

# Parallelizzazione del Problema degli N-Corpi

S. Bianco, A. Coppola, F. D'Aprile

February 12, 2024

# Contents

<b>1</b>	<b>Introduzione</b>	<b>3</b>
1.1	Descrizione ed analisi del problema . . . . .	3
1.2	Informazioni sulle statistiche . . . . .	5
<b>2</b>	<b>Simulazione tramite metodo esaustivo</b>	<b>6</b>
2.1	Implementazione sequenziale . . . . .	6
2.2	Parallelizzazione tramite OpenMP . . . . .	9
2.2.1	Approccio naïve . . . . .	9
2.2.2	Riduzione del false sharing . . . . .	12
2.2.3	Riduzione dell'overhead di OpenMP . . . . .	15
2.3	Parallelizzazione tramite MPI . . . . .	20
2.3.1	Modalità di implementazione . . . . .	20
2.3.2	Riduzione e localizzazione dei dati trasmessi . . . . .	25
2.4	Analisi dello speed-up e dell'efficienza . . . . .	29

<b>3</b>	<b>Simulazione tramite algoritmo di Barnes-Hut</b>	<b>30</b>
3.1	Descrizione dell'algoritmo . . . . .	30
3.2	Implementazione sequenziale . . . . .	33
3.2.1	Struttura dati e funzioni principali . . . . .	33
3.2.2	Ottimizzazioni della struttura dati . . . . .	42
3.3	Parallelizzazione tramite OpenMP . . . . .	47
3.3.1	Analisi, limitazioni riscontrate e implementazione . . .	47
3.4	Parallelizzazione tramite MPI . . . . .	51
3.4.1	Analisi, limitazioni riscontrate e implementazione . . .	51
3.5	Analisi dello speed-up e dell'efficienza . . . . .	54
<b>4</b>	<b>Conclusioni</b>	<b>55</b>

# Chapter 1

## Introduzione

### 1.1 Descrizione ed analisi del problema

In fisica, il **problema degli n-corpi** corrisponde al problema inerente alla previsione dei movimenti individuali di un gruppo di  $n$  oggetti celesti che interagiscono tra loro gravitazionalmente. Il problema degli n-corpi nella relatività generale è considerevolmente più difficile da risolvere a causa di fattori aggiuntivi come le distorsioni del tempo e dello spazio.

Nella fisica classi, il problema viene informalmente definito come:

**Definizione 1.** *Date le proprietà orbitali quasi-stazionarie (posizione istantanea, velocità e tempo) di un gruppo di  $n$  corpi celesti, prevedere le loro forze interattive al fine di prevedere i loro veri movimenti orbitali per tutti i tempi futuri.*

In particolare, dati  $n$  corpi  $b_1, \dots, b_n$  in uno spazio tridimensionale  $\mathbb{R}^3$ , siano:

- Gli scalari  $m_1, \dots, m_n$  uguali alle masse dei corpi
- I vettori  $\vec{p}_1, \dots, \vec{p}_n$  uguali alle posizioni dei corpi
- I vettori  $\vec{v}_1, \dots, \vec{v}_n$  uguali alle velocità dei corpi

La **legge di gravità di Newton** afferma che la forza gravitazionale subita in un singolo istante dal corpo  $b_i$  da parte di un corpo  $b_j$  corrisponde a:

$$\vec{F}_{ij} = \frac{Gm_i m_j}{\|\vec{p}_j - \vec{p}_i\|^2} \cdot \frac{\vec{p}_j - \vec{p}_i}{\|\vec{p}_j - \vec{p}_i\|} = \frac{Gm_i m_j}{\|\vec{p}_j - \vec{p}_i\|^3} \cdot (\vec{p}_j - \vec{p}_i)$$

dove  $G$  è la **costante gravitazionale**, ossia  $G = 6.67430 \cdot 10^{-11} \frac{\text{N} \cdot \text{m}^2}{\text{kg}^2}$  e  $\|\vec{p}_j - \vec{p}_i\|$  è la **distanza vettoriale** tra  $\vec{p}_j$  e  $\vec{p}_i$ , ossia:

$$\|\vec{p}_j - \vec{p}_i\| = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2}$$

Di conseguenza, la somma delle forze subite dal corpo  $b_i$  in un singolo istante corrisponde a:

$$\vec{F}_i = \sum_{\substack{j=0 \\ j \neq i}}^n \vec{F}_{ij} = \sum_{\substack{j=0 \\ j \neq i}}^n \frac{Gm_i m_j}{\|\vec{p}_j - \vec{p}_i\|^3} \cdot (\vec{p}_j - \vec{p}_i)$$

Tramite la **seconda legge di Newton**, invece, sappiamo che la somma delle forze applicate sul corpo  $b_i$  ad un istante  $t$  corrisponde al prodotto tra la massa  $m_i$  e la sua *accelerazione*  $\vec{a}_i$ , ossia:

$$\vec{F}_i = m_i \vec{a}_i$$

implicando dunque che:

$$m_i \vec{a}_i = \sum_{\substack{j=0 \\ j \neq i}}^n \frac{Gm_i m_j}{\|\vec{p}_j - \vec{p}_i\|^3} \cdot (\vec{p}_j - \vec{p}_i) \iff \vec{a}_i = \sum_{\substack{j=0 \\ j \neq i}}^n \frac{Gm_j}{\|\vec{p}_j - \vec{p}_i\|^3} \cdot (\vec{p}_j - \vec{p}_i)$$

A questo punto, siano  $\vec{v}_i'$  e  $\vec{p}_i'$  rispettivamente la velocità e la posizione del corpo  $i$  all'iterazione  $k + 1$  della simulazione. Risulta evidente che:

$$\begin{aligned} \vec{v}_i' &= \vec{v}_i + \Delta t \cdot \vec{a}_i \\ \vec{p}_i' &= \vec{p}_i + \Delta t \cdot \vec{v}_i' \end{aligned}$$

Tuttavia, affinché ogni passo della simulazione risulti **omogeneo**, è consigliato impostare un valore *non elevato* per  $\Delta t$ . Poiché tale parametro non influisce sulle performance della simulazione, in quanto la moltiplicazione richieda tempo costante, è stato utilizzato il valore  $\Delta t = 1$ .

## 1.2 Informazioni sulle statistiche

Le statistiche presentate nelle sezioni successive fanno riferimento a **tempistiche medie** raccolte eseguendo 10 volte ogni simulazione con un range da 1000 a 10000 corpi.

Ogni test è stato eseguito sui seguenti dispositivi:

### 1. Lenovo IdeaPad 5

- Processore: AMD Ryzen 7 5825U @ 8 cores, 2.0 GHz
- Cache per core:
  - Cache L1: 512 KB
  - Cache L2: 4 MB
  - Cache L3: 16 MB
- Memoria: 16 GB DDR4 @ 3200 MHz

### 2. MacBook Air M1

- Processore: Apple M1 @ 4 High core, 4 Low core, 2.0 GHz
- Cache per core:
  - Cache L1: 192+128 MB (High core), 128+64 MB (Low core)
  - Cache L1: 12 MB (High core), 4 MB (Low core)
- Memoria: 16 GB DDR4 @ 3200 MHz

Inoltre, le formule utilizzate per calcolare i parametri di **speed-up** (S) ed **efficienza** (E) tra le varie versioni sequenziali e parallele sono le seguenti:

$$S = \frac{T_{\text{sequential}}}{T_{\text{parallel}}} \qquad E = \frac{S}{p}$$

dove  $p$  è il numero di processori utilizzati

## Chapter 2

# Simulazione tramite metodo esaustivo

### 2.1 Implementazione sequenziale

All'avvio, il programma procederà con lo **scansionare il file** contenente i dati dei corpi dato come argomento in input, costruendo un array di struct **bodies**, ognuno corrispondente ad un corpo. Tali struct sono dotati di tre attributi: un double per la **massa** del corpo e due **Vec3D** per la *posizione* e la *velocità* del corpo, dove un **Vec3D** è un ulteriore struct composto da tre double, uno per ogni coordinata possibile dello spazio tridimensionale (dunque  $x, y$  e  $z$ ).

