```c
#include <stdio.h>
#include <math.h>
#include <float.h>
#include <stdlib.h>
#include<string.h>
#include<time.h>
#include <mpi.h>

int Np ;
int Np_d = 100; /* Default number of particles*/

float total_time ;
float time_d = 60; /* Default simulation time*/

float dt;
float dt_d =0.02; /* Default time step*/

float m = 10.0; /* mass */
int range = 20; /* range for position generation */

int rank,size; /* world rank and size*/

typedef struct {
   float x, y, z;
} Pos;            /* struct for position co-ordinates*/
Pos *position;

typedef struct {
    float vx,vy,vz;
} Vel;            /* struct for velocity */
Vel *velocity;

typedef struct {
    float ax,ay,az;
} Acc;            /* struct for acceleration */
Acc *a;

int lengths[3] = { 1, 1, 1 };
const MPI_Aint displacements[3] = { 0, sizeof(float), sizeof(float)*2 };
MPI_Datatype types[3] = { MPI_FLOAT, MPI_FLOAT, MPI_FLOAT }; /* to create MPI data types for position,velocity and acceleration */
MPI_Datatype POSITION;
MPI_Datatype VELOCITY;

int start,end,*sendcounts,*displs; /* variables to control imbalanced scatter/gather*/

void P_r(int number_of_particles,int size , int rank ,int *start ,int *end,int *sendcounts,int *displs )
{
int r = (number_of_particles)%size;
int s = 0;

for (int p = 0; p < size; p++)
    {
         sendcounts[p] = ((number_of_particles)/size); /* array of number of particles assigned for current rank*/
         if (r > 0) {
            sendcounts[p]=sendcounts[p]+1;
            r--;}

         displs[p] = s; /*start boundary of particles in current rank */
         s += sendcounts[p];
    }

if (rank==(size-1))
{
    *start = displs[rank];
    *end = number_of_particles;
}
else
{
    *start = displs[rank]; /* start n end in case of parallelizing loops*/
    *end = displs[rank+1];
}
}

void generate_pos_vel() /* Function to initialize position and velocity*/
{
    for(int i=0; i<Np; i++)
        {
            position[i].x= (rand()/((double)RAND_MAX + 1))*range;
            position[i].y= (rand()/((double)RAND_MAX + 1))*range;
            position[i].z= (rand()/((double)RAND_MAX + 1))*range;

            velocity[i].vx= ((double)rand()/RAND_MAX*2.0-1.0)*0.0001;
            velocity[i].vy= ((double)rand()/RAND_MAX*2.0-1.0)*0.0001;
            velocity[i].vz= ((double)rand()/RAND_MAX*2.0-1.0)*0.0001;
        }
}

void cal_acceleration() /* function to calculate resultant acceleration of current particle in particular dt*/
{
        float del_x,del_y,del_z,invr,invr3,f;

        for(int i=0;i<sendcounts[rank];i++)
        {
            a[i].ax = 0.0;
            a[i].ay = 0.0;
            a[i].az = 0.0;
            for(int j=0;j<Np;j++)
            {
                if((i+(displs[rank])) != j)
                {
                    del_x = position[i+(displs[rank])].x - position[j].x;
                    del_y = position[i+(displs[rank])].y - position[j].y;
                    del_z = position[i+(displs[rank])].z - position[j].z;
                    invr = 1.0 / sqrt((del_x*del_x)+(del_y*del_y)+(del_z*del_z)+DBL_EPSILON);
                    invr3 = pow(invr,3);
                    f = m*invr3;

                    a[i].ax += f*del_x;
                    a[i].ay += f*del_y;
                    a[i].az += f*del_z;
                }
            }
        }
}

void cal_vel() /* function to calculate resultant velocity of current particle in particular dt*/
{
    for (int i = 0; i < sendcounts[rank]; i++)
    {
        velocity[i+(displs[rank])].vx     += (dt*a[i].ax);
