

# N-body simulation using parallel programming

Andrea Buratti

September 6, 2024

## 1 Introduzione

Per realizzare la mia simulazione del problema degli N-corpi, ho utilizzato i due principali approcci conosciuti, ovvero:

- Brute Force algorithm
- Barnes-Hut algorithm

### 1.1 Il necessario

L'architettura della mia applicazione è abbastanza semplice. Ogni programma ha a disposizione, a seconda delle necessità, degli header che forniscono le strutture dati e le relative funzioni per utilizzarle.

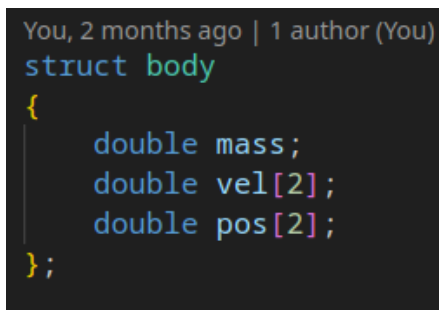
Nel caso Brute Force è necessario solo "particles.h" - "particles.c".

Per Barnes-Hut è invece anche necessario "tree.h" - "tree.c".

Questi header sono linkati e buildati utilizzando i makefile, quindi la loro gestione sarà completamente trasparente per i nostri scopi.

#### 1.1.1 Particles.h/Particles.c

Nell'header viene fornita l'implementazione della struct "body" che descrive una particella. Rappresentandone quindi massa, velocità istantanea e posizione istantanea. Volendo lavorare con dimensioni e forze sufficientemente grandi, i dati vengono rappresentati sotto forma di double.

A screenshot of a code editor showing the definition of a C struct named 'body'. The code is as follows:

```
You, 2 months ago | 1 author (You)
struct body
{
    double mass;
    double vel[2];
    double pos[2];
};
```

Figure 1: Body Struct

Velocità e Posizione sono ovviamente entrambe rappresentate sotto forma vettoriale, con vel[0] che rappresenta la componente x del vettore, e vel[1] che rappresenta la componente y.

Di seguito sono poi definite le signature delle funzioni implementate nel file Particles.c, che verranno utilizzate per eseguire le operazioni necessarie a lavorare con tali particelle.

```

// initialize the simulation
struct body* simulation__init(char *simulation_name, struct body *bodies, int *n_bodies);
// use case triangle test
struct body* earth_sun__init(struct body *bodies, int *n_bodies);
// use case square test
struct body* square__init(struct body *bodies, int *n_bodies);
// use case triangle test
struct body* triangle__init(struct body *bodies, int *n_bodies);
// compute the force between two particles
void __compute_force(const struct body body1,const struct body body2,double G,double force[]);
// compute the acceleration of a particle
void __compute_acceleration(struct body bodies, double force[],double acc[]);

void print_bodies(struct body *bodies,int n_bodies);

```

Figure 2: Signature

### 1.1.2 Tree.h/Tree.c

Nell'header viene fornita l'implementazione della struct "node", che descrive un nodo dell'albero di Barnes-Hut.

Per un corretto calcolo delle forze agenti su una particella tramite il suddetto metodo, ogni nodo dell'albero, che rappresenta un certo sottospazio in cui vivono le particelle, ha bisogno delle coordinate (x,y) del centro di massa del sottospazio, massa totale del sottospazio, coordinate (x,y) minime e massime del sottospazio, un puntatore ad una struct di tipo "body", che rappresenta l'eventuale singolarità presente all'interno del nodo, e i puntatori ai nodi figli, che nel caso di rappresentazione 2D dello spazio sono 4.

```

You, 2 months ago | 1 author (You)
struct node
{
    double com[2];
    double mass;
    double minX;
    double minY;
    double maxX;
    double maxY;
    struct body *body;
    struct node *ne;
    struct node *se;
    struct node *nw;
    struct node *sw;
};