Tramite l'analisi del capitolo precedente, risulta evidente che l'intera simulazione possa essere implementata tramite tre **cicli for annidati**:

- Un ciclo esterno che iteri sul numero di passi della simulazione
- Un primo ciclo interno che iteri su ognuno degli  $n$  corpi della simulazione
- Un secondo ciclo interno al primo che calcoli la sommatoria mostrata precedentemente

```

for (k = 0; k < num_steps; ++k){
    for (i = 0; i < num_bodies; ++i) {
        for (j = 0; j < num_bodies; ++j) {
            if (i == j) continue;

            dist_ij = vector_distance(&bodies[i].pos, &bodies[j].pos);
            scalar = G * bodies[j].mass * (pow(dist_ij, -3.0)) * TIME_DELTA;

            bodies[i].vel.x += scalar * (bodies[j].pos.x - bodies[i].pos.x);
            bodies[i].vel.y += scalar * (bodies[j].pos.y - bodies[i].pos.y);
            bodies[i].vel.z += scalar * (bodies[j].pos.z - bodies[i].pos.z);
        }
    }

    for (i = 0; i < num_bodies; ++i) {
        bodies[i].pos.x += bodies[i].vel.x * TIME_DELTA;
        bodies[i].pos.y += bodies[i].vel.y * TIME_DELTA;
        bodies[i].pos.z += bodies[i].vel.z * TIME_DELTA;
    }

    if(k % interval == 0){
        for (size_t i = 0; i < num_bodies; i++) {
            fprintf(out_fp, "%lf,%lf,%lf,", bodies[i].pos.x, bodies[i].pos.y, bodies[i].pos.z);
        }

        fprintf(out_fp, "\n");
    }
}

```

Figure 2.1: Screenshot del frammento di codice principale di `seq_ext_1.c`

Dati  $n$  corpi, eseguendo la simulazione per  $k$  passi risulta evidente che il numero complessivo di operazioni eseguite dalla simulazione sia  $kn(n-1) = O(kn^2)$ .

Tuttavia, poiché  $\|\vec{p}_i - \vec{p}_j\| = \|\vec{p}_j - \vec{p}_i\|$ , è possibile **migliorare l'efficienza del programma sequenziale** effettuando nella stessa iterazione i calcoli dell'interazione del corpo  $b_i$  con il corpo  $b_j$  e viceversa.

In tal modo, per  $n$  corpi e  $k$  passi di simulazione, il numero complessivo di iterazioni risulta essere:

$$k \cdot \sum_{i=0}^n i = k \cdot \frac{n(n+1)}{2}$$

**dimezzando** le iterazioni effettuate, presentando tuttavia comunque un comportamento asintotico pari a  $O(kn^2)$ . Inoltre, ogni iterazione effettua quasi il doppio dei calcoli, risparmiando effettivamente solo il ricalcolo della distanza.



```

for (k = 0; k < num_steps; ++k){
    for (i = 0; i < num_bodies; ++i) {
        for (j = i+1; j < num_bodies; ++j) {

            dist_ij = vector_distance(&bodies[i].pos, &bodies[j].pos);
            scalar_i = G * bodies[j].mass * (pow(dist_ij, -3.0)) * TIME_DELTA;
            scalar_j = G * bodies[i].mass * (pow(dist_ij, -3.0)) * TIME_DELTA;

            bodies[i].vel.x += scalar_i * (bodies[j].pos.x - bodies[i].pos.x);
            bodies[i].vel.y += scalar_i * (bodies[j].pos.y - bodies[i].pos.y);
            bodies[i].vel.z += scalar_i * (bodies[j].pos.z - bodies[i].pos.z);

            bodies[j].vel.x += scalar_j * (bodies[i].pos.x - bodies[j].pos.x);
            bodies[j].vel.y += scalar_j * (bodies[i].pos.y - bodies[j].pos.y);
            bodies[j].vel.z += scalar_j * (bodies[i].pos.z - bodies[j].pos.z);

        }

        for (i = 0; i < num_bodies; ++i) {
            bodies[i].pos.x += bodies[i].vel.x * TIME_DELTA;
            bodies[i].pos.y += bodies[i].vel.y * TIME_DELTA;
            bodies[i].pos.z += bodies[i].vel.z * TIME_DELTA;
        }
    }
}

```

Figure 2.2: Screenshot del frammento di codice principale di `seq_ext_2.c`

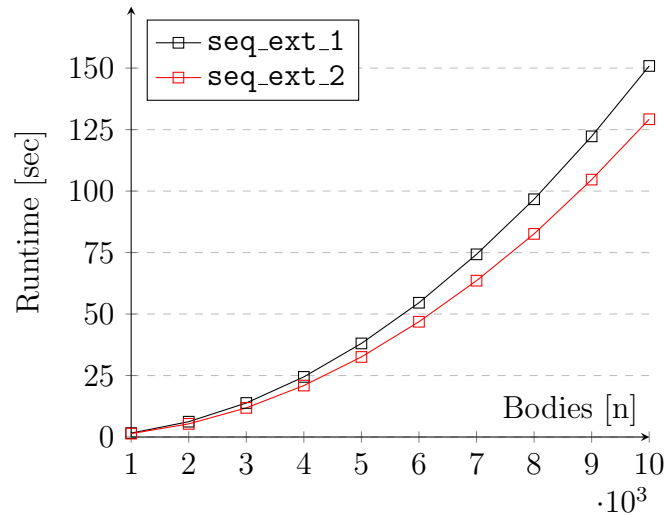


Figure 2.3: Confronto delle performance di `seq_ext_1` e `seq_ext_2` su 100 passi di simulazione da 1000 a 10000 corpi (Lenovo IdeaPad 5)

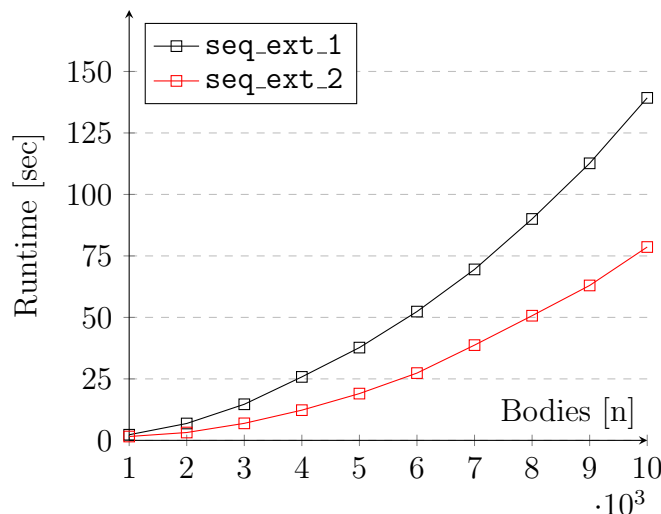


Figure 2.4: Confronto delle performance di `seq_ext_1` e `seq_ext_2` su 100 passi di simulazione da 1000 a 10000 corpi (MacBook Air M1)

## 2.2 Parallelizzazione tramite OpenMP

### 2.2.1 Approccio naïve

Tramite la libreria **OpenMP**, il codice mostrato nella sezione precedente può facilmente essere parallelizzato applicando la direttiva `#pragma omp for` sul primo *ciclo for* interno (ossia quello che itera sui corpi), in modo che ognuno dei  $p$  thread su cui viene parallelizzato il ciclo vada a lavorare con  $\frac{n}{p}$  corpi.

