(fp2,"\n");
```

```
          for(lld i1 = 0 ;i1<nobj; i1++)
          {

              cudaEventCreate(&start);
              cudaEventCreate(&stop);
              cudaEventRecord(start,0);
              cudaEventRecord(stop,0);




          cudaMemcpy(dposx,posx,array_bytes,cudaMemcpyHostToDevice);
          cudaMemcpy(dposy,posy,array_bytes,cudaMemcpyHostToDevice);
          cudaMemcpy(dmass,mass,array_bytes,cudaMemcpyHostToDevice);
          cudaMemcpy(dvelex,velex,array_bytes,cudaMemcpyHostToDevice);
          cudaMemcpy(dveley,veley,array_bytes,cudaMemcpyHostToDevice);




kernelcomputenewvel<<<1,threads_in_block>>>(dmass,dvelex,dveley,dposx,dpos
y,forecearrayx,forecearrayy,i1,threads_in_block,nobj);
// __global__ void kernelcomputenewvel(double idx ,double masslist[],
double velocitylistx[],double velocitylisty[], double postionx[],double
postiony[],lld threadcunt,double Forcex1[],double Forcey2[] )




cudaMemcpy(hoforecearrayx,forecearrayx,threadsize,cudaMemcpyDeviceToHost);

cudaMemcpy(hoforecearrayy,forecearrayy,threadsize,cudaMemcpyDeviceToHost);

          cudaEventSynchronize(stop);
      cudaEventElapsedTime(&et, start, stop);
      cudaEventDestroy(start);
      cudaEventDestroy(stop);
```

```
          if(DEBUG_MODE>=1)
          {

              printf("\nGPU Time to generate kernel %lld:  %f  \n",
timestep*nobj+i1,et);
          }
          average_kernel_time+=et;

          Forcex1 = 0;Forcey2 = 0;
          for(int ij1=0;ij1<threads_in_block;ij1++)
          {
              Forcex1 += hoforecearrayx[ij1];
              Forcey2 += hoforecearrayy[ij1] ;
          }


          //once done update with new vector

          m1 = mass[i1];

          // accelerationx = Gconstant*Forcex1/m1;
          accelerationx = Gconstant*Forcex1;

          tempvx1 = velex[i1] + accelerationx*deltat;
          posx[i1] += (velex[i1]*deltat+(0.5*accelerationx*deltat*deltat)
);
          velex[i1] = tempvx1;

   //moement in y direction

          // accerleration y = Gconstant*Forcey2/m1;
          accerleration y = Gconstant*Forcey2;

          tempvy2 = veley[i1] + accerleration y*deltat;
          posy[i1] +=
(veley[i1]*deltat+(0.5*accerleration y*deltat*deltat) );
          veley[i1] = tempvy2;
```

```
            if(DEBUG_MODE>=2)
                {
                  cout<<" acceleration at each step of
objects"<<accelerationx<<" "<< accerlerationy << endl;
                }




          }


          //write postion x ,position y


          for (lld xdata = 0; xdata < nobj; xdata++)
          {

              // printf("%lf,",posx[xdata]);
              // fprintf(fp2,"%lf ",posx[xdata]);
              fprintf(fp4,"%lf ",posx[xdata]);

          }
          fprintf(fp4,"\n");



          for (lld ydata = 0; ydata < nobj; ydata++)
          {
              // printf("%lf,",posx[ydata]);

              fprintf(fp4,"%lf ",posy[ydata]);

          }
          fprintf(fp4,"\n");

      }


  end = clock();
```

```
   printf("\naverage kernel time :  %f  \n",
float((deltat*average_kernel_time) /(ToalTimeStep*nobj) ) );
   printf("\nTotal time:  %f  \n", float(end-startc) );



       cudaFree(dposx );
       cudaFree(dposy );
       cudaFree(dmass );
       cudaFree(dvelex );
       cudaFree(dveley );
       cudaFree(forecearrayx );
       cudaFree(forecearrayy );


       // cudaFree(d_in1);
       // cudaFree(d_in2);
       // cudaFree(d_out);
       free(hoforecearrayx);
       free(hoforecearrayy);
       // hoforecearrayx = new double[threads_in_block];
   // hoforecearrayy = new double[threads_in_block];

   fclose(fp);
   fclose(fp2);
   fclose(fp3);
   fclose(fp4);

       return 0;



}
```

[redacted]