Simulazione parallela di un sistema gravitazionale ad N corpi

Sistemi di calcolo paralleli e distribuiti

2 aprile 2023

Matteo Camanni

Matr. 812320

Indice

1 Progettazione

1.1 Problema scientifico e algoritmo scelto

Il codice ha lo scopo di risolvere un sistema gravitazionale ad N corpi nell'ambito della fisica classica. L'interazione gravitazionale classica tra due corpi è descritta dall'equazione di Newton:

$$F = G\frac{m_1 m_2}{r^2} \tag{1}$$

Essa è risolvibile analiticamente solo se il numero dei corpi massivi coinvolti è limitato a due oppure per tre corpi in configurazioni specifiche. Per risolvere un sistema generale con N corpi ci si deve quindi affidare ad algoritmi numerici. Una simulazione numerica, se opportunamente tarata, sarà in grado di descrivere l'evolvere del sistema con un buon grado di approssimazione sui tempi scala desiderati. In questo codice si è fatto uso di un algoritmo simpletico *"Velocity Verlet"* per risolvere l'equazione del moto. Esso agisce in tre fasi:

$$v\left(t + \frac{dt}{2}\right) = v(t) + \frac{dt}{2}g(x(t)) \tag{2}$$

$$x(t + dt) = x(t) + v\left(t + \frac{dt}{2}\right)dt \tag{3}$$

$$v(t + dt) = v\left(t + \frac{dt}{2}\right) + \frac{dt}{2}g(x(t + dt)) \tag{4}$$

$$g(t) = G\frac{m}{r^2(t)} \tag{5}$$

Sono necessari due valori dell'accelerazione gravitazionale ad ogni step temporale ma il secondo sarà uguale al primo del passaggio successivo.
Questo modello si applica ad ogni coppia di corpi facente parte del sistema in esame in maniera indipendente e quindi per ognuna di esse ci sarà un'equazione da aggiungere al sistema. Per ogni step temporale ci saranno quindi N-1 accelerazioni da calcolare per ogni corpo. Questo, soprattutto con un elevato valore di N, genera un altissimo numero di operazioni. Per ridurre il problema si è utilizzato l'algoritmo di Barnes-Hut. Per prima cosa lo spazio (in questo caso cubico) viene diviso in ottanti calcolando ricorsivamente il centro di massa di ogni nodo dell'octree che si viene così a formare. Si considerano tutti gli ottanti contenenti almeno un corpo fino a dei nodi "foglia" che conterranno i singoli corpi del sistema. Al momento del calcolo della gravità si applica una condizione di clustering per cui per i corpi più lontani si considereranno i centri di massa dei nodi che li contengono riducendo considerevolmente i calcoli necessari con un'approssimazione minima. Questo soprattutto con molti corpi e/o con distribuzioni disomogenee, casi comuni in questo tipo di sistema.

1.2 Parallelizzazione

Per parallelizzare il codice si utilizza un modello di programmazione message passing implementato tramite la libreria MPI per C++. L'algoritmo di Barnes-Hut si basa su una divisione del problema quindi si è impiegato un paradigma dividi e conquista.
La struttura del codice è la seguente (sono indicati anche i termini portati alla legge di Amdahl. t_s sono operazioni sequenziali, t_c tempi di comunicazione e t_p operazioni parallele):

1.2.1 Importo dei dati da file - t_{s1}, t_{c1}

La prima sezione del codice è sequenziale. Il primo processo si occupa di leggere da un file i valori di massa, posizione e velocità iniziali dei corpi del sistema ed informa gli altri processi sul loro numero tramite una primitiva Broadcast. Questa comunicazione serve per allocare i vari array necessari.

1.2.2 Divisione del problema - t_{s2}, t_{sp1}, t_{c2}

Una volta importati i dati il problema viene suddiviso in sottoproblemi analoghi seguendo il paradigma dividi e conquista. La divisione avviene in ottanti ed è ricorsiva. Il primo processo divide in otto sottogruppi i corpi inviandoli ai processi fratelli con una primitiva Scatter (operazione sequenziale). Se i processi usati sono più di otto i primi otto in parallelo effettueranno una nuova divisione ridistribuendo ulteriormente i dati. Questa operazione è analoga ad una divisione ad albero ed è "semiparallela" nel senso che è parallelizzata solo su un sottoinsieme dei processi.

1.2.3 Costruzione dell'Octree locale e computazione della gravità - t_{p1}, t_{c3}

Il paradigma dividi e conquista prevede di risolvere le parti del problema come se fossero il problema completo. Si sfrutta quindi una funzione scritta per una versione sequenziale del codice per costruire un octree locale per ognuno dei nodi cubici di pertinenza di un processo e calcolare il vettore di accelerazione gravitazionale di ognuno dei corpi a lui affidati.
I sottosistemi non sono però veramente indipendenti. Una volta costruiti gli octree locali i processi si scambiano i centri di massa tramite una primitiva Allgather e verificano la condizione di clustering. I corpi per cui è verificata sono gestibili localmente mentre gli altri vengono inviati agli altri processi per aggiungere i termini gravitazionali dati dai sottonodi dagli octree locali di altri processi. L'invio avviene in una serie di cicli in cui ogni processo raccoglie o distribuisce da/verso tutti gli altri processi a cui si alternano le operazioni parallele di allocazione e calcolo della gravità.

1.2.4 Primo passo dell'algoritmo velocity Verlet - t_{p2}

A questo punto inizia il ciclo nel tempo per l'evoluzione del sistema. L'algoritmo velocity Verlet è separato in due diverse sezioni in quanto nel mezzo è necessario ricalcolare la gravità (e gli octree locali). Questa sezione è completamente parallela.

1.2.5 Scambio corpi - t_{p3}, t_{c4}

Prima di aggiornare gli octree è necessario affrontare le conseguenze degli spostamenti dei corpi nella sezione precedente. Una parte di essi infatti sarà migrata dal volume di competenza di un processo a quella di un altro. Questo è eseguito tramite un ciclo di invii a cerchio tra tutti i processi dove viene prima valutato il vettore di corpi proprio dividendo tra oggetti da tenere, scartare perchè usciti dal sistema in esame o passare ad un altro processo e poi man mano valutati i vettori dei corpi passati dai processi. Alla fine del cerchio tutti i corpi saranno stati inviati al processo corretto senza rischiare un deadlock.

1.2.6 Secondo passo dell'algoritmo velocity Verlet - t_{p4}

A questo punto viene completato il calcolo con l'algoritmo in questa seconda sezione completmente parallela.

1.2.7 Esporto i risultati su file - t_{s3}, t_{c5}

Dopo aver concluso il ciclo sul tempo si torna ad una sezione sequenziale dove il processo zero raccoglie i corpi da tutti gli altri e li stampa su file.

1.2.8 Difetti del codice

Il codice presenta due problematiche note.
La prima riguarda il load balancing. La struttura ad octree dell'algoritmo divide in maniera equa i

volumi di competenza ai vari processi ma il carico del calcolo è funzione principalmente del numero dei corpi. Questo vuol dire che per avere un load balancing ottimale il dataset di partenza dovrà essere il più possibile uniforme senza perdere troppo questa caratteristica durante l'evoluzione temporale. Questa richiesta è parzialmente in conflitto con la disomogeneità ideale per la condizione di clustering anche se non sono esattamente sulla stessa scala. Ho quindi una dipendenza del load balancing dal dataset soprattutto con pochi corpi coinvolti.
La seconda problematica riguarda la divisione del sistema. Fino ad otto processi con un solo livello di divisione al momento del calcolo della gravità non incontro problemi. Invece, dai sedici processi in su, accedo ai centri di massa del secondo livello dell'octree complessivo del sistema (64 centri di massa) perdendo l'informazione sul primo. Questo non invalida il conto ma riduce l'efficacia dell'approssimazione con condizione di clustering. La struttura dei comunicatori usati e la mancanza di un processo master non permette una soluzione semplice e si è preferito non aggiungere ulteriori complesse comunicazioni al codice.