Tuttavia, tale parallelizzazione non può essere applicata sulla seconda versione del programma sequenziale (ossia `seq_ext_2`), poiché potrebbe verificarsi una **race condition** nell'aggiornare la velocità del corpo  $b_j$ :

- Supponiamo che un primo thread stia calcolando l'interazione tra il corpo  $b_i$  e  $b_j$ , aggiornando le velocità di quest'ultimi e che un secondo thread stia calcolando l'interazione tra il corpo  $b_j$  con un altro corpo  $b_x$ , aggiornando la velocità di quest'ultimi

- Se i due thread eseguono l'aggiornamento della velocità di  $b_j$  simultaneamente, una delle due modifiche verrebbe persa

Consideriamo ora il *ciclo for* che si occupa della *scrittura dell'output*. Parallelizzando tale ciclo tramite la direttiva `#pragma omp for`, i vari thread potrebbero accedere al file simultaneamente, scrivendo

Di conseguenza, solo il calcolo delle velocità della versione `seq_ext_1` risulta essere parallelizzabile. Per tanto, i **confronti statistici** effettuati nelle sezioni successive verranno effettuati solo con la versione `seq_ext_1`.

```
#pragma omp parallel num_threads(thread_count) \
    default(none) shared(bodies, num_bodies, num_steps, out_fp, interval) \
    private(i, j, k, dist_ij, scalar)
{
    for (k = 0; k < num_steps; ++k){

        #pragma omp for
        for (i = 0; i < num_bodies; ++i) {
            for (j = 0; j < num_bodies; ++j) {
                if(i == j) continue;

                dist_ij = vector_distance(&bodies[i].pos, &bodies[j].pos);
                scalar = G * bodies[j].mass * (pow(dist_ij, -3.0)) * TIME_DELTA;

                bodies[i].vel.x += scalar * (bodies[j].pos.x - bodies[i].pos.x);
                bodies[i].vel.y += scalar * (bodies[j].pos.y - bodies[i].pos.y);
                bodies[i].vel.z += scalar * (bodies[j].pos.z - bodies[i].pos.z);
            }
        }

        // Implicit barrier here thanks to #pragma omp for

        #pragma omp for
        for (i = 0; i < num_bodies; ++i) {
            bodies[i].pos.x += bodies[i].vel.x * TIME_DELTA;
            bodies[i].pos.y += bodies[i].vel.y * TIME_DELTA;
            bodies[i].pos.z += bodies[i].vel.z * TIME_DELTA;
        }

        // Implicit barrier here thanks to #pragma omp for

        // Only the master thread writes on file
        if(k % interval == 0){
            #pragma omp single
            {
                for (size_t i = 0; i < num_bodies; i++) {
                    fprintf(out_fp, "%lf,%lf,%lf,", bodies[i].pos.x, bodies[i].pos.y, bodies[i].pos.z);
                }

                fprintf(out_fp, "\n");
            }
        }
    }
}
```

Figure 2.5: Screenshot del frammento di codice principale di `omp_ext_1.c`

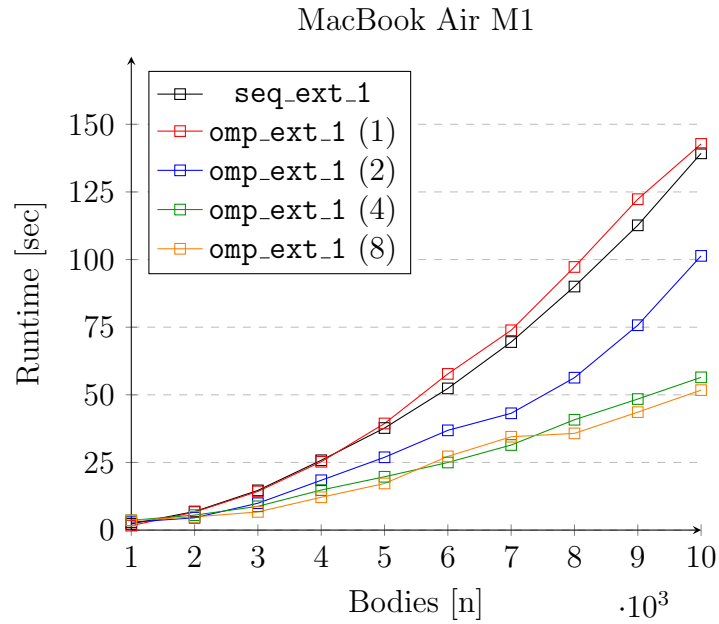
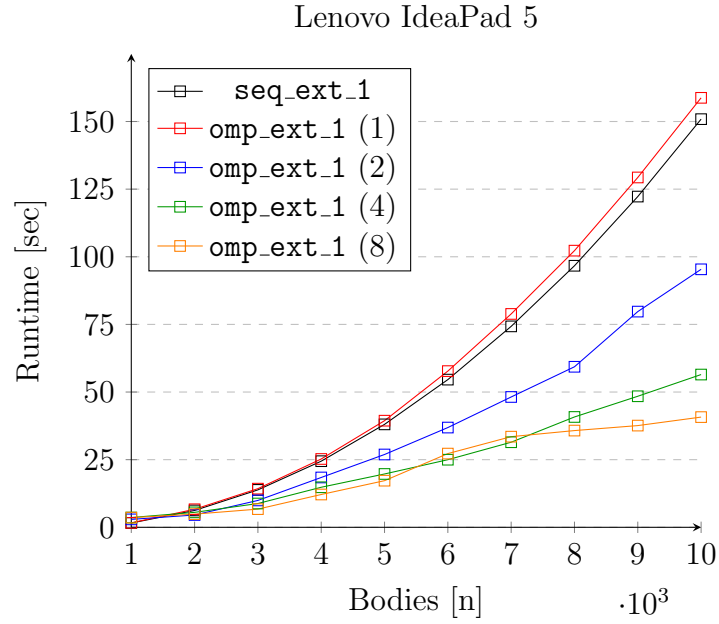
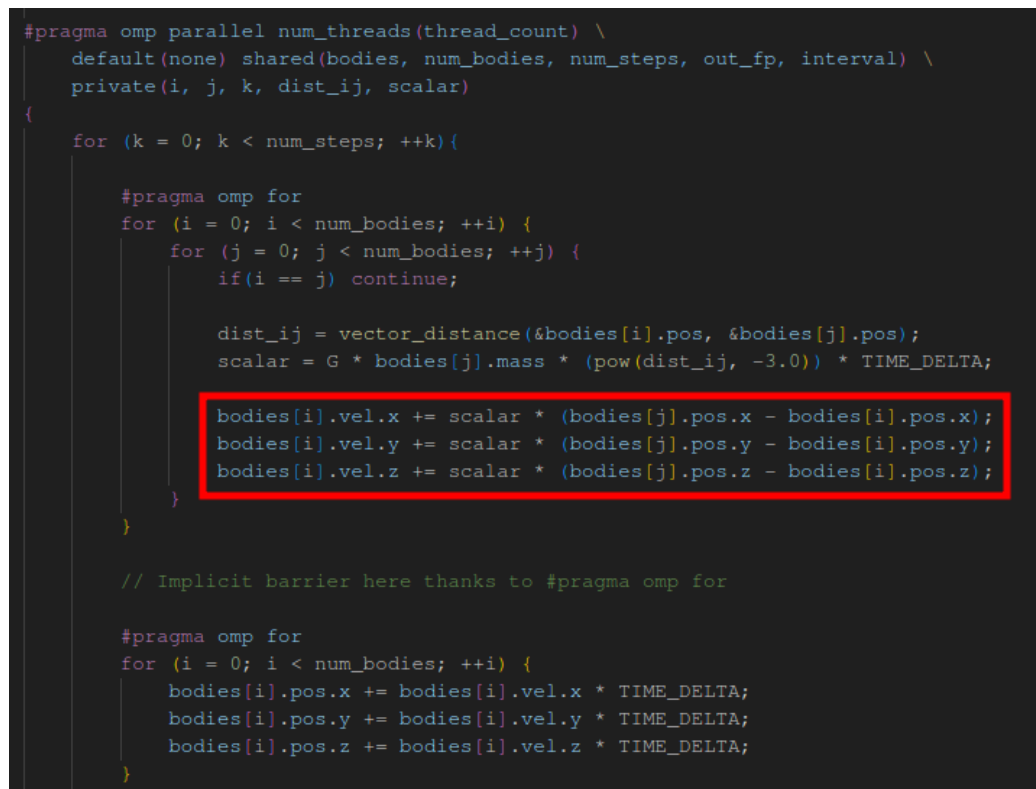


Figure 2.6: Confronto delle performance di `seq_ext_1` e `omp_ext_1` su 100 passi di simulazione da 1000 a 10000 corpi. I numeri tra parentesi nella legenda del grafico indicano il numero di thread attivi

## 2.2.2 Riduzione del false sharing

Nonostante l'incremento delle performance a livello di tempo di esecuzione ottenuto con la versione `omp_ext_1`, tale parallelizzazione risulta inefficiente da un punto di vista di **gestione delle cache**.

