

N-body simulation using parallel programming

Andrea Buratti

October 1, 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.

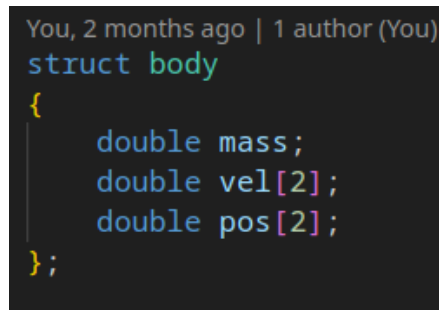
Nel caso Brute Force è necessario solo "particles.h" - "particles.c".

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.



```
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 necessitava di scorrere un numero elevatissimo di volte la lista di corpi.

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 dimensione dello spazio del nodo, e la posizione della particella che sto inserendo e sceglie il sottospazio adatto per continuare l'inserimento.

La decisione per il sottospazio da scegliere viene fatta nel seguente modo:

Siano (a,b) e (c,d) le coordinate dei vertici dello spazio, rispettivamente del vertice in basso a sinistra, e in alto a destra, allora:

- Il sottospazio nord-est avrà vertici $(\frac{a+c}{2}, \frac{b+d}{2})$ e (c, d)
- Il sottospazio sud-est avrà vertici $(\frac{a+c}{2}, b)$ e $(c, \frac{d+b}{2})$
- Il sottospazio nord-ovest avrà vertici $(a, \frac{b+d}{2})$ e $(\frac{a+c}{2}, d)$
- Il sottospazio sud-ovest avrà vertici (a, b) e $(\frac{a+c}{2}, \frac{b+d}{2})$

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, sfruttando la posizione del centro di massa risultante dai due corpi per garantire una suddivisione univoca del sottospazio relativo.

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

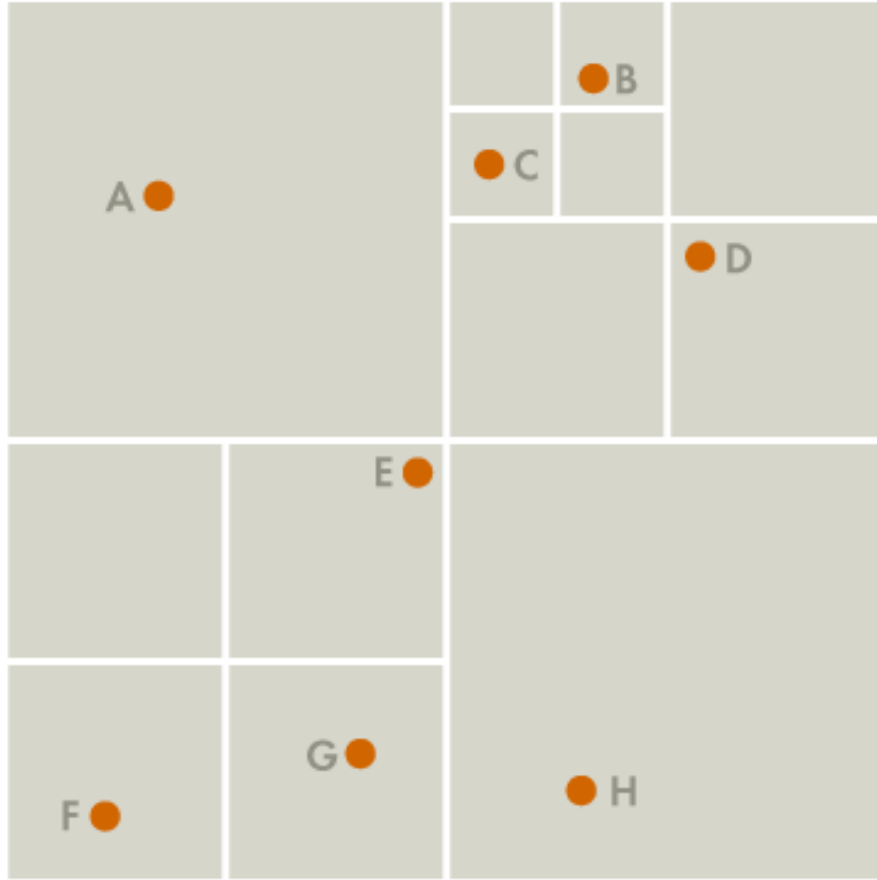


Figure 5: BH space representation

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.

3.2.1 Utilizzo di lock

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 reso la costruzione dell'albero seriale, lockando la root infatti solo un thread alla volta avrebbe avuto accesso per inserire il corpo.

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.

Per quanto riguarda l'algoritmo brute force invece, inizialmente era presente un'implementazione che faceva utilizzo di read-write locks, che risultavano però in un degrado delle performance, in quanto i thread si bloccavano a vicenda per calcolare le forze.

3.2.2 Utilizzare memoria locale

Per ovviare ai problemi di concorrenza per le risorse generati dall'approccio con read-write lock ho deciso di utilizzare degli array locali, per permettere ai thread di effettuare i calcoli solo su sezioni dell'array, dividendosi equamente il lavoro. Sincronizzare i calcoli necessita ovviamente di una barrier, evitando così che l'array venga modificato mentre un processo lo sta utilizzando per ultimi calcoli.

4 Risultati

Per runnare i test mi sono avvalso dell'utilizzo di un semplice script python che utilizza le librerie subprocess e time per startare 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 2000 a 3000 corpi, considerando prima 2 e poi 4 thread che lavorano in parallelo. I tempi di esecuzione sono:

```
tester > results.csv
1  mpi-bh-times.csv [2000] : 0.8332488
2
3  mpi-bh-times.csv [3000]: 0.944452
4
5  mpi-parallel-times.csv [2000] : 5.4818074
6
7  mpi-parallel-times.csv [3000]: 7.920464
8
9  pthread-bh-times.csv [2000] : 0.9400238
10
11 pthread-bh-times.csv [3000]: 1.167682
12
13 pthread-parallel-times.csv [2000] : 5.8437946
14
15 pthread-parallel-times.csv [3000]: 8.558942
16
17 serial-bh-times.csv [2000] : 1.7507034000000001
18
19 serial-bh-times.csv [3000]: 2.045659
20
21 serial-nbodies-times.csv [2000] : 20.0437352
22
23 serial-nbodies-times.csv [3000]: 29.235556
24
25
```

Figure 6: Tempi di esecuzione di ogni algoritmo

4.1 Test con 2 processi

Lo speedup desiderato è di 2, in quanto parallelizzando con 2 processi al massimo posso dimezzare i tempi di esecuzione. Di seguito riporto i risultati ottenuti:

4.1.1 Speedup


```
tester >  speedups2.csv
1  MPI-BH-2000: 1.802275159782236
2  MPI-Parallel-2000: 1.9192008430794236
3  Pthread-BH-2000: 1.8644269680509558
4  Pthread-Parallel-2000: 1.9446077806467597
5
6
7  MPI-BH-3000: 1.7401404846145616
8  MPI-Parallel-3000: 1.9532525101748326
9  Pthread-BH-3000: 1.7925239115304739
10 Pthread-Parallel-3000: 1.930493907923825
11
```

Figure 7: Lo speedup ottenuto usando 2 processi

4.1.2 Efficiency


```
tester >  efficiencies2.csv
1  MPI-BH-2000:: 0.901137579891118
2  MPI-Parallel-2000:: 0.9596004215397118
3  Pthread-BH-2000:: 0.9322134840254779
4  Pthread-Parallel-2000:: 0.9723038903233798
5  MPI-BH-3000:: 0.8700702423072808
6  MPI-Parallel-3000:: 0.9766262550874163
7  Pthread-BH-3000:: 0.8962619557652369
8  Pthread-Parallel-3000:: 0.9652469539619125
9
```

Figure 8: L'efficienza delle implementazioni considerando 2 processi

4.2 Test con 4 processi

Analogamente, lo speedup desiderato è di 4, tuttavia la non parallelizzabilità della costruzione dell'albero pone un limite allo speedup ottenibile. Poichè nonostante l'aumentare del numero di processi, comunque la lista dei corpi dovrà essere percorsa per intero da ogni processo. Di seguito riporto quindi gli speedup ottenuti utilizzando 4 processi.

4.2.1 Speedup

```
tester > speedups4.csv
1 MPI-BH-2000: 2.101057211243509
2 MPI-Parallel-2000: 3.6564099643486196
3 Pthread-BH-2000: 1.8624032710661158
4 Pthread-Parallel-2000: 3.429917814017625
5
6
7
8 MPI-BH-3000: 2.165974554556505
9 MPI-Parallel-3000: 3.691141832094685
10 Pthread-BH-3000: 1.7518973487644751
11 Pthread-Parallel-3000: 3.415790876956521
12
```

Figure 9: Lo speedup ottenuto usando 4 processi

4.2.2 Efficiency

Notiamo infine il crollo dell'efficienza nel caso delle implementazioni che usano l'algoritmo di Barnes-Hut, per i motivi sopra citati

```
tester > efficiencies4.csv
1 MPI-BH-2000:: 0.5252643028108772
2 MPI-Parallel-2000:: 0.9141024910871549
3 Pthread-BH-2000:: 0.46560081776652895
4 Pthread-Parallel-2000:: 0.8574794535044062
5 MPI-BH-3000:: 0.5414936386391263
6 MPI-Parallel-3000:: 0.9227854580236713
7 Pthread-BH-3000:: 0.4379743371911188
8 Pthread-Parallel-3000:: 0.8539477192391303
9
```

Figure 10: L'efficienza delle implementazioni considerando 4 processi