```

Figure 3: BH-node struct

La segnatura delle funzioni necessarie ad interagire con una struttura di questo tipo sono quindi:

```

// unused
You: 1 second ago - Uncommitted changes
void calculate_force(struct node *root, struct body *body, double theta, double *force, double G);
// print the whole tree (debug purposes)
void print_tree(struct node *root);
// unused
void getBoundCoordinates(struct body *bodies, int n_bodies, double *minx, double *miny, double *maxx, double *maxy);
// initialize a node
void init_node(struct node *node);
// unused
void insert_node(struct node *root, struct body *body);
// unused
void build_tree(struct node *root, struct body *bodies, int n_bodies, double minx, double miny, double maxx, double maxy);
// unused
void tree__insert(struct node *root, struct body *body);
// free the whole memory allocated for the tree
void tree__free(struct node *root);
// free a single node
void node__free(struct node *root);
// insert a node into the tree
void tree__insert(struct node *root, struct body *body);
// calculate total force acting on a body using the BH-tree
void tree__calculate_force(struct node *root, struct body *body, double theta, double *force, double G);

```

Figure 4: BH functions signature

Ovviamente mi servo dell'header per le particles per rappresentare i corpi all'interno dei nodi dell'albero.

## 2 Implementazioni Seriali

Di seguito presenterò le implementazioni iniziali, necessarie al confronto prestazionale rispetto alla seguente parallelizzazione del sistema.

### 2.1 Brute Force approach

Il main si serve dell'header Particle per rappresentare i corpi del sistema. Nell'approccio brute force semplicemente si scorre l'array di particelle, e per ognuna si calcola la forza che intercorre tra lei e tutte le altre particelle del sistema. Ovviamente ciò ha un costo computazionale molto elevato. Il sistema è ovviamente temporizzato, e la norma che ho deciso prevede di salvare all'interno del file ".csv" i valori di posizione e velocità delle particelle ogni 10.000 istanti temporali.

### 2.2 Barnes-Hut approach

Il main si serve di entrambi gli header discussi precedentemente per eseguire le operazioni necessarie.

Il primo step necessario, dopo l'inizializzazione della lista di particelle, è la costruzione dell'albero di Barnes-Hut.

Inizialmente l'approccio utilizzato era quello di identificare le coordinate minime e massime tra il set di particelle definito. Calcolando i dati del nodo radice e proseguendo poi la costruzione dell'albero, analizzando di volta in volta la lista di corpi per includere nel nodo solo quelli presenti all'interno del sottospazio appropriato.

Questo tipo di implementazione è risultata tuttavia totalmente inefficiente e inutilmente complicata, in quanto non solo necessitava di scorrere un numero elevatissimo di volte la lista di corpi, ma anche di un algoritmo di costruzione per l'albero fondamentalmente indecifrabile.

Ho quindi deciso di adottare un approccio molto più semplice ed elegante, ovvero costruire l'albero utilizzando un'operazione di insert per ogni particella. Aggiustando i valori dei nodi man mano che l'inserzione di un corpo attraversava l'albero.

L'algoritmo di inserzione quindi controlla la posizione del centro di massa del nodo, considerando anche la particella che sto inserendo (in questo modo è come se avesse una visione aggiornata del sottospazio in analisi), in base alla posizione relativa quindi l'algoritmo sceglie il sottospazio adatto per continuare l'inserimento.

Banalmente, se la particella si trova in alto a destra rispetto al centro di massa, inserirà il nodo nel sottospazio nord-est, nel caso in cui si trovi in basso a sinistra rispetto al centro di massa, inserirà il nodo a sud-ovest, e così via...

Nel caso in cui due corpi finiscano nello stesso sottospazio, l'algoritmo procederà a riinserire entrambi i corpi, partendo dal nodo in cui sono situati e creando opportunamente i relativi sottospazi.

Per motivi di chiarezza, una possibile rappresentazione per un sottospazio di Barnes-Hut è:

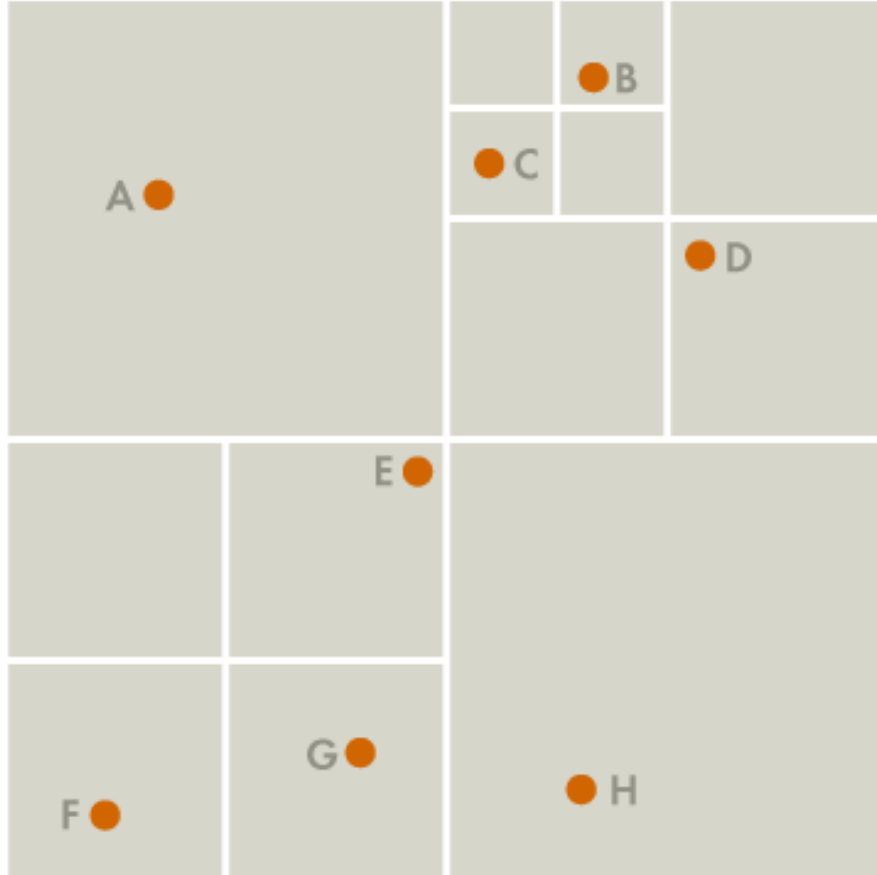


Figure 5: BH space representation

Per calcolare la forza che viene esercitata su una particella sarà quindi sufficiente scorrere l'albero costruito ed eseguire le dovute approssimazioni dell'algoritmo tradizionale.

### 3 Implementazioni Parallele

La parallelizzazione degli algoritmi prevede l'utilizzo di due librerie, MPI e Pthread. Entrambi gli algoritmi, per motivi di confronto prestazionale sono stati implementati usando entrambe le librerie. Ovviamente anche in questo caso mi sono avvalso di utilizzare gli header definiti in precedenza.

#### 3.1 MPI

Il problema principale riscontrato con MPI coinvolge la costruzione dell'albero. Inizialmente il mio obiettivo era la totale parallelizzazione dei calcoli. Tuttavia dovendo costruire un albero attraverso operazioni di inserzione, e non essendo condivisa la memoria tra i processi di MPI, la costruzione parallela dell'albero risulta totalmente infattibile. Il processo padre invierà quindi la lista di corpi ed il numero totale di corpi tramite una Bcast. Ogni processo procederà poi a costruire il proprio albero

ed a calcolare le forze agenti su un proprio sotto-set di particelle. Infine il processo padre richiederà le particelle aggiornate tramite un operazione di Allgather.

Analogamente l'algoritmo brute force seguirà la stessa linea operativa, non necessitando della costruzione di un albero e semplicemente calcolando le forze agenti su ogni particella, per ogni altra particella del sistema.

### 3.2 Pthread

Analogamente a quanto accaduto con MPI, la difficoltà maggiore di realizzare la parallelizzazione tramite pthread risiede nell'utilizzo di puntatori e la conseguente liberazione della memoria. Il primo approccio per implementare un algoritmo di Barnes-Hut parallelo utilizzando pthread è stato quello di introdurre delle barriere per coordinare le operazioni tra i thread. In particolare la suddivisione precisa è tra tutte le operazioni di insert e il seguente calcolo delle forze.

Ovviamente essendo la memoria condivisa, sono necessarie diverse accortezze per operare sull'albero, in particolare durante la fase embrionale avevo pensato di inserire all'interno di ogni nodo un lock, così da permettere a tutti i thread di accedere liberamente all'albero e di operare in totale parallelismo. Questa implementazione è stata tuttavia immediatamente scoraggiata dal fatto che avrebbe fondamentalmente implicato una totale reimplementazione di tutti gli algoritmi necessari ad operare con l'albero, cosa totalmente possibile in teoria, ma enormemente ed inutilmente tediosa in pratica, anche a fronte dei memory leak riscontrati per le operazioni di free e calloc.