Nel secondo *ciclo for* interno, ogni thread legge il proprio valore `bodies[i].pos` e aggiorna i valori del proprio valore `bodies[i].vel`, entrambi dettati dal proprio indice attuale `i`.



```
#pragma omp parallel num_threads(thread_count) \
    default(none) shared(bodies, num_bodies, num_steps, out_fp, interval) \
    private(i, j, k, dist_ij, scalar)
{
    for (k = 0; k < num_steps; ++k){

        #pragma omp for
        for (i = 0; i < num_bodies; ++i) {
            for (j = 0; j < num_bodies; ++j) {
                if(i == j) continue;

                dist_ij = vector_distance(&bodies[i].pos, &bodies[j].pos);
                scalar = G * bodies[j].mass * (pow(dist_ij, -3.0)) * TIME_DELTA;

                bodies[i].vel.x += scalar * (bodies[j].pos.x - bodies[i].pos.x);
                bodies[i].vel.y += scalar * (bodies[j].pos.y - bodies[i].pos.y);
                bodies[i].vel.z += scalar * (bodies[j].pos.z - bodies[i].pos.z);
            }
        }

        // Implicit barrier here thanks to #pragma omp for

        #pragma omp for
        for (i = 0; i < num_bodies; ++i) {
            bodies[i].pos.x += bodies[i].vel.x * TIME_DELTA;
            bodies[i].pos.y += bodies[i].vel.y * TIME_DELTA;
            bodies[i].pos.z += bodies[i].vel.z * TIME_DELTA;
        }
    }
}
```

Figure 2.7: Screenshot del frammento di codice di interesse di `omp_ext_1.c`

Poiché all'interno dello struct `bodies` i due attributi `vel` e `pos` risultano essere memorizzati sequenzialmente all'interno della memoria (ossia accanto), ogni volta che il valore di `bodies[i].vel` viene modificato da un thread e successivamente un altro thread accede a `bodies[j].pos`, se  $i = j$  allora l'intero blocco di cache contenente lo struct `bodies[i]` verrà considerato

dal secondo thread come *"sporco"* poiché non sincronizzato con quello del primo thread, dunque potenzialmente scorretto, venendo per tanto **ricaricato** nonostante il valore di `bodies[i].pos` non sia stato realmente modificato (**false sharing tra le cache**). Inoltre, considerando il fatto che anche l'array `bodies` memorizzi i vari corpi sequenzialmente, il false sharing si verifica potenzialmente ad ogni aggiornamento di un corpo.

Introducendo una **variabile di supporto** che funga da *"accumulatore"* per l'incremento della velocità del corpo attualmente computato, è possibile **azzerare il false sharing**

```
#pragma omp for
for (i = 0; i < num_bodies; ++i) {

    vel_i = bodies[i].vel;
    pos_i = bodies[i].pos;

    for (j = 0; j < num_bodies; ++j) {
        if (i == j) continue;

        pos_j = bodies[j].pos;
        dist_ij = vector_distance(&pos_i, &pos_j);
        scalar = G * bodies[j].mass * (pow(dist_ij, -3.0)) * TIME_DELTA;

        vel_i.x += scalar * (pos_j.x - pos_i.x);
        vel_i.y += scalar * (pos_j.y - pos_i.y);
        vel_i.z += scalar * (pos_j.z - pos_i.z);
    }

    bodies[i].vel = vel_i;
}

// Implicit barrier here thanks to #pragma omp for

#pragma omp for
for (i = 0; i < num_bodies; ++i) {
    bodies[i].pos.x += bodies[i].vel.x * TIME_DELTA;
    bodies[i].pos.y += bodies[i].vel.y * TIME_DELTA;
    bodies[i].pos.z += bodies[i].vel.z * TIME_DELTA;
}

// Implicit barrier here thanks to #pragma omp for
```

Figure 2.8: Screenshot del frammento di codice principale di `omp_ext.2.c`

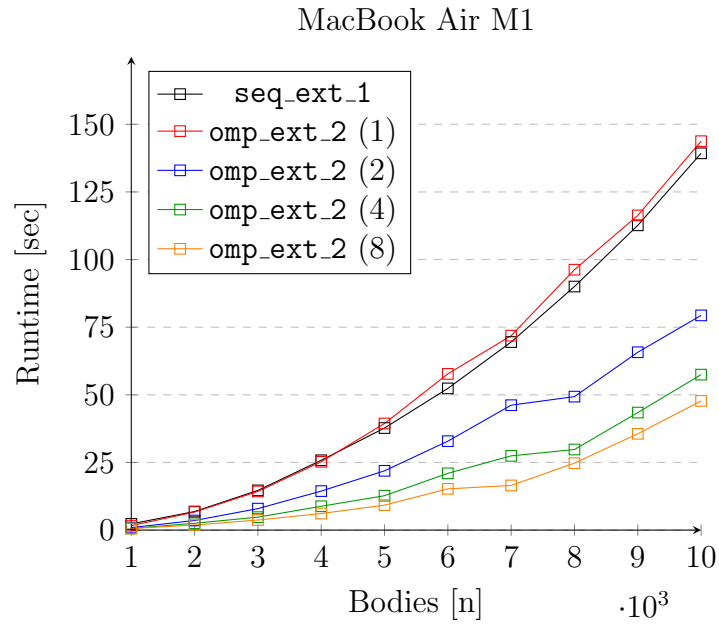
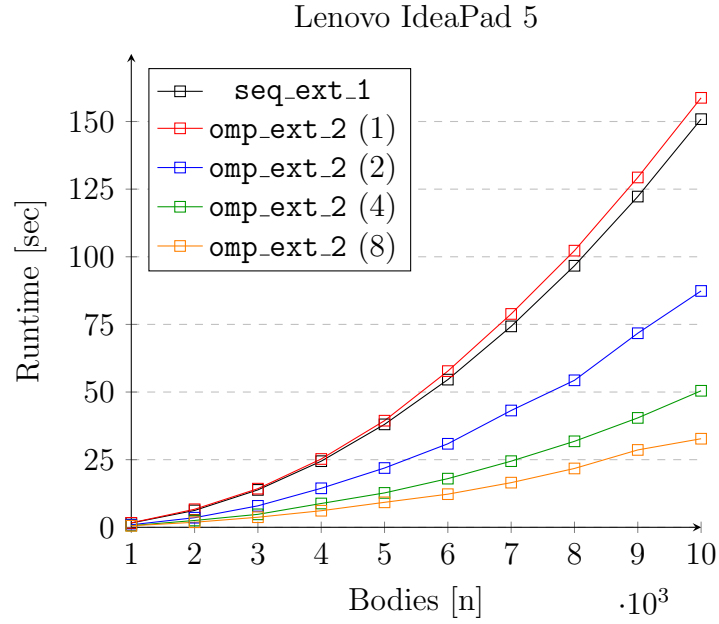


Figure 2.9: Confronto delle performance di `seq_ext_1` e `omp_ext_2` su 100 passi di simulazione da 1000 a 10000 corpi. I numeri tra parentesi nella legenda del grafico indicano il numero di thread attivi

### 2.2.3 Riduzione dell'overhead di OpenMP

Oltre a migliorare la gestione delle cache, è possibile **minimizzare l'overhead** imposto da OpenMPI. Difatti, nella versione `omp_ext_2`, le due direttive `#pragma omp for` vengono maneggiate dal pre-compilatore di OpenMP, venendo sostituite con dei pezzi di codice che vadano a calcolare il modo in cui vengono separati automaticamente gli indici del *ciclo for*.

Poiché nel programma sono presenti due direttive `#pragma omp for` situate all'interno del ciclo esterno che itera  $k$  volte sui passaggi della simulazione, tali suddivisioni vengono effettuate per un totale di  $2k$  volte.