2 Sperimentazione

2.1 Condizioni sperimentali

2.1.1 Dataset sintetico

Per testare il programma si è creato un dataset sintetico composto da sei corpi massivi in orbite stabili intorno all'origine e una nuvola di corpi minori centrale generati randomicamente. Il sistema così composto non è ovviamente del tutto stabile e quindi bisogna limitare la lunghezza delle simulazioni per evitare perdite di corpi.

2.1.2 Hardware impiegato

Le simulazioni i cui dati sono presentati in questa sezione sono state eseguite su di una serie di nodi computazionali Lenovo NeXtScale nx360 M5 ognuno dotato di 125 GiB di RAM DDR4 e due processori Intel Xeon E5-2697 v4. Tali CPU sono dotate di 18 core e gestiscono 36 thread. La loro frequenza di clock base è di 2.3 GHz con la possibilità di raggiungere un massimo di 3.6 GHz e possiedono 45 MB di cache il cui ultimo livello è dinamicamente condiviso tra i core (Intel Smart Cache).

2.2 Scalabilità

La prima analisi effettuata è stata uno studio del fattore di scalabilità. La simulazione considerata prevede 2000 corpi su un tempo di 2 unità temporali $\left(\sqrt{\frac{UA^3}{10^{-3}M_\odot}}\right)$ con $dt = 0.001$.

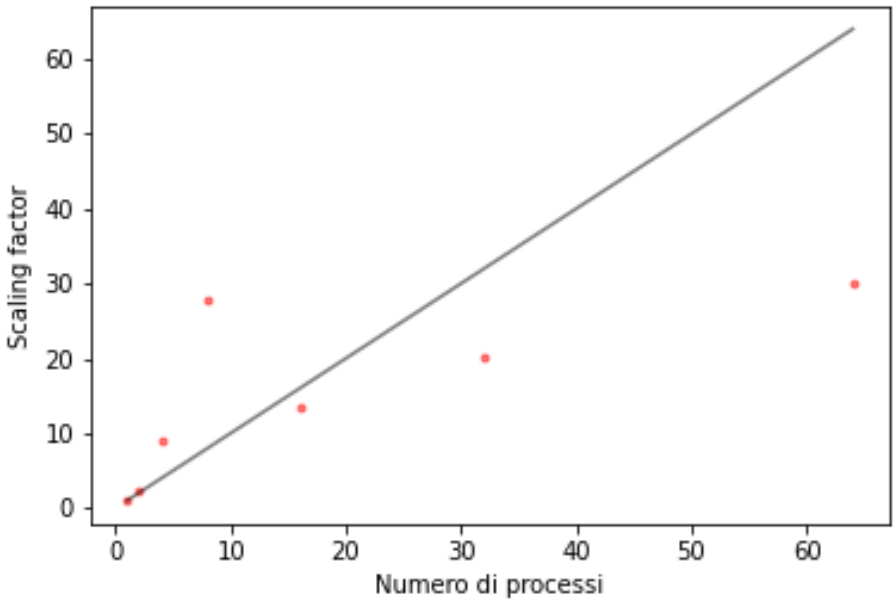


Figura 1: Scaling factor ($S(p) = t(1)/t(p)$ con p=numero di processori)

L'andamento dello scaling factor, esclusi i punti per quattro e otto, segue una curva che inizialmente è vicina alla retta che rappresenta lo scaling factor ideale per poi discostarsene tendendo ad un asintoto orizzontale. Questo andamento è atteso ed è dovuto al fatto che all'aumentare del numero di processi il tempo di calcolo sarà dominato dalla parte non parallelizzata del codice, come ad esempio input e output.
Per quattro e otto processi invece l'andamento si discosta molto da questo modello. Una discontinuità tra otto e sedici processi era attesa in quanto vi è un passaggio di divisione aggiuntiva in sottoproblemi, ma la superlinearità è inaspettata. Non sembra dipendere dal numero di dati in quanto si ripropone anche provando con 8000 corpi. Misurando i tempi della specifica sezione di codice che costruisce l'octree locale (oggetto allocato dinamicamente e quindi in una memoria condivisa) si osserva un rapido calo del tempo necessario fino ad una stabilizzazione intorno agli otto processi in linea con il comportamento del tempo complessivo. Questa è probabilmente la sezione di codice responsabile delle discontinuità osservate.

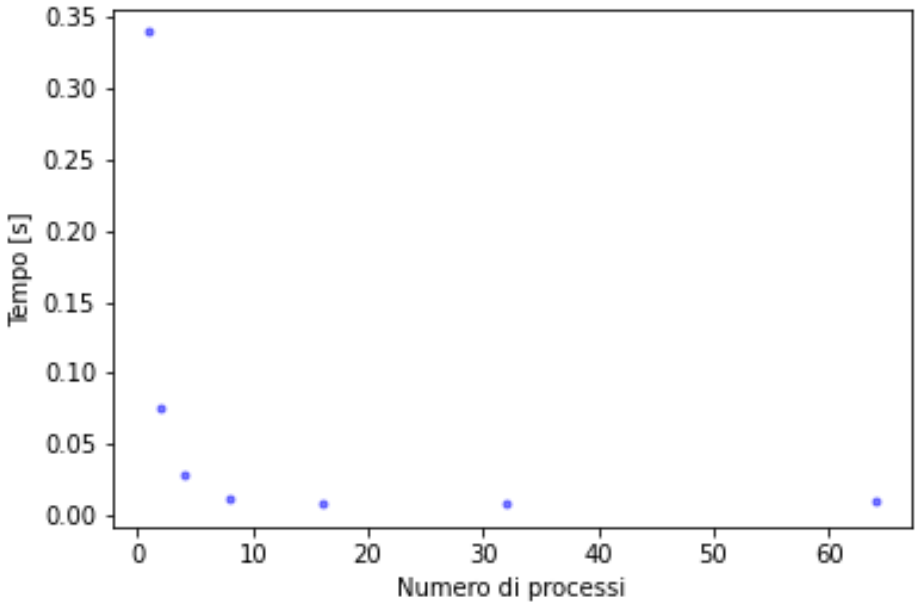


Figura 2: Tempo di calcolo sulla prima sezione di costruzione dell'octree e computazione della gravità in funzione del numero di processi

E' possibile che l'architettura hardware su cui sono state eseguite le simulazioni favorisca questo numero di processi per la presenza di banchi di memoria più rapidamente accessibili ad alcuni core o processori o cache dimensionate in modo tale da rendere più efficienti queste configurazioni. Si è quindi proceduto ad uno studio sulla località per approfondire la questione.

2.3 Studio sulla località

La velocità delle comunicazioni tra i processi e degli accessi in memoria è dipendente dalla struttura hardware. Il centro di calcolo sul quale sono state eseguite le run del software possiede 68 nodi da 36 core (distribuiti su due processori da 18 core ciascuno).
Per valutare l'effetto della località delle operazioni eseguite si sono effettuate due prove. Nella prima, limitandosi a 32 processi, si è forzato il sistema ad eseguire tutti i calcoli su di un unico nodo mentre nella seconda lo si è forzato ad eseguirlo su di un numero di nodi pari al numero di processi (per simmetria ci si è limitati anche qui a 32 processi).

