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);
 }
```

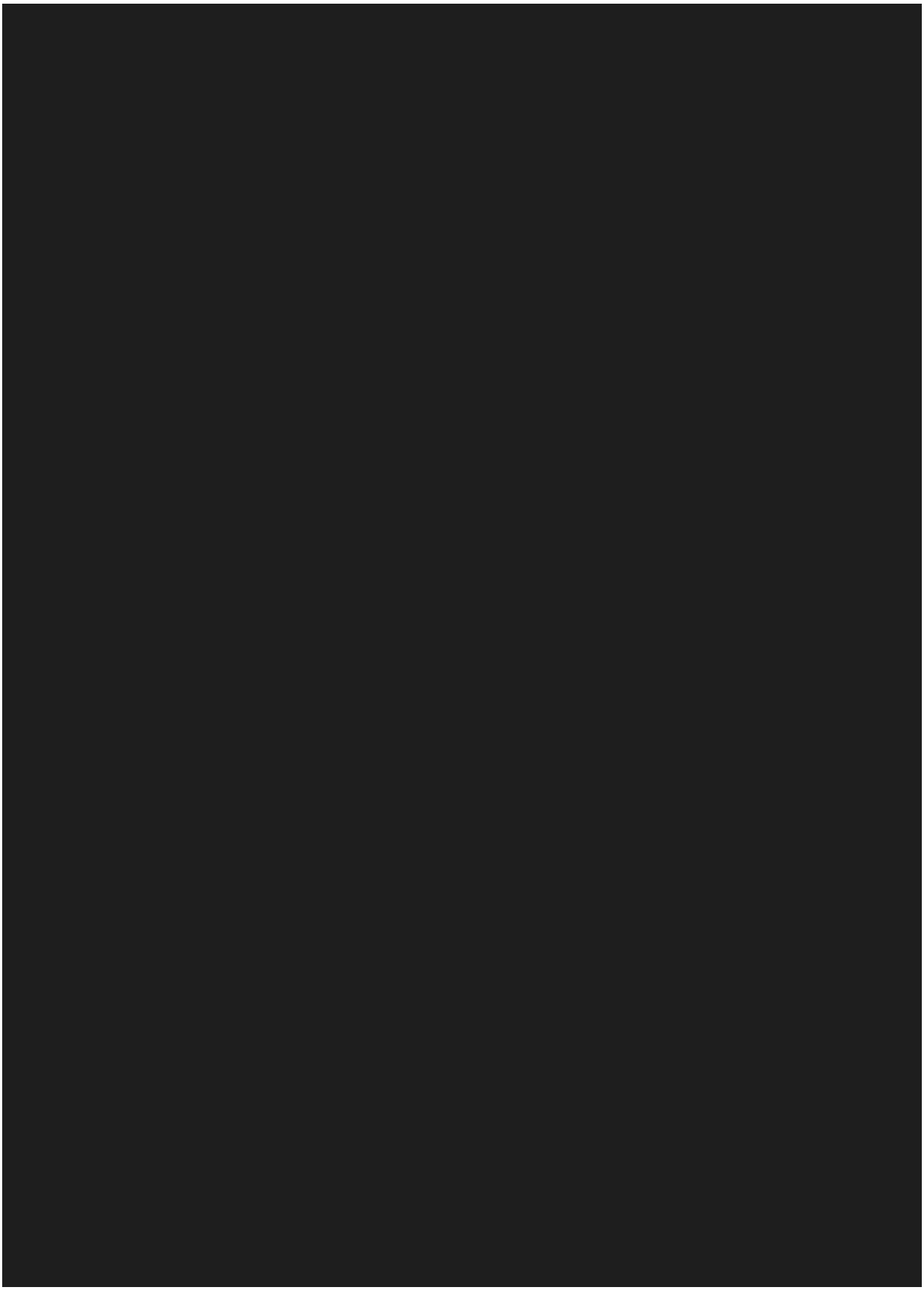
```
 double end_time = MPI_Wtime();
 if (rank_of_processor == MASTER_PROCESSOR_RANK)
 {
   printf("Time take = %lf s.\n", end_time - start_time);
 }