        velocity[i+(displs[rank])].vy     += (dt*a[i].ay);
        velocity[i+(displs[rank])].vz     += (dt*a[i].az);
    }
}

void cal_position() /* function to update position of particles from resultant velocity and acceleration*/
{
    for (int i = 0; i < sendcounts[rank]; i++)
    {
        position[i+(displs[rank])].x += (dt*velocity[i+(displs[rank])].vx)+(0.5*dt*dt*a[i].ax);
        position[i+(displs[rank])].y += (dt*velocity[i+(displs[rank])].vy)+(0.5*dt*dt*a[i].ay);
        position[i+(displs[rank])].z += (dt*velocity[i+(displs[rank])].vz)+(0.5*dt*dt*a[i].az);
    }
}

void write_p() /* function to write the updated positions at all time steps*/
{
    FILE *fp1 = fopen("pos.txt", "a");
    for(int i=0; i<Np; i++)
    {
        fprintf(fp1,"%f %f %f \n",position[i].x,position[i].y,position[i].z);
    }
    fprintf(fp1,"\n");
    fclose(fp1);
}

void sim() /* function to start n particle simulation*/
{
    MPI_Bcast(position,Np,POSITION,0,MPI_COMM_WORLD);
    MPI_Bcast(velocity,Np,VELOCITY,0,MPI_COMM_WORLD);

    int iterations = total_time / dt;

    for(int t=0;t<iterations;t++)
    {
        cal_acceleration();
        cal_vel();
        cal_position();
```

```c

MPI_Allgatherv(position+(displs[rank]),sendcounts[rank],POSITION,position,sendcounts,displs,POSITION,MPI_COMM_WORLD);
MPI_Allgatherv(velocity+(displs[rank]),sendcounts[rank],VELOCITY,velocity,sendcounts,displs,VELOCITY,MPI_COMM_WORLD);

    if(rank==0)
       { write_p();}
    }
}

int main(int argc, char **argv)
{
   srand(time(0));
   MPI_Init(&argc, &argv); /* Initiate MPI*/

   MPI_Barrier(MPI_COMM_WORLD);
   double tcal = MPI_Wtime(); /* Computation start time*/

   if (argc >= 2)
      Np = atoi(argv[1]);
   else
      Np = Np_d;

   if (argc >= 3)
      total_time    = atoi(argv[2]);      /* user input through command line arguments */
   else
      total_time    = time_d;

   if (argc >= 4)
      dt = atof(argv[3]);
   else
      dt = dt_d;

   MPI_Comm_size(MPI_COMM_WORLD,&size); /* size and rank of MPI*/
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);

   MPI_Type_create_struct(3, lengths, displacements, types, &POSITION);
   MPI_Type_commit(&POSITION);                                      /*committing user defined MPI data types*/

   MPI_Type_create_struct(3, lengths, displacements, types, &VELOCITY);
   MPI_Type_commit(&VELOCITY);

   sendcounts = malloc(sizeof(int)*size); /* allocating memory for arrays*/
   displs = malloc(sizeof(int)*size);

   P_r(Np,size,rank,&start,&end,sendcounts,displs); /* calling function to calculate send counts and displacements*/

   position = (Pos *) malloc(Np * sizeof(Pos));   /* allocating memory for structs*/
   velocity = (Vel *) malloc(Np * sizeof(Vel));
   a        = (Acc *) malloc(sendcounts[rank] * sizeof(Acc));

   if (rank == 0)
        {
          generate_pos_vel(); /* initializing position and velocity in root process*/

         FILE *fp = fopen("pos.txt", "w"); /*writing initial positions*/
         for(int i=0; i<Np; i++)
           {
                fprintf(fp,"%f %f %f \n",position[i].x,position[i].y,position[i].z);
           }
         fprintf(fp,"\n");
         fclose(fp);
        }

   sim(); /* function call for particle simulation */

   MPI_Barrier(MPI_COMM_WORLD);
   double elapsedt = MPI_Wtime() - tcal; /* computational time taken to complete the process*/

   if (rank==0)
    {
            printf("Np=%d,Total time= %f,dt=%f \n",Np,total_time  ,dt);
        printf("run time = %f \n",elapsedt);
    }

   MPI_Finalize();
}

```