Per ridurre tale overhead, è possibile manualmente a priori tale suddivisione prima del ciclo esterno, in modo che ogni thread sappia già su quale porzione di corpi andare ad eseguire i calcoli.

```
local_n = num_bodies / thread_count;
remainder = num_bodies % thread_count;

#pragma omp parallel num_threads(thread_count) \
    default(none) \
    shared(bodies, num_bodies, num_steps, out_fp, interval, local_n, remainder) \
    private(i, j, k, vel_i, pos_i, pos_j, dist_ij, scalar, \
        my_rank, my_slice_size, my_slice_start, my_slice_end)
{
    my_rank = omp_get_thread_num();

    my_slice_size = local_n;
    my_slice_start = my_rank * my_slice_size;

    if(my_rank < remainder){
        my_slice_size++;
        my_slice_start += my_rank
    }
    else
        my_slice_start += remainder;

    my_slice_end = my_slice_start + my_slice_size - 1;
```



```

for (k = 0; k < num_steps; ++k){
    for (i = my_slice_start; i <= my_slice_end; ++i) {

        vel_i = bodies[i].vel;
        pos_i = bodies[i].pos;

        for (j = 0; j < num_bodies; ++j) {
            if (i == j) continue;

            pos_j = bodies[j].pos;

            dist_ij = vector_distance(&pos_i, &pos_j);
            scalar = G * bodies[j].mass * (pow(dist_ij, -3.0)) * TIME_DELTA;

            vel_i.x += scalar * (pos_j.x - pos_i.x);
            vel_i.y += scalar * (pos_j.y - pos_i.y);
            vel_i.z += scalar * (pos_j.z - pos_i.z);
        }

        bodies[i].vel = vel_i;
    }

    #pragma omp barrier

    for (i = my_slice_start; i <= my_slice_end; ++i) {
        bodies[i].pos.x += bodies[i].vel.x * TIME_DELTA;
        bodies[i].pos.y += bodies[i].vel.y * TIME_DELTA;
        bodies[i].pos.z += bodies[i].vel.z * TIME_DELTA;
    }

    #pragma omp barrier

```

Figure 2.10: Screenshot del frammento di codice principale di `omp_ext_3.c`

Nonostante ciò, tale cambiamento **influisce poco** sulle prestazioni del programma. Difatti, rispetto alla versione precedente, notiamo come l'overhead **cresca** nel caso in cui vi siano pochi thread in utilizzo (circa il 15% più lento), mentre esso **diminuisca** nel caso in cui vi siano molti thread in utilizzo (circa il 10% più veloce).

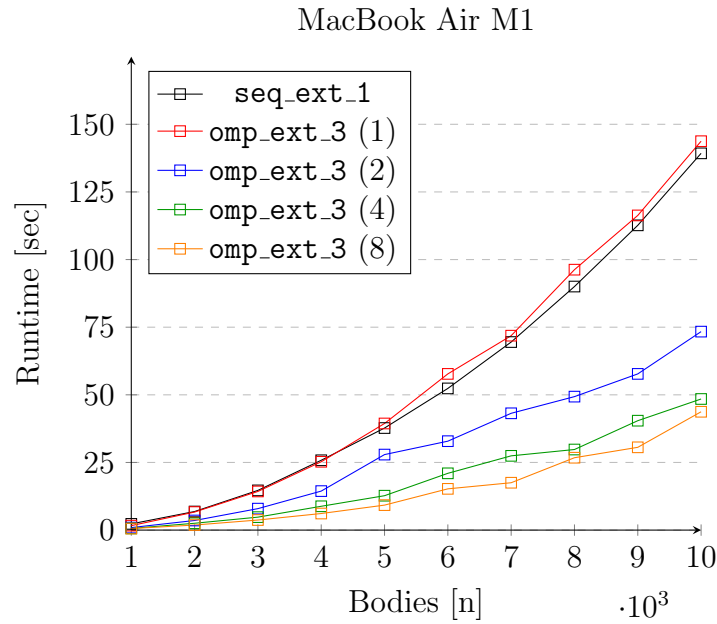
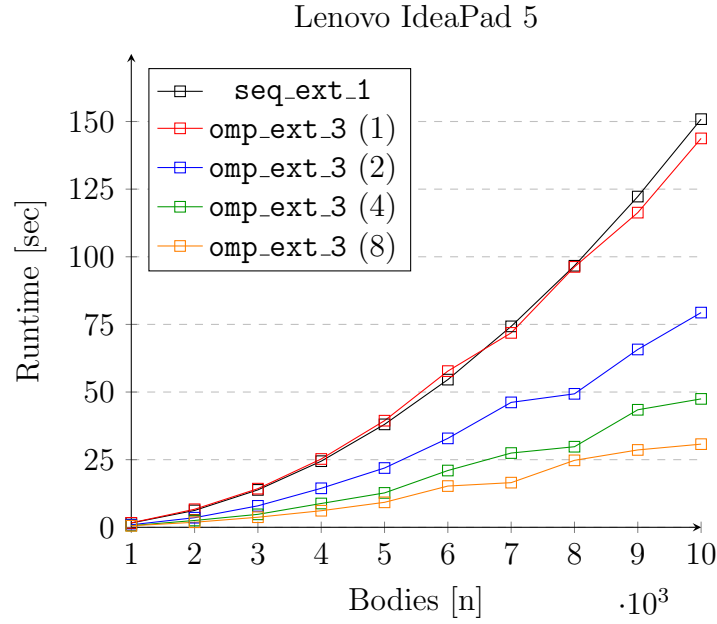


Figure 2.11: Confronto delle performance di `seq_ext_1` e `omp_ext_3` su 100 passi di simulazione da 1000 a 10000 corpi. I numeri tra parentesi nella legenda del grafico indicano il numero di thread attivi

Un'ulteriore tentativo di *iper-ottimizzazione* delle performance del programma prevede l'**eliminazione completa** del false sharing. Difatti, poiché anche i corpi sono memorizzati in modo sequenziale nell'array `bodies`, potrebbe verificarsi il caso in cui due corpi siano caricati all'interno dello stesso blocco di cache, creando potenzialmente del false sharing.

Per risolvere tale problema, è sufficiente *abbandonare* l'uso dello struct `bodies` e sostituire l'array `bodies` con tre array `masses`, `positions` e `velocities`. In tal modo, i valori dello stesso tipo (ossia massa, posizione e velocità) saranno adiacenti a loro, implicando che scrivere e leggere due tipi di valori contemporaneamente non vada a creare false sharing.

Tuttavia, tale modifica non **influisce**

```
for (k = 0; k < num_steps; ++k) {
    for (i = my_slice_start; i <= my_slice_end; ++i) {
        vel_i = velocities[i];
        pos_i = positions[i];

        for (j = 0; j < num_bodies; ++j) {
            if (i == j) continue;

            pos_j = positions[j];
            dist_ij = vector_distance(&pos_i, &pos_j);
            scalar = G * masses[j] * (pow(dist_ij, -3.0)) * TIME_DELTA;

            vel_i.x += scalar * (pos_j.x - pos_i.x);
            vel_i.y += scalar * (pos_j.y - pos_i.y);
            vel_i.z += scalar * (pos_j.z - pos_i.z);
        }
        velocities[i] = vel_i;
    }
    #pragma omp barrier

    for (i = my_slice_start; i <= my_slice_end; ++i) {
        positions[i].x += velocities[i].x * TIME_DELTA;
        positions[i].y += velocities[i].y * TIME_DELTA;
        positions[i].z += velocities[i].z * TIME_DELTA;
    }
    #pragma omp barrier
}
```