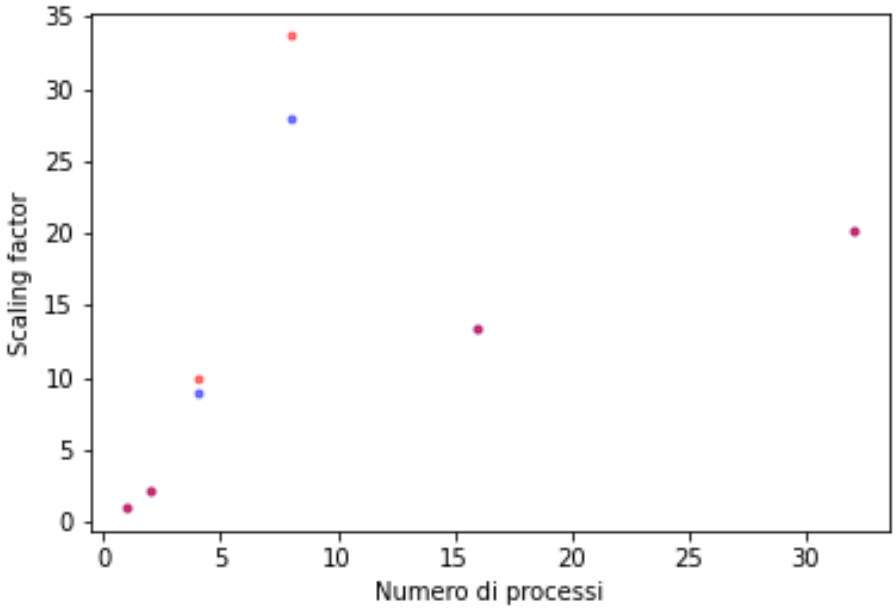


Figura 3: Scaling factor per due diversi casi: in blu per il programma eseguito su di un singolo nodo mentre in rosso se eseguito su p nodi differenti

Come si può vedere dall'immagine non ci sono sostanziali differenze tra i due casi, soprattutto per quanto riguarda i punti che non mostravano comportamenti anormali. I punti problematici, a differenza di quanto si prevedeva, mostrano un ulteriore aumento di scaling factor nel caso di esecuzione su diversi nodi e quindi fisicamente su diversi processori. Questo sembrerebbe escludere l'ipotesi di accelerazione data dalla memoria condivisa su un sottoinsieme di core di una singola macchina.

2.4 Stime sulla legge di Amdahl

La legge di Amdahl permette una stima dello speed-up di un codice in funzione del parmetro f indicante la percentuale di codice non parallelizzata.

$$S(p) = \frac{t_s}{f t_s + (1-f) t_s(p)} \tag{6}$$

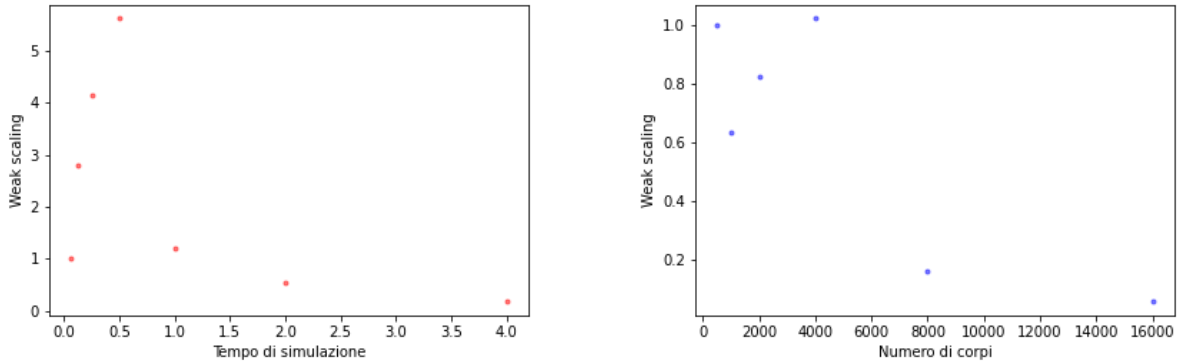
Per misurare lo speed-up servirebbe il miglior algoritmo sequenziale disponibile ma con alcune misure si può valutare, almeno parzialmente, il valore di f per questo codice. Si è presa in considerazione la run con un processo della sezione 2.2 e le componenti descritte nella sezione 1.2 (assumendo trascurabili i tempi di comunicazione visto il caso con un solo processo).

Componente	Tempo (s)
t_{s1}	0.0133631
t_{s2}	0.00107666
t_{s3}	0.00932205
t_{tot}	690.016

Dai dati in tabella si ottiene un $f \simeq 0.0034\%$ ovvero una percentuale trascurabile di codice non è parallelizzata su una simulazione da t=0 a t=2. La diminuzione di scaling factor osservata è quindi da imputarsi ai tempi delle numerose comunicazioni e sincronizzazioni necessarie.

2.5 Weak Scaling

Come ultimo passaggio si è andati ad analizzare la capacità di weak scaling del codice. Ciò che si vorrebbe avere è la capacità di mantenere costante il tempo di calcolo se a fronte ad un raddoppio dei dati si raddoppia il numero dei processi.
Ci sono due parametri che possiamo definire dati del problema che il codice risolve. Il primo è il numero degli oggetti coinvolti mentre il secondo è l'intervallo temporale della simulazione. Si sono quindi effettuati due studi raddoppiando questi parametri e il numero di processi impiegati. La grossa differenza tra questi raddoppi è il fatto che duplicare il numero dei corpi influenza i tempi sequenziali aumentando il parametro f, mentre aumentare la lunghezza della simulazione influenza solo le parti di codice parallele.



(a) *Raddoppio dei tempi*

(b) *Raddoppio del numero di oggetti*

Figura 4: Andamento del weak scaling al raddoppio del tempo di simulazione o del numero di oggetti coinvolti

Si ritrova nuovamente in entrambi i grafici la superlinearità su quattro e otto processi alla quale si aggiunge un più marcato boost di velocità con due processi.
A livello temporale il weak scaling non è troppo basso anche se non ottimale. Risente anche del fatto che il limite a t=4 non è arbitrario ma è tarato nella zona in cui questo sistema sintetico inizia a diventare molto instabile.
Il numero di corpi in input è invece un fattore che influenza molto i tempi di calcolo e aumentare il numero di processi non basta a controbilanciare l'aumento del peso delle parti sequenziali e delle comunicazioni necessarie (anch'esse legate al numero dei dati coinvolti). Il weak scaling in questo caso è abbastanza basso da controbilanciare il boost di cui gode la zona tra due e otto processi. Un altro fattore che penalizza i casi con più di otto processi all'aumentare dei corpi è la loro distribuzione in questo dataset sintetico. Infatti la nuvola di corpi è inizialmente condensata al centro e, mentre ciò non disturba il load balancing di una divisione in ottanti (che mantiene la simmetria centrale), lo peggiora quando la divisione è in sessantaquattro nodi diminuendo l'effettiva parallelizzazione del calcolo nella fase iniziale.

3 Conclusioni

Il codice è in grado di migliorare l'efficienza del calcolo tramite parallelizzazione anche se le sincronizzazioni necessarie tra le sue parti sono una considerevole zavorra per la scalabilità.
Si è osservato un comportamento di superlinearità in alcuni casi, in particolare per quattro e otto processi, probabilmente frutto della struttura hardware ma non del tutto compresa.