Infatti nell'implementazione seriale, ad ogni istante temporale l'albero veniva totalmente costruito, ed in seguito distrutto utilizzando una visita in post-order. Poi nell'istante temporale successivo veniva riinizializzata la root con una calloc ed inizializzato il nodo per permettere gli inserimenti.

Questo però non è altrettanto fattibile utilizzando pthread. Poiché ogni thread simula tutti gli istanti temporali infatti, non è possibile cancellare totalmente l'albero e riinizializzarlo poiché la root cambia locazione di memoria se istanziata da un'altro thread, e i restanti thread non avrebbero più accesso a tale indirizzo poiché sarebbe aggiornato e non comunicato.

Ancora una volta quindi, ogni thread costruisce l'intero albero, e procede a scorrere solo la sezione assegnata dell'array. Stavolta non sarà necessario l'utilizzo di lock in quanto i thread non interferiscono tra loro per aggiornare l'array dei corpi.

Per quanto riguarda l'algoritmo brute force invece i lock sono necessari, in particolare è necessario un read-write lock, in quanto ogni thread legge parti dell'array assegnate ad altri thread, risultando in una corsa critica. Quindi è opportunamente modificata la struct body in modo da includere anche un read-write lock, così da coordinare l'accesso agli elementi dell'array.

## 4 Risultati

Per runnare i test mi sono avvalso dell'utilizzo di un semplice script python che utilizza le librerie subprocess e time per partire i processi singolarmente e calcolarne il tempo di esecuzione medio, per poi scrivere i risultati all'interno di un file .csv specifico.

Per ogni programma viene calcolato il tempo medio prendendo 5 istanze d'esecuzione e facendo variare il numero di corpi da 100 a 200 corpi, e considerando 4 thread che lavorano in parallelo.

Il numero di istanti rimane sempre lo stesso, 10.000

Le implementazioni seriali hanno prodotto i risultati seguenti:

Brute Force:

```

21    [t=10000,n=100] elapsed time : 3.673414
22    [t=10000,n=100] elapsed time : 3.733376
23    [t=10000,n=100] elapsed time : 3.623322
24    [t=10000,n=100] elapsed time : 3.278941
25    [t=10000,n=100] elapsed time : 3.182009

```

Figure 6: Serial Brute Force Performance (100)

Con Media di 3.49 secondi e Deviazione Standard di 0.223 secondi

```

26    [t=10000,n=200] elapsed time : 15.929214
27    [t=10000,n=200] elapsed time : 15.066488
28    [t=10000,n=200] elapsed time : 15.196888
29    [t=10000,n=200] elapsed time : 14.390729
30    [t=10000,n=200] elapsed time : 14.035121

```

Figure 7: Serial Brute Force Performance (200)

Con Media di 14.9 secondi e Deviazione Standard di 0.66 secondi

Barnes-Hut:

```

11    [t=10000,n=100] Elapsed time : 38.561798
12    [t=10000,n=100] Elapsed time : 37.694398
13    [t=10000,n=100] Elapsed time : 39.323209
14    [t=10000,n=100] Elapsed time : 37.428087
15    [t=10000,n=100] Elapsed time : 37.330946

```

Figure 8: Serial Barnes-Hut Performance (100)

Con Media di 38.06 secondi e Deviazione Standard di 0.763 secondi

```

16    [t=10000,n=200] Elapsed time : 157.661085
17    [t=10000,n=200] Elapsed time : 156.496374
18    [t=10000,n=200] Elapsed time : 148.618357
19    [t=10000,n=200] Elapsed time : 152.867313
20    [t=10000,n=200] Elapsed time : 153.991063

```

Figure 9: Serial Barnes-Hut Performance (200)

Con Media di 153.9 secondi e Deviazione Standard di 3.16 secondi

Il degrado delle performance in questo caso è ovviamente dovuto a tutte le operazioni di free e calloc necessarie a gestire la costruzione e la distruzione dell'albero per ogni istante temporale.