Figure 2.12: Screenshot del frammento di codice principale di `omp_ext_4.c`

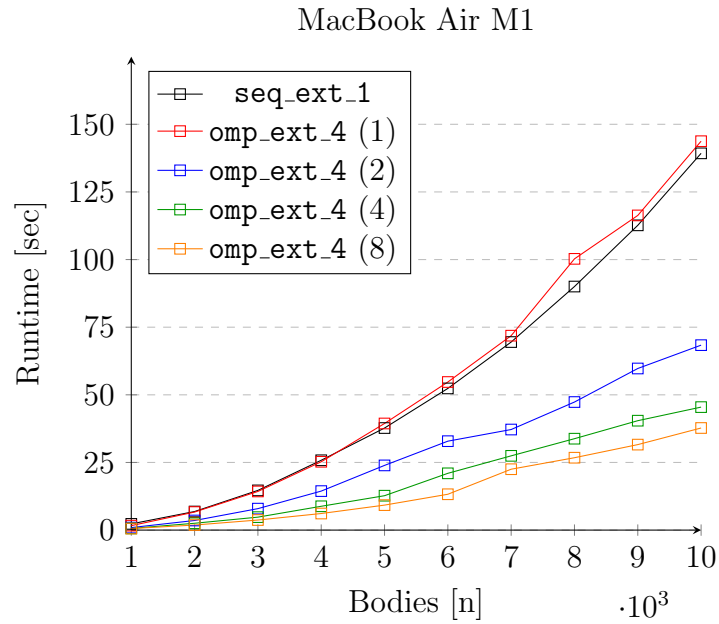
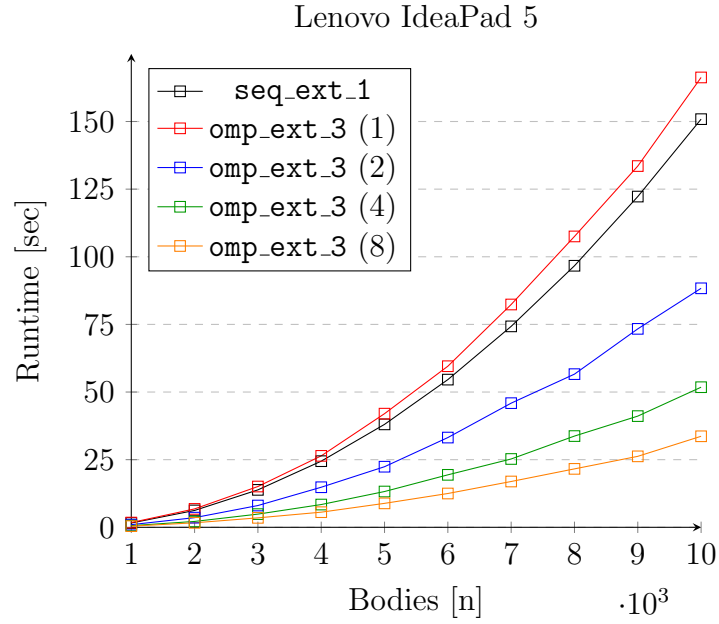


Figure 2.13: Confronto delle performance di `seq_ext_1` e `omp_ext_4` su 100 passi di simulazione da 1000 a 10000 corpi. I numeri tra parentesi nella legenda del grafico indicano il numero di thread attivi

## 2.3 Parallelizzazione tramite MPI

### 2.3.1 Modalità di implementazione

Tramite la libreria **MPI** è possibile parallelizzare il programma creando diversi processi separati (**memoria distribuita**) che comunicano tramite direttive di scambio di messaggi. Lavorando su zone di memoria separate si evita di incorrere in fenomeni come **race contion** o **false sharing** al costo di dover *impostare correttamente gli scambi* tra i processi e degli scambi stessi.

Un primo approccio per parallelizzare la versione estensiva prevede la suddivisione dei corpi su cui lavorare tra i processi generati per poi riunire i risultati.

Nel caso particolare in cui si dovesse effettuare **un solo step** della simulazione, è possibile far sì che un solo processo acquisisca i dati di tutti gli altri processi, restituendoli in output. In tal caso, risulta sufficiente far sì che il *master process* (ossia il processo 0) raccolga i dati calcolati, effettuando una comunicazione All-to-One. Per effettuare una simulazione di **più passi**, invece, tutti i processi hanno bisogno di conoscere i valori calcolati da ogni altro processo al passo precedente, richiedendo una comunicazione All-to-All.

Per tanto, ogni processo eseguirà le seguenti fasi:

1. Lettura del file contenente i dati in input
2. Identificazione dei propri corpi
3. Calcolo delle nuove posizioni
4. Condivisione All-to-All dei risultati
5. Ripetizione dei passi 3, 4 e 5 per il numero di passi della simulazione

Poiché la lettura del file **non causa problemi di parallelizzazione** e che tutti i processi hanno bisogno delle informazioni di tutti i corpi da esaminare, per **risparmiare una comunicazione broadcast**, tutti i processi leggono il file di input contemporaneamente riempiendo il loro vettore dei corpi.

Per quanto riguarda l'**identificazione dei corpi**, dato che ogni processo lavorerà su una sola frazione di corpi ma il vettore contiene gli stessi valori ordinati allo stesso modo tra tutti i processi, ogni processo dovrà calcolare la propria porzione di corpi su cui lavorerà e il displacement sull'array dal quale iniziare a lavorare, comunicandolo agli altri processi in modo che essi possano sincronizzarsi tra loro.

```
void compute_slices()
{
    int remainder;

    my_slice_size = num_bodies / comm_size;
    my_slice_start = my_rank * my_slice_size;
    remainder = num_bodies % comm_size;

    if (my_rank < remainder)
    {
        my_slice_size++;
        my_slice_start += my_rank;
    }
    else
        my_slice_start += remainder;

    my_slice_end = my_slice_start + my_slice_size - 1;

    count = (int *)malloc(comm_size * sizeof(int));
    disp = (int *)malloc(comm_size * sizeof(int));

    count[my_rank] = my_slice_size;
    disp[my_rank] = my_slice_start;

    MPI_Allgather(&my_slice_size, 1, MPI_INT, count, 1, MPI_INT, MPI_COMM_WORLD);
    MPI_Allgather(&my_slice_start, 1, MPI_INT, disp, 1, MPI_INT, MPI_COMM_WORLD);
}
```

Figure 2.14: Screenshot del frammento di codice di interesse di `mpi_ext_1.c`

Inoltre, per poter effettuare le comunicazioni, è necessario che ogni processo crei a priori i datatype `MPI_Vec3D` e `MPI_Body` del tutto equivalenti ai due struct utilizzati nella versione seriale, ma in grado di essere trattati dalle comunicazioni di MPI.

```

void create_MPI_datatypes(){
    /*
        AoB = Array of Blocks
        AoT = Array of Types
        AoD = Array of Displacements
    */
    Body dummy = {0, {0, 0, 0}, {0, 0, 0}};

    MPI_Aint a_addr, b_addr;

    // Vec3D contains 3 doubles
    int AoB_Vec3D[1] = {3};
    MPI_Datatype AoT_Vec3D[1] = {MPI_DOUBLE};
    MPI_Aint AoD_Vec3D[1] = {0};

    MPI_Type_create_struct(1, AoB_Vec3D, AoD_Vec3D, AoT_Vec3D, &MPI_Vec3D);
    MPI_Type_commit(&MPI_Vec3D);

    // Body contains 1 double and 2 Vec3D
    int AoB_Body[2] = {1, 2};
    MPI_Datatype AoT_Body[2] = {MPI_DOUBLE, MPI_Vec3D};

    // Find displacement from double to Vec3D
    MPI_Get_address(&dummy.mass, &a_addr);
    MPI_Get_address(&dummy.pos.x, &b_addr);
    MPI_Aint AoD_Body[2] = {0, b_addr - a_addr};

    MPI_Type_create_struct(2, AoB_Body, AoD_Body, AoT_Body, &MPI_Body);
    MPI_Type_commit(&MPI_Body);
}

```

Figure 2.15: Screenshot del frammento di codice di interesse di `mpi_ext_1.c`

Una volta calcolata la suddivisione, i *cicli for* vengono modificati per far sì che i processi svolgano solo i calcoli inerenti alla propria porzione di corpi. In fine solo il master process (rank = 0) scrive i risultati sull'output.