4 Appendice: Codice usato

4.1 main_mpi_v1.0.cpp

```
#include "mpi.h"
#include <string.h>
#include <stdlib.h>
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cmath>
#include <vector>
#include <math.h>

using namespace std;

//-------------------------------------------------------------------------+
//|                    INTERFACCIA: VARIABILI DA MODIFICARE                 |
//+-------------------------------------------------------------------------+
#define L          100   //Dimensione dello spazio dell'octree
#define T          2  //Tempo totale di simulazione

//-------------------------------------------------------------------------+
//|                                STRUCT BODY                              |
//+-------------------------------------------------------------------------+
struct Body{
        double mass;
  double coord[3];
  double vel[3];

  Body(){};
  Body(double m, double v[3], double x[3]){
    mass = m;
    for(int i=0;i<3;i++){
      coord[i] = x[i];
      vel[i] = v[i];
    }
  };
  Body(double m, double x[3]){
    mass = m;
    for(int i=0;i<3;i++){
      coord[i] = x[i];
      vel[i] = 0;
    }
  };
  bool IsInCube(double c[], double l){
    bool in = false;
    double x_min = min(c[0]+l/2,c[0]-l/2);
```

```
45      double x_max = max(c[0]+l/2,c[0]-l/2);
46      double y_min = min(c[1]+l/2,c[1]-l/2);
47      double y_max = max(c[1]+l/2,c[1]-l/2);
48      double z_min = min(c[2]+l/2,c[2]-l/2);
49      double z_max = max(c[2]+l/2,c[2]-l/2);
50      if(coord[0]>=x_min && coord[0]<x_max &&
51         coord[1]>=y_min && coord[1]<y_max &&
52         coord[2]>=z_min && coord[2]<z_max){
53          in = true;
54      }
55      return in;
56    };
57    bool IsInCubes(double c[], double l, int n){
58      int a = 0;
59      for(int i=0;i<n;i++){
60        if(IsInCube(&c[3*i], l)==true) a++;
61      }
62      if(a==0){
63        return false;
64      }
65      else{
66        return true;
67      }
68    };
69    void Print(){
70      cout << "m = " << mass << "\n";
71      cout << "x = " << coord[0] << "," << coord[1] << "," << coord[2] << "\n";
72      cout << "v = " << vel[0] << "," << vel[1] << "," << vel[2] << "\n";
73    }
74  };
75
76  //-------------------------------------------------------------------------+
77  //|                              STRUCT VECT3D                              |
78  //+-------------------------------------------------------------------------+
79  struct Vect3D{
80    double v[3];
81  };
82
83  //-------------------------------------------------------------------------+
84  //|                              CLASSE BARNES                              |
85  //+-------------------------------------------------------------------------+
86  class Barnes{
87      public:
88          Barnes(){
89            empty = false;
90          }
91          Barnes(double m, double *x){
92            cm.mass=m;
```

```
93            cm.coord[0]=x[0];
94            cm.coord[1]=x[1];
95            cm.coord[2]=x[2];
96            cm.vel[0]=0.;
97            cm.vel[1]=0.;
98            cm.vel[2]=0.;
99            empty = false;
100         }
101         Barnes(Body &b){
102           cm = b;
103           empty = false;
104         }
105         Barnes(int &N, double c[], double Len, Body bodies[]){
106           //creo il nodo base
107           double m = 0;
108           double mm;
109           double p[3] = {0,0,0};  //per il centro di massa
110           int nb = 0;
111           int lf;
112           for(int i=0;i<N;i++){
113             if(bodies[i].IsInCube(c,Len)==true){
114               nb++;
115               if(nb==1){
116                 lf=i;
117               }
118               mm = bodies[i].mass;
119               m += mm;
120               p[0] += bodies[i].coord[0]*mm;
121               p[1] += bodies[i].coord[1]*mm;
122               p[2] += bodies[i].coord[2]*mm;
123             }
124           }
125           if(nb==0){
126             empty = true;
127           }
128           if(nb>0){  //lo considero solo se contiene corpi
129             p[0]=p[0]/m;
130             p[1]=p[1]/m;
131             p[2]=p[2]/m;
132             if (nb==1){
133               cm = bodies[lf];
134               empty = false;
135             }
136             else{
137               cm.mass = m;
138               cm.coord[0]=p[0];
139               cm.coord[1]=p[1];
140               cm.coord[2]=p[2];
```

```
141             empty = false;
142           }
143         }
144         if(nb>1){
145           //divido in ottanti e chiamo il costruttore
146           double l = Len/2;
147           //salvo il centro madre
148           double cc[3] = {c[0],c[1],c[2]};
149           //primo ottante
150           c[0] = cc[0] - l/2;
151           c[1] = cc[1] - l/2;
152           c[2] = cc[2] - l/2;
153           Barnes* nd1 = new Barnes(N,c,l,bodies);
154           if(nd1->empty==true){
155             delete nd1;
156           }
157           else{
158             subnodes.push_back(nd1);
159           }
160           //secondo ottante
161           c[2] = cc[2] + l/2;
162           Barnes* nd2 = new Barnes(N,c,l,bodies);
163           if(nd2->empty==true){
164             delete nd2;
165           }
166           else{
167             subnodes.push_back(nd2);
168           }
169           //terzo ottante
170           c[1] = cc[1] + l/2;
171           c[2] = cc[2] - l/2;
172           Barnes* nd3 = new Barnes(N,c,l,bodies);
173           if(nd3->empty==true){
174             delete nd3;
175           }
176           else{
177             subnodes.push_back(nd3);
178           }
179           //quarto ottante
180           c[2] = cc[2] + l/2;
181           Barnes* nd4 = new Barnes(N,c,l,bodies);
182           if(nd4->empty==true){
183             delete nd4;
184           }
185           else{
186             subnodes.push_back(nd4);
187           }
188           //quinto ottante
```

```
189             c[0] = cc[0] + l/2;
190             c[1] = cc[1] - l/2;
191             c[2] = cc[2] - l/2;
192             Barnes* nd5 = new Barnes(N,c,l,bodies);
193             if(nd5->empty==true){
194               delete nd5;
195             }
196             else{
197               subnodes.push_back(nd5);
198             }
199             //sesto ottante
200             c[2] = cc[2] + l/2;
201             Barnes* nd6 = new Barnes(N,c,l,bodies);
202             if(nd6->empty==true){
203               delete nd6;
204             }
205             else{
206               subnodes.push_back(nd6);
207             }
208             //settimo ottante
209             c[1] = cc[1] + l/2;
210             c[2] = cc[2] - l/2;
211             Barnes* nd7 = new Barnes(N,c,l,bodies);
212             if(nd7->empty==true){
213               delete nd7;
214             }
215             else{
216               subnodes.push_back(nd7);
217             }
218             //ottavo ottante
219             c[2] = cc[2] + l/2;
220             Barnes* nd8 = new Barnes(N,c,l,bodies);
221             if(nd8->empty==true){
222               delete nd8;
223             }
224             else{
225               subnodes.push_back(nd8);
226             }
227           }
228         }
229         ~Barnes(){