 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

```
// 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;







       }
```

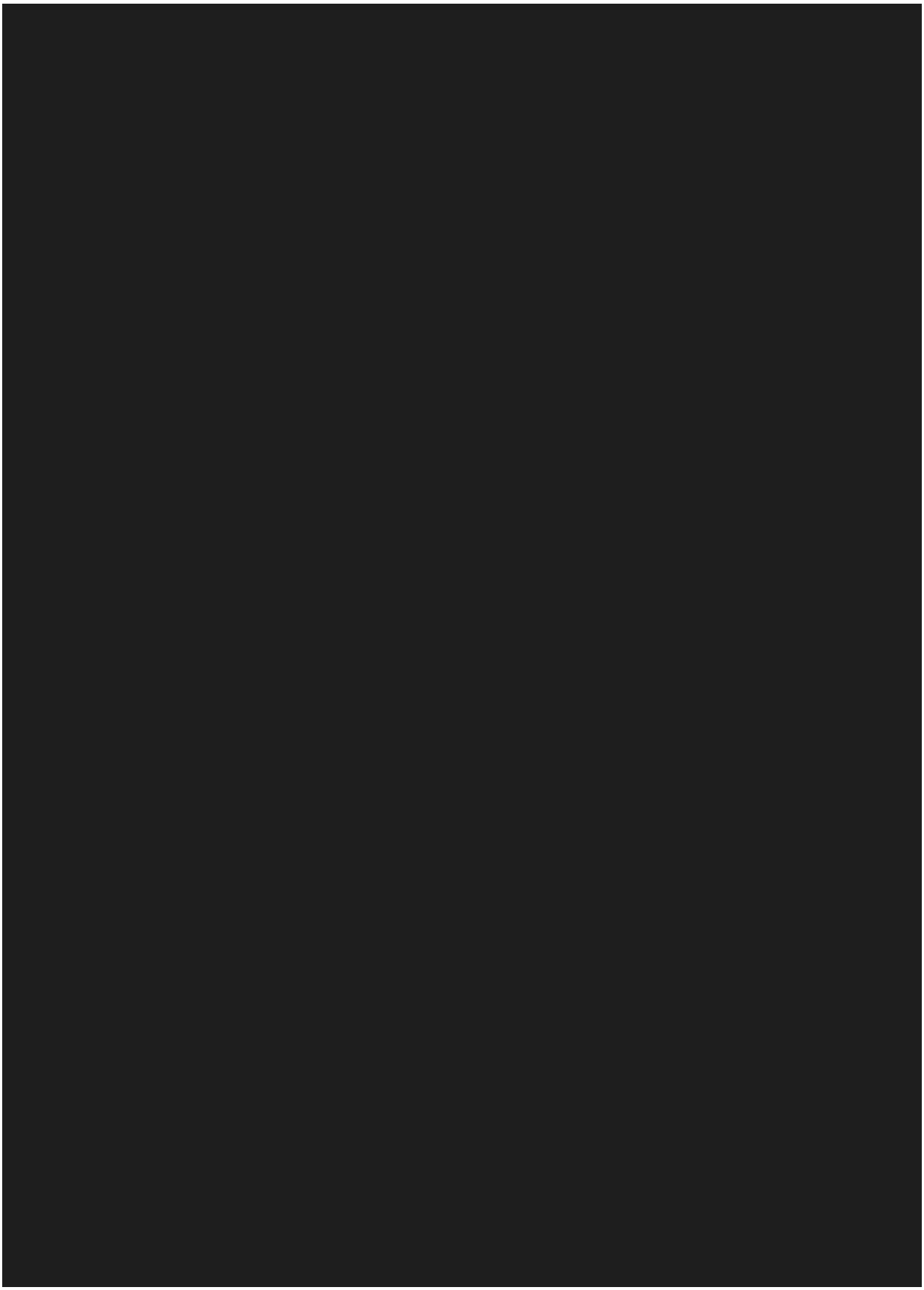
```
// //debug mode
// cin>>zemp;
// cout<<"FORCE1"<<Forcex1;
// cout<<"FORCE2"<<Forcey2;
// cin>>zemp;