Tuttavia, poiché la suddivisione dei corpi tra i processi potrebbe essere **non equa**, non è possibile utilizzare la funzione `MPI_Allgheter`, rendendo necessario l'uso della funzione `MPI_Allgheterv`, la quale condivide oltre alle informazioni sul tipo di dato trasmesso e ricevuto anche le informazioni sul **numero di blocchi** e il relativo **displacement** sull'array di destinazione. In tal modo, non è necessario aggiungere del padding ai dati inviati per riempire eventuali *"buchi"*.

```

for (n = 0; n < num_steps; ++n){
    for (i = my_slice_start; i <= my_slice_end; ++i){
        for (j = 0; j < num_bodies; ++j){
            if (i == j) continue;

            dist_ij = vector_distance(&bodies[i].pos, &bodies[j].pos);
            scalar = G * bodies[j].mass * (pow(dist_ij, -3.0)) * TIME_DELTA;

            bodies[i].vel.x += scalar * (bodies[j].pos.x - bodies[i].pos.x);
            bodies[i].vel.y += scalar * (bodies[j].pos.y - bodies[i].pos.y);
            bodies[i].vel.z += scalar * (bodies[j].pos.z - bodies[i].pos.z);
        }
    }

    for (i = my_slice_start; i <= my_slice_end; ++i)
    {
        bodies[i].pos.x += bodies[i].vel.x * TIME_DELTA;
        bodies[i].pos.y += bodies[i].vel.y * TIME_DELTA;
        bodies[i].pos.z += bodies[i].vel.z * TIME_DELTA;
    }

    MPI_Allgather(bodies + my_slice_start, my_slice_size, MPI_Body,
        bodies, count, disp, MPI_Body, MPI_COMM_WORLD);

    if (n % interval == 0 && my_rank == 0){
        for (size_t i = 0; i < num_bodies; i++){
            fprintf(out_fp, "%lf,%lf,%lf,", bodies[i].pos.x, bodies[i].pos.y, bodies[i].pos.z);
        }

        fprintf(out_fp, "\n");
    }
}

```

Figure 2.16: Screenshot del frammento di codice di interesse di `mpi_ext_1.c`

A seguito dell'analisi precedente, dunque, possiamo stabilire che, per natura stessa dell'algoritmo, l'uso di una parallelizzazione a memoria distribuita risulta **inefficiente** rispetto a quella a memoria condivisa. Tale risultato si evince dall'elevata mole di comunicazioni richieste tra i vari processi (una per passo della simulazione), rallentando significativamente il programma. In particolare, all'aumentare dei processi coinvolti nella parallelizzazione, tali comunicazioni richiedono un **vasto overhead**.



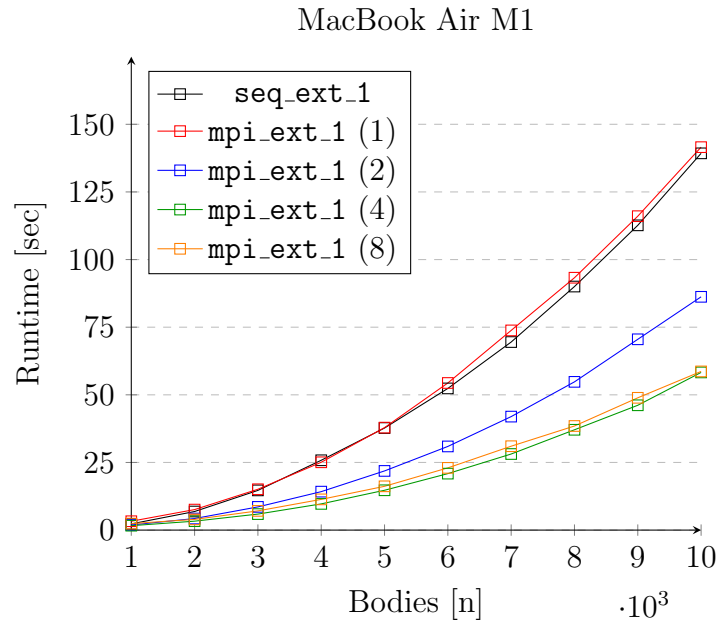
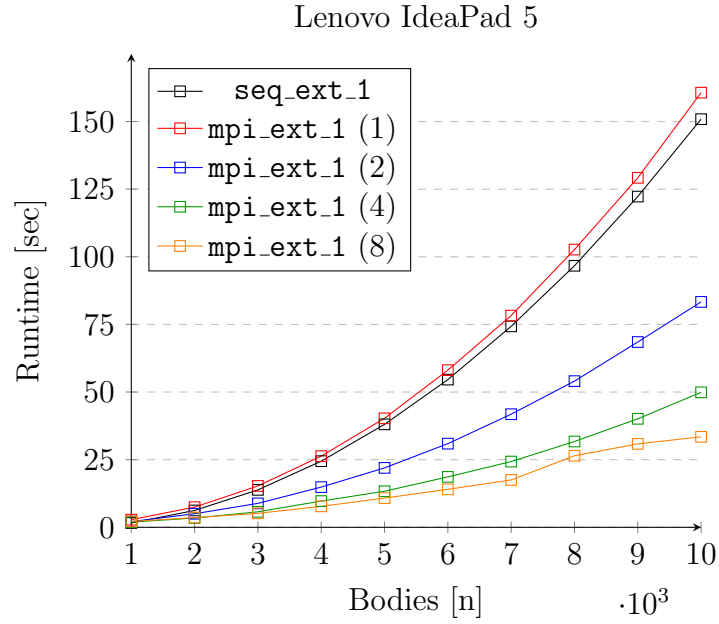


Figure 2.17: Confronto delle performance di `seq_ext_1` e `mpi_ext_1` su 100 passi di simulazione da 1000 a 10000 corpi. I numeri tra parentesi nella legenda del grafico indicano il numero di thread attivi

### 2.3.2 Riduzione e localizzazione dei dati trasmessi

Poiché le comunicazioni risultano essere il *bottleneck* della versione `mpi_ext_1`, risulta ovvio analizzare **modalità per alleggerire** tali comunicazioni riducendo la quantità dei dati trasmessi da esse.

In particolare, notiamo che al fine di poter eseguire il passo successivo della simulazione, è sufficiente che ogni processo trasmetta a tutti gli altri processi solo ed esclusivamente le proprie **nuove posizioni** in quanto le masse dei corpi rimangono invariate durante la simulazione e in quanto le nuove velocità dei corpi esterni alla porzione assegnata al processo stesso non siano rilevanti per il calcolo. Di conseguenza, i dati comunicati da ogni processo si riducono  $\frac{n}{p} \cdot 7$  double a solo  $\frac{n}{p} \cdot 3$  double.

```
for (n = 0; n < num_steps; ++n){
    for (i = my_slice_start; i <= my_slice_end; ++i){
        for (j = 0; j < num_bodies; ++j){

            if (i == j) continue;
            dist_ij = vector_distance(&positions[i], &positions[j]);
            scalar = G * masses[j] * (pow(dist_ij, -3.0)) * TIME_DELTA;

            velocities[i].x += scalar * (positions[j].x - positions[i].x);
            velocities[i].y += scalar * (positions[j].y - positions[i].y);
            velocities[i].z += scalar * (positions[j].z - positions[i].z);
        }
    }

    for (i = my_slice_start; i < my_slice_end; ++i){
        positions[i].x += velocities[i].x * TIME_DELTA;
        positions[i].y += velocities[i].y * TIME_DELTA;
        positions[i].z += velocities[i].z * TIME_DELTA;
    }

    MPI_Allgatherv(positions + my_slice_start, my_slice_size, MPI_Vec3D,
        positions, count, disp, MPI_Vec3D, MPI_COMM_WORLD);
}
```

Figure 2.18: Screenshot del frammento di codice di interesse di `mpi_ext_2.c`

Tuttavia, i test effettuati dimostrano che riducendo la dimensione del pacchetto trasmesso da ogni processo i tempi di comunicazione rimangono **per lo più invariati**, rendendo tale cambiamento utile solo superficialmente.