230
231         }
232         void Delete_Subnodes(){
233           if(subnodes.size()>0){
234             for(int i=0;i<subnodes.size();i++){
235               subnodes[i]->Delete_Subnodes();
236               delete subnodes[i];
```

```
237             }
238           }
239         }
240
241         Body cm;
242         vector<Barnes*> subnodes; //sottonodi dell'octree
243         bool empty;
244 };
245
246 //-------------------------------------------------------------------------+
247 //|                          DICHIARAZIONI FUNZIONI                         |
248 //+-------------------------------------------------------------------------+
249 //funzione per sapere quanti corpi sono descritti dal file
250 int count(string file);
251 //funzione per leggere il file con un array di corpi
252 void load(string file, Body bodies[], int dimension);
253 //funzione per salvare su file i valori di un array di corpi
254 void save(string file, Body bodies[], int dimension);
255 //funzione per costruire la parte di octree locale di ogni processo
256 Barnes* Generate_Octree(int &N, double c[], double Len, Body bodies[], int myrank);
257 //funzione per gestire l'allocazione dinamica dell'octree locale
258 void Delete_Octree(Barnes* root);
259 //funzioni Distance per il calcolo di distanze tra Body
260 double Distance(Body b1, Body b2);
261 void Vectorial_Distance(Body b1, Body b2, double *d);
262 //funzione per l'equazione di Newton
263 void acc(double m, double d, double *x, double *a);
264 //funzioni per il calcolo della gravità
265 void Compute_Gravity(Body body, Barnes* current, double *g, double Len);
266 void Compute_Gravity_cm(Body body, int i, Body nodes_cm[], int Ncm, int Np, int rank, i
267
268 //-------------------------------------------------------------------------+
269 //|                                  MAIN                                   |
270 //+-------------------------------------------------------------------------+
271 int main(int argc, char **argv){
272
273 //Inizializzo MPI
274   MPI_Init(&argc, &argv);
275   double t0,t1,t2,t3,t4,t5;
276   t0 = MPI_Wtime();
277   int myrank[2];
278   int Np[2];
279   MPI_Comm COMM_LVL1;
280   MPI_Comm COMM_LVL2;
281   MPI_Comm_rank(MPI_COMM_WORLD, myrank);
282   MPI_Comm_size(MPI_COMM_WORLD, Np);
283   int color;
284   int color1;
```

```
285   if(myrank[0]<8){
286     color = myrank[0]%8;
287     color1 = 1;
288   }
289   if(myrank[0]>=8){
290     color = (myrank[0]-8)/((Np[0]-8)/8);
291     color1 = 0;
292   }
293   MPI_Comm_split(MPI_COMM_WORLD, color1, myrank[0], &COMM_LVL1);
294   MPI_Comm_split(MPI_COMM_WORLD, color, myrank[0], &COMM_LVL2);
295   MPI_Comm_rank(COMM_LVL2, &myrank[1]);
296   MPI_Comm_size(COMM_LVL2, &Np[1]);
297
298
299 //Dichiarazione variabili necessarie
300   double   t     = 0.;
301   double   dt    = 0.001; //salto temporale
302   string input_file = "nbody_start.txt";
303   string output_file = "nbody_end.txt";
304   int N;
305   MPI_Status status;
306   double cc[3] = {0,0,0}; //centro del sistema
307   double ccc[3] = {0,0,0};
308   double l;
309   int Npc;
310   if(Np[0]<=8){
311     Npc=Np[0];
312   }
313   else{
314     Npc=Np[1];
315   }
316
317 //definizione tipo per mandare le struct Body e vect3D
318   MPI_Datatype Bodytype;
319   MPI_Type_contiguous(7,MPI_DOUBLE,&Bodytype);
320   MPI_Type_commit(&Bodytype);
321   MPI_Aint extent,lb;
322   MPI_Type_get_extent(Bodytype,&lb, &extent);
323
324   MPI_Datatype type3D;
325   MPI_Type_contiguous(3,MPI_DOUBLE,&type3D);
326   MPI_Type_commit(&type3D);
327   MPI_Aint extent1,lb1;
328   MPI_Type_get_extent(type3D,&lb1, &extent1);
329
330 //+-------------------------------------------------------------------------+
331 //|                      IMPORTO I CORPI DEL SISTEMA                        |
332 //+-------------------------------------------------------------------------+
```

```
333   if (myrank[0]==0) {
334     N = count(input_file);
335     cout << "Dati contati: " << N << "\n";
336   }
337   MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD); //serve per allocare vettori abbastanza
338
339   Body * bodies = NULL;
340   bodies = new Body[N];
341   Body * bodies_end = NULL;
342   bodies_end = new Body[N];
343   Body * octree = NULL;
344   octree = new Body[8*N];
345   Barnes* local_octree[8/Npc];
346   Body cm1[8/Npc];
347   double g[N][3];
348   Body *cm = NULL;
349   int Ncm;
350   if(Np[0]<=8){
351     Ncm=8;
352   }
353   if(Np[0]>8){
354     Ncm=64;
355   }
356   cm = new Body[Ncm];
357   double *ctr = NULL;
358   ctr = new double[3*(8/Npc)];
359   Body send[N];
360   Body newbodies[N];
361   int n_recv;
362   int n[8];
363   int n_end[Np[0]];
364   int shift[Np[0]];
365   int a;
366   if (myrank[0]==0) {
367     load(input_file,bodies,N);
368   }
369   t1 = MPI_Wtime();
370
371 //-------------------------------------------------------------------------+
372 //|                          DIVISIONE DEL PROBLEMA                         |
373 //+-------------------------------------------------------------------------+
374
375 int lvl=0;
376 if(Np[0]>8){
377   lvl=1;
378 }
379 l=L;
380 for(int k=0;k<=lvl;k++){
```

```
381     int Npp;
382     if(Np[k]<=8){
383       Npp=Np[k];
384     }
385     else{
386       Npp=8;
387     }
388     if (myrank[k]==0){
389       //divido in ottanti
390       l = l/2;
391       //posizioni dei centri
392       if(k>0){
393         ccc[0]=ctr[0];
394         ccc[1]=ctr[1];
395         ccc[2]=ctr[2];
396       }
397       double c[8][3] = {{ccc[0] - l/2,ccc[1] - l/2,ccc[2] - l/2},
398                         {ccc[0] - l/2,ccc[1] - l/2,ccc[2] + l/2},
399                         {ccc[0] - l/2,ccc[1] + l/2,ccc[2] - l/2},
400                         {ccc[0] - l/2,ccc[1] + l/2,ccc[2] + l/2},
401                         {ccc[0] + l/2,ccc[1] - l/2,ccc[2] - l/2},
402                         {ccc[0] + l/2,ccc[1] - l/2,ccc[2] + l/2},
403                         {ccc[0] + l/2,ccc[1] + l/2,ccc[2] - l/2},
404                         {ccc[0] + l/2,ccc[1] + l/2,ccc[2] + l/2}};
405       //spedisco ai processi fratelli i centri di loro competenza
406       if(k==0){
407         MPI_Scatter(&c, 3*(8/Npp), MPI_DOUBLE, ctr, 3*(8/Npp), MPI_DOUBLE, 0, COMM_LVL1);
408       }
409       if(k==1){
410         MPI_Scatter(&c, 3*(8/Npp), MPI_DOUBLE, ctr, 3*(8/Npp), MPI_DOUBLE, 0, COMM_LVL2);
411       }
412       vector<int> o[8];
413       for(int j=0;j<8;j++){
414         n[j]=0;
415       }
416       for(int i=0;i<N;i++){
417         for(int j=0;j<8;j++){
418           if(bodies[i].IsInCube(c[j],l)==true){
419             o[j].push_back(i);
420             n[j]++;
421           }
422         }
423       }
424       int aa = 0;
425       for(int j=0;j<8;j++){
426         for(int y=0;y<o[j].size();y++){
427           octree[aa+y]=bodies[o[j][y]];
428         }
```

```
429         aa+=o[j].size();
430       }
431     }
432     //ricevo nei processi fratelli i centri di loro competenza
433     if(myrank[k]>0 && myrank[k]<8){
434       l = L/2;
435       double c; //serve solo per il compilatore
436       if(k==0){
437         MPI_Scatter(&c, 3*(8/Npp), MPI_DOUBLE, ctr, 3*(8/Npp), MPI_DOUBLE, 0, COMM_LVL1);
438       }
439       if(k==1){