  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");
```

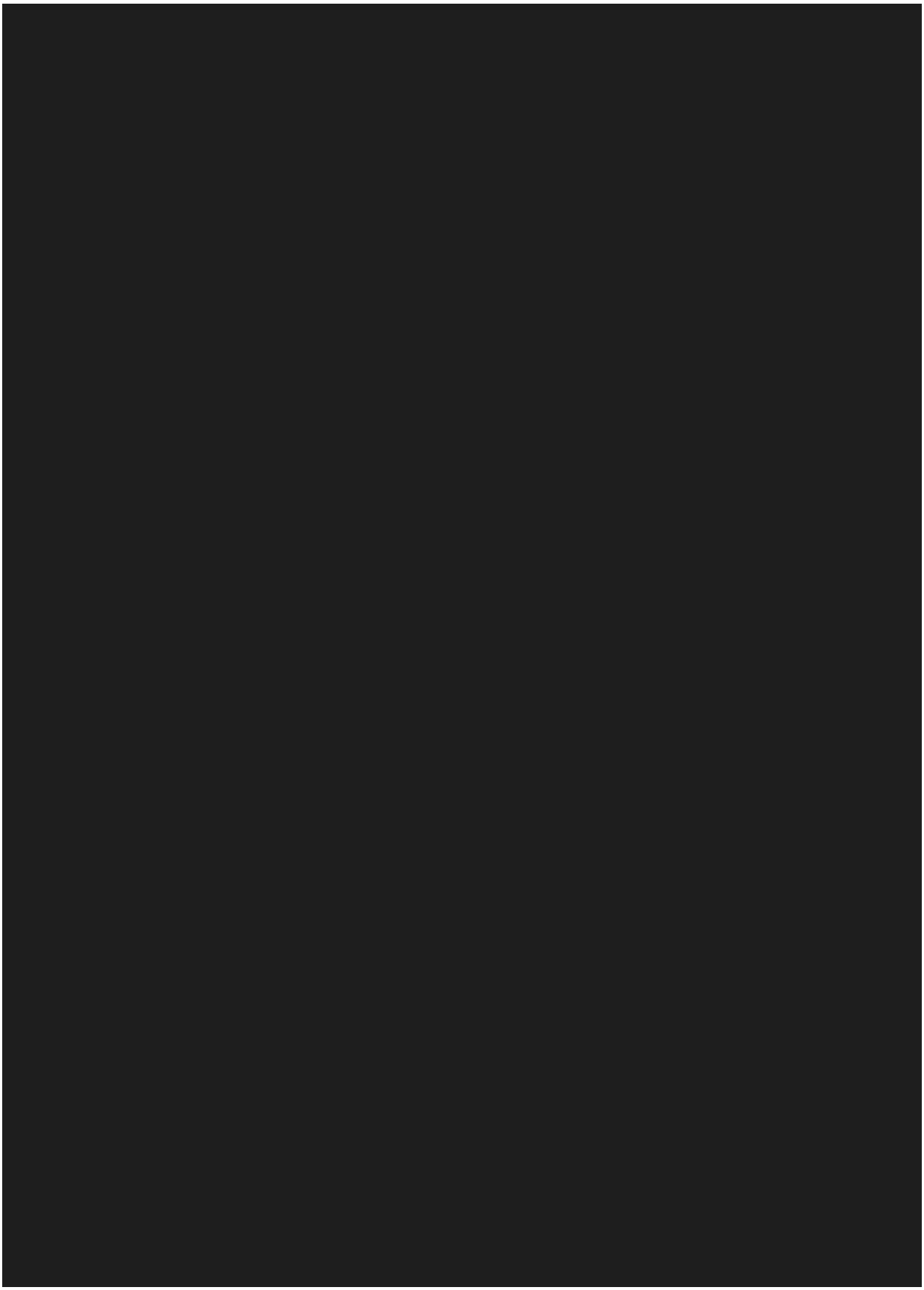
```
    fprintf(fp2,"\n");



}
```

```
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<<" "<< accerleration y << 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;



}
```