Mentre quelle parallelizzate usando Mpi e Pthread:

Brute Force parallelizzato usando Pthread:

```
11 [t=10000,n=100] Elapsed time : 16.958432
12 [t=10000,n=100] Elapsed time : 16.946788
13 [t=10000,n=100] Elapsed time : 16.539757
14 [t=10000,n=100] Elapsed time : 16.622134
15 [t=10000,n=100] Elapsed time : 16.736958
```

Figure 10: Pthread Brute Force Performance (100)

Con Media di 16.76 secondi e Deviazione Standard di 0.168 secondi

```
16 [t=10000,n=200] Elapsed time : 53.897120
17 [t=10000,n=200] Elapsed time : 53.971211
18 [t=10000,n=200] Elapsed time : 51.420678
19 [t=10000,n=200] Elapsed time : 52.761622
20 [t=10000,n=200] Elapsed time : 54.785525
```

Figure 11: Pthread Brute Force Performance (200)

Con Media di 53.367 secondi e Deviazione Standard di 1.166 secondi

Brute Force parallelizzato usando Mpi:

```
6 [t=10000,n=100] Elapsed time : 1.014472
7 [t=10000,n=100] Elapsed time : 0.995683
8 [t=10000,n=100] Elapsed time : 1.060490
9 [t=10000,n=100] Elapsed time : 0.961434
10 [t=10000,n=100] Elapsed time : 0.966031
```

Figure 12: Mpi Brute Force Performance (100)

Con Media di 0.999 secondi e Deviazione Standard di 0.037 secondi

```

11 [t=10000,n=200] Elapsed time : 3.842630
12 [t=10000,n=200] Elapsed time : 3.720276
13 [t=10000,n=200] Elapsed time : 4.411916
14 [t=10000,n=200] Elapsed time : 4.098866
15 [t=10000,n=200] Elapsed time : 4.524670

```

Figure 13: Mpi Brute Force Performance (200)

Con Media di 4.119 secondi e Deviazione Standard di 0.275 secondi

Barnes-Hut parallelizzato usando Pthread:

```

11 [t=10000,n=100] Elapsed time : 20.877864
12 [t=10000,n=100] Elapsed time : 21.339680
13 [t=10000,n=100] Elapsed time : 20.883362
14 [t=10000,n=100] Elapsed time : 20.935144
15 [t=10000,n=100] Elapsed time : 20.996928

```

Figure 14: Barnes Hut Pthread Performance (100)

Con Media di 20.84 secondi e Deviazione Standard di 0.209 secondi

```

16 [t=10000,n=200] Elapsed time : 85.474420
17 [t=10000,n=200] Elapsed time : 93.468895
18 [t=10000,n=200] Elapsed time : 75.635006
19 [t=10000,n=200] Elapsed time : 81.692574
20 [t=10000,n=200] Elapsed time : 80.955912

```

Figure 15: Barnes Hut Pthread Performance (200)

Con Media di 81.64 secondi e Deviazione Standard di 6.999 secondi

Barnes-Hut parallelizzato usando Mpi:

```

6 [t=10000,n=100] Elapsed time : 9.572171
7 [t=10000,n=100] Elapsed time : 10.074031
8 [t=10000,n=100] Elapsed time : 9.504416
9 [t=10000,n=100] Elapsed time : 9.502156
10 [t=10000,n=100] Elapsed time : 9.459764

```

Figure 16: Barnes Hut Mpi Performance (100)



Con Media di 9.662 secondi e Deviazione Standard di 0.221 secondi

```
11 [t=10000,n=200] Elapsed time : 45.361392
12 [t=10000,n=200] Elapsed time : 35.354041
13 [t=10000,n=200] Elapsed time : 35.383791
14 [t=10000,n=200] Elapsed time : 35.851208
15 [t=10000,n=200] Elapsed time : 43.544739
```

Figure 17: Barnes Hut Mpi Performance (200)

Con Media di 39.6 secondi e Deviazione Standard di 4.45 secondi

Dai risultati ottenuti, l'algoritmo più stabile secondo i dati in input è il Brute Force parallelizzato usando Mpi. Ovviamente gestendo diversamente l'allocazione e la liberazione della memoria potremmo ottenere risultati più stabili ed efficienti tramite l'algoritmo di Barnes-Hut.