440         MPI_Scatter(&c, 3*(8/Npp), MPI_DOUBLE, ctr, 3*(8/Npp), MPI_DOUBLE, 0, COMM_LVL2);
441       }
442     }
443     //condivido la visione globale
444     if(k==0 && myrank[0]<8){
445       MPI_Bcast(&n, 8, MPI_INT, 0, COMM_LVL1);
446     }
447     if(k==1){
448       MPI_Bcast(&n, 8, MPI_INT, 0, COMM_LVL2);
449     }
450     for(int j=0;j<Npp;j++){
451       n[j]=n[(j*(8/Npp))];
452       for(int i=1;i<(8/Npp);i++){
453         n[j]+=n[(j*(8/Npp))+i];
454       }
455     }
456     N=n[myrank[k]];
457     shift[0]=0;
458     for(int i=1;i<Npp;i++){
459       shift[i]=shift[i-1]+n[i-1];
460     }
461     if(k==0 && myrank[0]<8){
462       MPI_Scatterv(octree,n,shift,Bodytype,bodies,N,Bodytype,0,COMM_LVL1);
463     }
464     if(k==1){
465       MPI_Scatterv(octree,n,shift,Bodytype,bodies,N,Bodytype,0,COMM_LVL2);
466     }
467   }
468   t2 = MPI_Wtime();
469
470   //-------------------------------------------------------------------------+
471   //|                            OCTREE E GRAVITA'                            |
472   //+-------------------------------------------------------------------------+
473
474     for(int i=0;i<(8/Npc);i++){
475       local_octree[i] = Generate_Octree(N, &ctr[3*i], l, bodies, myrank[0]);
476       cm1[i] = local_octree[i]->cm;
```

```
477     }
478     MPI_Allgather(&cm1, 8/Npc, Bodytype, cm, 8/Npc, Bodytype, MPI_COMM_WORLD);
479
480     vector<int> to[Np[0]];
481     int rrank;
482     if(Np[0]<=8){
483       rrank=myrank[0];
484     }
485     if(Np[0]>8){
486       rrank=myrank[1];
487     }
488
489     for(int i=0;i<N;i++){
490       //inizializzo gravità
491       g[i][0] = 0;
492       g[i][1] = 0;
493       g[i][2] = 0;
494       for(int j=0;j<(8/Npc);j++){
495         Compute_Gravity(bodies[i], local_octree[j], g[i], l);
496       }
497       Compute_Gravity_cm(bodies[i], i, cm, Ncm,  Np[1], rrank, myrank[0], to, g[i], l);
498     }
499
500     //Computazione gravità negli altri processi
501     int n_rec[Np[0]];
502     //invio dimensioni
503     for(int i=0;i<Np[0];i++){
504       int sn = to[i].size(); //Numero di invii
505       MPI_Gather(&sn,1,MPI_INT,&n_rec,1,MPI_INT,i,MPI_COMM_WORLD);
506     }
507     //operazioni parallele
508     int tot=0;
509     for(int j=0;j<Np[0];j++){
510       tot+=n_rec[j];
511       if(j==0){
512         shift[j]=0;
513       }
514       else{
515         shift[j]=shift[j-1]+n_rec[j-1];
516       }
517     }
518     Body in[tot];
519     Vect3D gy[tot];
520     //invio corpi
521     for(int i=0;i<Np[0];i++){
522       int sn = to[i].size(); //Numero di invii
523       Body out[sn];  //cosa invio
524       for(int j=0;j<sn;j++){
```

```
525         out[j]=bodies[to[i][j]];
526       }
527       MPI_Gatherv(&out,sn,Bodytype,&in,n_rec,shift,Bodytype,i,MPI_COMM_WORLD);
528     }
529     //Computo gravità
530     for(int j=0;j<tot;j++){
531       //inizializzo gravità
532       gy[j].v[0] = 0;
533       gy[j].v[1] = 0;
534       gy[j].v[2] = 0;
535       for(int k=0;k<(8/Npc);k++){
536         Compute_Gravity(in[j], local_octree[k], gy[j].v, 1);
537       }
538     }
539     for(int i=0;i<Np[0];i++){
540       int sn = to[i].size(); //Numero di invii
541       Vect3D gg[sn];
542       MPI_Scatterv(&gy,n_rec,shift,type3D,&gg,sn,type3D,i,MPI_COMM_WORLD);
543       for(int j=0;j<sn;j++){
544         g[to[i][j]][0]+=gg[j].v[0];
545         g[to[i][j]][1]+=gg[j].v[1];
546         g[to[i][j]][2]+=gg[j].v[2];
547       }
548     }
549
550     for(int i=0;i<(8/Npc);i++){
551       Delete_Octree(local_octree[i]);
552     }
553   t3 = MPI_Wtime();
554   //------------------------------------------------------------------------+
555   //|                              CICLO SUL TEMPO                           |
556   //+------------------------------------------------------------------------+
557
558     while(t<=T){
559       if(myrank[0]==0){
560         cout << "ciclo al tempo " << t << endl;
561       }
562
563   //------------------------------------------------------------------------+
564   //|                       VELOCITY VERLET PRIMA PARTE                      |
565   //+------------------------------------------------------------------------+
566       for(int i=0;i<N;i++){
567         bodies[i].vel[0] = bodies[i].vel[0] + 0.5*dt*g[i][0];
568         bodies[i].vel[1] = bodies[i].vel[1] + 0.5*dt*g[i][1];
569         bodies[i].vel[2] = bodies[i].vel[2] + 0.5*dt*g[i][2];
570         bodies[i].coord[0] = bodies[i].coord[0] + dt*bodies[i].vel[0];
571         bodies[i].coord[1] = bodies[i].coord[1] + dt*bodies[i].vel[1];
572         bodies[i].coord[2] = bodies[i].coord[2] + dt*bodies[i].vel[2];
```

```
573     }
574
575 //-------------------------------------------------------------------------+
576 ///|                                SCAMBIO CORPI                          |
577 //+-------------------------------------------------------------------------+
578     int n_send;
579     vector<int> s;
580     vector<int> h;
581     //inizializzo newbodies e n_recv sullo stato attuale
582     for(int i=0;i<N;i++){
583       newbodies[i]=bodies[i];
584     }
585     n_recv = N;
586     N = 0; //lo ricalcolo nello scambio
587     for(int j=0;j<Np[0];j++){
588       for(int i=0;i<n_recv;i++){
589         if(newbodies[i].IsInCube(cc,L)==false){
590            //bodies fuori dal sistema, non li considero oltre
591         }
592         else if(newbodies[i].IsInCubes(ctr, l, 8/Npc)==false){
593           s.push_back(i); //salvo l'indice dei bodies fuori dal sottosistema da inviar
594         }
595         else{
596           h.push_back(i); //salvo l'indice dei bodies da tenere
597         }
598       }
599       for(int i=0;i<s.size();i++){
600         send[i]= newbodies[s[i]];
601       }
602       for(int i=0;i<h.size();i++){
603         bodies[N]= newbodies[h[i]];
604         N++;
605       }
606       if(j<(Np[0]-1)){
607         //spedizioni di scambio a cerchio
608         n_send=s.size();
609         int id_s = myrank[0]+1;
610         int id_r = myrank[0]-1;
611         if(myrank[0]==(Np[0]-1)){
612           id_s = 0;
613         }
614         if(myrank[0]==0){
615           id_r = Np[0]-1;
616         }
617         if(myrank[0]%2==0){
618           MPI_Send(&n_send, 1, MPI_INT, id_s, j+1, MPI_COMM_WORLD);
619           MPI_Recv(&n_recv, 1, MPI_INT, id_r, j+1, MPI_COMM_WORLD, &status);
620
```

```
621           MPI_Send(&send, n_send, Bodytype, id_s, j+8, MPI_COMM_WORLD);
622           MPI_Recv(&newbodies, n_recv, Bodytype, id_r, j+8, MPI_COMM_WORLD, &status);
623         }
624         else{
625           MPI_Recv(&n_recv, 1, MPI_INT, id_r, j+1, MPI_COMM_WORLD, &status);
626           MPI_Send(&n_send, 1, MPI_INT, id_s, j+1, MPI_COMM_WORLD);
627
628           MPI_Recv(&newbodies, n_recv, Bodytype, id_r, j+8, MPI_COMM_WORLD, &status);
629           MPI_Send(&send, n_send, Bodytype, id_s, j+8, MPI_COMM_WORLD);
630         }
631       }
632       s.clear();
633       h.clear();
634     }
635
636 //-------------------------------------------------------------------------+
637 //|                              OCTREE E GRAVITA'                          |
638 //+-------------------------------------------------------------------------+
639
640   //aggiorno l'octree e la gravità
641   for(int i=0;i<(8/Npc);i++){
642     local_octree[i] = Generate_Octree(N, &ctr[3*i], l, bodies, myrank[0]);
643     cm1[i] = local_octree[i]->cm;
644   }
645   MPI_Allgather(&cm1, 8/Npc, Bodytype, cm, 8/Npc, Bodytype, MPI_COMM_WORLD);
646
647   vector<int> to[Np[0]];
648   int rrank;
649   if(Np[0]<=8){
650     rrank=myrank[0];
651   }
652   if(Np[0]>8){
653     rrank=myrank[1];
654   }
655
656   for(int i=0;i<N;i++){
657     //inizializzo gravità
658     g[i][0] = 0;
659     g[i][1] = 0;
660     g[i][2] = 0;
661     for(int j=0;j<(8/Npc);j++){
662       Compute_Gravity(bodies[i], local_octree[j], g[i], l);
663     }
664     Compute_Gravity_cm(bodies[i], i, cm, Ncm,  Np[1], rrank, myrank[0], to, g[i], l);
665   }
666
667   //Computazione gravità negli altri processi
668   int n_rec[Np[0]];
```

```
669     //invio dimensioni
670     for(int i=0;i<Np[0];i++){
671       int sn = to[i].size(); //Numero di invii
672       MPI_Gather(&sn,1,MPI_INT,&n_rec,1,MPI_INT,i,MPI_COMM_WORLD);
673     }
674     //operazioni parallele
675     int tot=0;
676     for(int j=0;j<Np[0];j++){
677       tot+=n_rec[j];
678       if(j==0){
679         shift[j]=0;
680       }
681       else{
682         shift[j]=shift[j-1]+n_rec[j-1];
683       }
684     }
685     Body in[tot];
686     Vect3D gy[tot];
687     //invio corpi
688     for(int i=0;i<Np[0];i++){
689       int sn = to[i].size(); //Numero di invii
690       Body out[sn];  //cosa invio
691       for(int j=0;j<sn;j++){
692         out[j]=bodies[to[i][j]];
693       }
694       MPI_Gatherv(&out,sn,Bodytype,&in,n_rec,shift,Bodytype,i,MPI_COMM_WORLD);
695     }
696     //Computo gravità
697     for(int j=0;j<tot;j++){
698       //inizializzo gravità
699       gy[j].v[0] = 0;
700       gy[j].v[1] = 0;
701       gy[j].v[2] = 0;
702       for(int k=0;k<(8/Npc);k++){
703         Compute_Gravity(in[j], local_octree[k], gy[j].v, l);
704       }
705     }
706     //restituisco valori della gravità
707     for(int i=0;i<Np[0];i++){
708       int sn = to[i].size(); //Numero di invii
709       Vect3D gg[sn];
710       MPI_Scatterv(&gy,n_rec,shift,type3D,&gg,sn,type3D,i,MPI_COMM_WORLD);
711       for(int j=0;j<sn;j++){
712         g[to[i][j]][0]+=gg[j].v[0];
713         g[to[i][j]][1]+=gg[j].v[1];
714         g[to[i][j]][2]+=gg[j].v[2];
715       }
716     }
```

```
717
718   for(int i=0;i<(8/Npc);i++){
719     Delete_Octree(local_octree[i]);
720   }
721
722 //-------------------------------------------------------------------------+
723 //|                        VELOCITY VERLET SECONDA PARTE                    |
724 //+-------------------------------------------------------------------------+
725     //velocity Verlet parte 2
726     for(int i=0;i<N;i++){
727       bodies[i].vel[0] = bodies[i].vel[0] + 0.5*dt*g[i][0];
728       bodies[i].vel[1] = bodies[i].vel[1] + 0.5*dt*g[i][1];
729       bodies[i].vel[2] = bodies[i].vel[2] + 0.5*dt*g[i][2];
730     }
731     t += dt;
732   }
733   t4 = MPI_Wtime();
734
735 //-------------------------------------------------------------------------+
736 //|                          ESPORTO I RISULTATI                            |
737 //+-------------------------------------------------------------------------+
738   MPI_Gather(&N,1,MPI_INT,n_end,1,MPI_INT,0,MPI_COMM_WORLD); //raccolgo n dai vari proc
739   shift[0]=0;
740   for(int i=1;i<Np[0];i++){
741     shift[i]=shift[i-1]+n_end[i-1];
742   }
743   MPI_Gatherv(bodies,N,Bodytype,bodies_end,n_end,shift,Bodytype,0,MPI_COMM_WORLD);
744   if (myrank[0]==0) {
745     N=0;
746     for(int i=0;i<Np[0];i++){
747         N += n_end[i];
748     }
749     save(output_file,bodies_end,N);
750     cout << "Dati salvati: " << N << "\n";
751   }
752   delete octree;
753   delete cm;
754   delete ctr;
755   delete bodies_end;
756   delete bodies;
757   t5 = MPI_Wtime();
758   MPI_Finalize();
759   if(myrank[0]==0){
760     ofstream time_data;
761           time_data.open("Performance_"+to_string(Np[0])+".txt");
762     time_data << "Number of bodies: " << N << "\n";
763     time_data << "Total time (MPI) is " << t5-t0 << "\n";
764     time_data << "Import time (MPI) is " << t1-t0 << "\n";
```

```
765     time_data << "Division time (MPI) is " << t2-t1 << "\n";
766     time_data << "First Octree time (MPI) is " << t3-t2 << "\n";
767     time_data << "Export time (MPI) is " << t5-t4 << "\n";
768   }
769   return 0;
770 }
771
772 //--------------------------------------------------------------------------+
773 //|                                FUNZIONI                                  |
774 //+--------------------------------------------------------------------------+
775 int count(string file){
776         int c;
777         c = 0;
778         ifstream txt;
779         txt.open(file);
780         string s;
781         while(txt.eof() == false){
782                 getline(txt,s);
783                 if(s != "" && s.at(0) != '#'){
784                         c++;
785                 }
786         }
787         txt.close();
788 return c;
789 }
790
791 void load(string file, Body bodies[], int dimension){
792         ifstream txt;
793         txt.open(file);
794         string s;
795         string p[3];
796         string v[3];
797         string m;
798         int i;
799         i = 0;  //numero riga
800     int k;
801         k = 0;        //numero colonna
802         while(txt.eof() == false){
803                 getline(txt,s);
804                 if(s != "" && s.at(0) != '#'){
805                         if(i < dimension){
806                                 for(int j=0; j < s.length(); j++){
807                                         if(s[j] != ' ' && k==0){
808                                                 m=m+s[j];
809                                         }
810                                         else if(s[j] != ' ' && k>0 && k<4){
811                                                 p[k-1]=p[k-1]+s[j];
812                                         }
```

```
813                                     else if(s[j] != ' ' && k>3 && k<7){
814                                                     v[k-4]=v[k-4]+s[j];
815                                             }
816                                             else{
817                                       k++;
818                                             }
819                                     }
820                                     double x[3] = {stod(p[0]),stod(p[1]),stod(p[2])};
821                                     double vl[3] = {stod(v[0]),stod(v[1]),stod(v[2])};
822                                     Body bd(stof(m),vl,x);
823                                     bodies[i]=bd;
824                                     m = "";
825                                    p[0] = "";
826                                     p[1] = "";
827                             p[2] = "";
828                                     v[0] = "";
829                                     v[1] = "";
830                             v[2] = "";
831                                     i++;
832                             }
833                             else{
834                                     cout << "Errore: file oltre il limite del vettore" << en
835                                     cout << "Non tutti i dati sono stati raccolti" << endl;
836                                     break;
837                             }
838                             k = 0;
839                     }
840             }
841             txt.close();
842     }
843
844     void save(string file, Body bodies[], int dimension){
845             ofstream txt;
846             txt.open(file);
847             for(int i=0;i<dimension;i++){
848                     txt << setiosflags(ios::scientific);
849                     txt << bodies[i].mass << " ";
850                     txt << bodies[i].coord[0] << " ";
851                     txt << bodies[i].coord[1] << " ";
852                     txt << bodies[i].coord[2] << " ";
853                     txt << bodies[i].vel[0] << " ";
854                     txt << bodies[i].vel[1] << " ";
855                     txt << bodies[i].vel[2] << endl;
856             }
857             txt.close();
858     }
859
860     Barnes* Generate_Octree(int &N, double c[], double Len, Body bodies[], int myrank){
```

```
861       //creo il nodo root
862       double m = 0;
863       double mm;
864       double p[3] = {0,0,0};  //per il centro di massa
865       for(int i=0;i<N;i++){
866           mm = bodies[i].mass;
867           m += mm;
868           p[0] += bodies[i].coord[0]*mm;
869           p[1] += bodies[i].coord[1]*mm;
870           p[2] += bodies[i].coord[2]*mm;
871       }
872       p[0]=p[0]/m;
873       p[1]=p[1]/m;
874       p[2]=p[2]/m;
875       Barnes* root = new Barnes(m,p);
876       if(N>1){
877         //divido in ottanti e chiamo il costruttore
878         double l = Len/2;
879         //salvo il centro madre
880         double cc[3] = {c[0],c[1],c[2]};
881         //primo ottante
882         c[0] = cc[0] - l/2;
883         c[1] = cc[1] - l/2;
884         c[2] = cc[2] - l/2;
885         Barnes* nd1 = new Barnes(N,c,l,bodies);
886         if(nd1->empty==true){
887             delete nd1;
888         }
889         else{
890             root->subnodes.push_back(nd1);
891         }
892         //secondo ottante
893         c[2] = cc[2] + l/2;
894         Barnes* nd2 = new Barnes(N,c,l,bodies);
895         if(nd2->empty==true){
896             delete nd2;
897         }
898         else{
899             root->subnodes.push_back(nd2);
900         }
901         //terzo ottante
902         c[1] = cc[1] + l/2;
903         c[2] = cc[2] - l/2;
904         Barnes* nd3 = new Barnes(N,c,l,bodies);
905         if(nd3->empty==true){
906             delete nd3;
907         }
908         else{
```

```
909             root->subnodes.push_back(nd3);
910         }
911         //quarto ottante
912         c[2] = cc[2] + l/2;
913         Barnes* nd4 = new Barnes(N,c,l,bodies);
914         if(nd4->empty==true){
915             delete nd4;
916         }
917         else{
918             root->subnodes.push_back(nd4);
919         }
920         //quinto ottante
921         c[0] = cc[0] + l/2;
922         c[1] = cc[1] - l/2;
923         c[2] = cc[2] - l/2;
924         Barnes* nd5 = new Barnes(N,c,l,bodies);
925         if(nd5->empty==true){
926             delete nd5;
927         }
928         else{
929             root->subnodes.push_back(nd5);
930         }
931         //sesto ottante
932         c[2] = cc[2] + l/2;
933         Barnes* nd6 = new Barnes(N,c,l,bodies);
934         if(nd6->empty==true){
935             delete nd6;
936         }
937         else{
938             root->subnodes.push_back(nd6);
939         }
940         //settimo ottante
941         c[1] = cc[1] + l/2;
942         c[2] = cc[2] - l/2;
943         Barnes* nd7 = new Barnes(N,c,l,bodies);
944         if(nd7->empty==true){
945             delete nd7;
946         }
947         else{
948             root->subnodes.push_back(nd7);
949         }
950         //ottavo ottante
951         c[2] = cc[2] + l/2;
952         Barnes* nd8 = new Barnes(N,c,l,bodies);
953         if(nd8->empty==true){
954             delete nd8;
955         }
956         else{
```

```
957             root->subnodes.push_back(nd8);
958           }
959         //risistemo la variabile centro passata by reference
960           c[0] = cc[0];
961           c[1] = cc[1];
962           c[2] = cc[2];
963         }
964             return root;
965 }
966
967 void Delete_Octree(Barnes* root){ //gestione allocazione dinamica
968             root->Delete_Subnodes();
969             delete root;
970 }
971
972 double Distance(Body b1, Body b2){
973
974             double a[3] = {b1.coord[0],b1.coord[1],b1.coord[2]};
975             double b[3] = {b2.coord[0],b2.coord[1],b2.coord[2]};
976
977         double d = sqrt(abs(((a[0]-b[0])*(a[0]-b[0]))+
978                             ((a[1]-b[1])*(a[1]-b[1]))+
979                             ((a[2]-b[2])*(a[2]-b[2]))));
980
981         return d;
982 }
983
984 void Vectorial_Distance(Body b1, Body b2, double *d){
985
986             double a[3] = {b1.coord[0],b1.coord[1],b1.coord[2]};
987             double b[3] = {b2.coord[0],b2.coord[1],b2.coord[2]};
988
989       d[0] = b[0]-a[0];
990             d[1] = b[1]-a[1];
991             d[2] = b[2]-a[2];
992
993 }
994
995 void acc(double m, double d, double *x, double *a){
996   double mr = m/(d*d*d);
997   a[0] += x[0]*mr;
998   a[1] += x[1]*mr;
999   a[2] += x[2]*mr;
1000 }
1001
1002 void Compute_Gravity(Body body, Barnes* current, double *g, double Len){
1003     double dx[3]; //per la distanza vettoriale tra corpi
1004     double m;
```

```
1005      double l;
1006            double d;
1007      for(int k=0;k<current->subnodes.size();k++){
1008          current=current->subnodes[k];
1009          d = Distance(body,current->cm);
1010          l = Len/(2);
1011          if(d<l && current->subnodes.size()>0){
1012                                Compute_Gravity(body,current,g,l);
1013          }
1014          else if(d<=pow(10,-16)){ //escludo i casi di distanze tra un corpo e se stesso
1015
1016          }
1017          else{
1018              m = current->cm.mass;
1019              Vectorial_Distance(body,current->cm,dx);
1020              acc(m,d,dx,g);  //aggiungo gravità
1021            }
1022        }
1023  }
1024
1025  void Compute_Gravity_cm(Body body, int i, Body nodes_cm[], int Ncm, int Np, int rank, i
1026    double dx[3]; //per la distanza vettoriale tra corpi
1027    double m;
1028    double l;
1029          double d;
1030    for(int k=0;k<Ncm;k++){
1031          if(k>=((rank*(8/Np))+(rank0/8)) && k<(((rank+1)*(8/Np)))+(rank0/8)){
1032            //escludo i centri di massa dell'octree locale
1033          }
1034          else{
1035            m = nodes_cm[k].mass;
1036            if(m>0){ //escludo i nodi vuoti
1037              d = Distance(body,nodes_cm[k]);
1038              l = Len/(2);
1039              if(d<l){
1040                v[(k/(8/Np))].push_back(i);
1041              }
1042              else if(d<=pow(10,-16)){ //escludo i casi di distanze tra un corpo e se st
1043
1044              }
1045              else{
1046                Vectorial_Distance(body,nodes_cm[k],dx);
1047                acc(m,d,dx,g);
1048              }
1049            }
1050          }
1051    }
1052  }
```