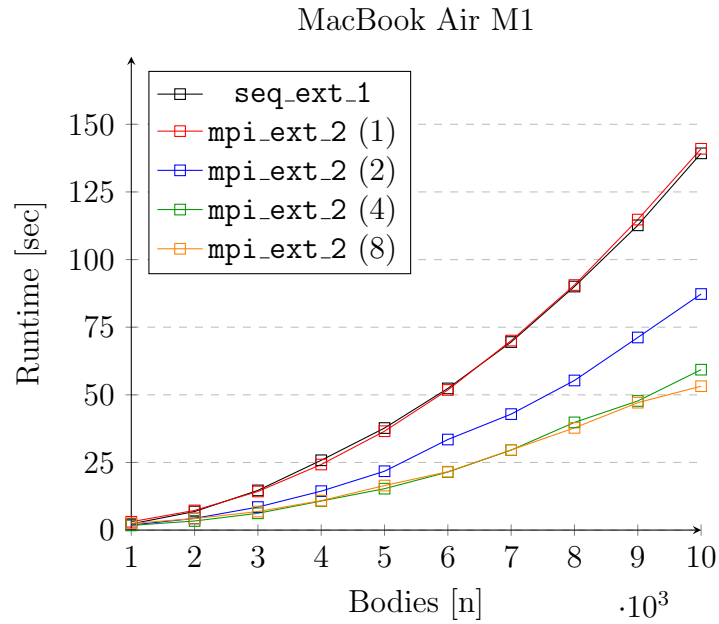
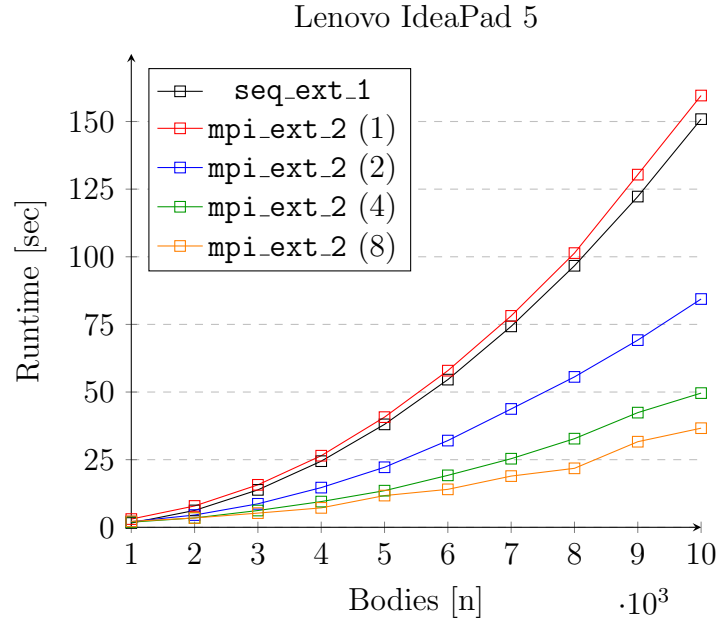


Figure 2.19: Confronto delle performance di `seq_ext_1` e `mpi_ext_2` su 100 passi di simulazione da 1000 a 10000 corpi. I numeri tra parentesi nella legenda del grafico indicano il numero di thread attivi

Nonostante ciò, è possibile ottenere un miglioramento delle prestazioni di circa il 15% riscrivendo le velocità dei corpi utilizzati all'interno di un **array più piccolo** e lavorando su quest'ultimo anziché sul vettore che contiene le informazione di tutti i corpi.

```
/*
 * each process saves only the speeds of its own bodies in a smaller local vector
 */
Vec3D localv[my_slice_size];
for (size_t i = 0; i < my_slice_size; i++)
{
    localv[i] = velocities[i + my_slice_start];
}

for (n = 0; n < num_steps; ++n)
{
    for (i = my_slice_start; i < my_slice_end; ++i)
    {
        for (j = 0; j < num_bodies; ++j)
        {
            if (i == j)
                continue;

            dist_ij = vector_distance(&positions[i], &positions[j]);
            scalar = G * masses[j] / (pow(dist_ij, 3.0)) * TIME_DELTA;

            localv[i - my_slice_start].x += scalar * (positions[j].x - positions[i].x);
            localv[i - my_slice_start].y += scalar * (positions[j].y - positions[i].y);
            localv[i - my_slice_start].z += scalar * (positions[j].z - positions[i].z);
        }
    }

    for (i = my_slice_start; i < my_slice_end; ++i)
    {
        positions[i].x += localv[i - my_slice_start].x * TIME_DELTA;
        positions[i].y += localv[i - my_slice_start].y * TIME_DELTA;
        positions[i].z += localv[i - my_slice_start].z * TIME_DELTA;
    }

    MPI_Allgatherv(positions + my_slice_start, my_slice_size, MPI_Vec3D,
        positions, count, disp, MPI_Vec3D, MPI_COMM_WORLD);
}
```

Figure 2.20: Screenshot del frammento di codice di interesse di `mpi_ext_3.c`

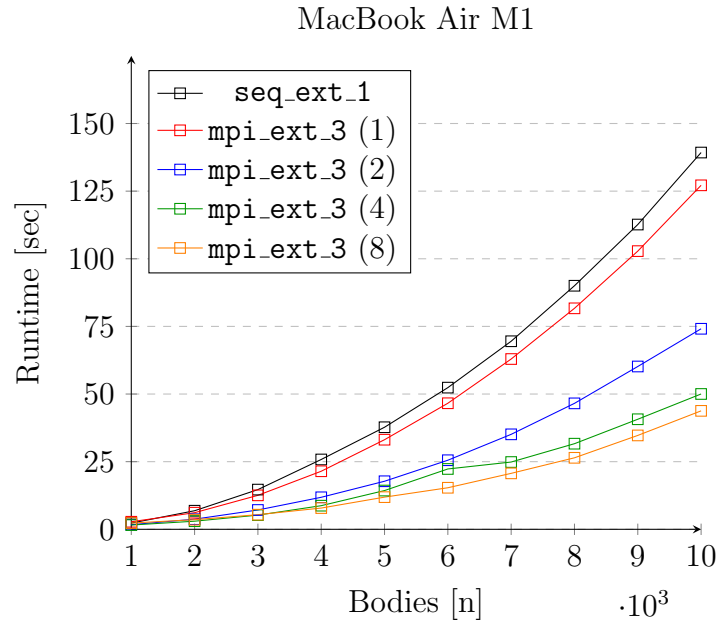


Figure 2.21: Confronto delle performance di `seq_ext_1` e `mpi_ext_3` su 100 passi di simulazione da 1000 a 10000 corpi. I numeri tra parentesi nella legenda del grafico indicano il numero di thread attivi

Tuttavia, tale miglioramento viene ottenuto solo ed esclusivamente sul dispositivo MacBook Air M1, mentre per il dispositivo Lenovo IdeaPad 5 i risultati rimangono ancora una volta **invariati**.

## 2.4 Analisi dello speed-up e dell'efficienza

Tramite i dati inerenti al runtime delle varie versioni parallelizzate e non mostrati nelle sezioni precedenti, consideriamo lo **speed-up** e l'**efficienza** ottenute da ognuna di tali versioni rispetto all'implementazione sequenziale `seq_ext_1` nel caso con 10000 corpi.

In particolare, per entrambi i dispositivi e per ogni modalità di implementazione parallela viene riportato il programma i cui runtime medi hanno formato tendenzialmente meglio, ossia i programmi `omp_ext_3` e `mpi_ext_3`.

Programma	Runtime	Speed-up	Efficienza
<code>seq_ext_1</code>	150.870804667	1.000000000	1.00
<code>omp_ext_3</code> (1)	166.075013638	0.908449750	0.90
<code>omp_ext_3</code> (2)	92.657269955	1.628267320	0.81
<code>omp_ext_3</code> (4)	50.705098152	2.975456318	0.74
<code>omp_ext_3</code> (8)	31.612117290	4.772562473	0.59
<code>mpi_ext_3</code> (1)	159.60092449	0.945300318	0.94
<code>mpi_ext_3</code> (2)	84.38146210	1.787961490	0.89
<code>mpi_ext_3</code> (4)	49.61076498	3.041090068	0.76
<code>mpi_ext_3</code> (8)	36.63834620	4.117838830	0.51

Figure 2.22: Statistiche per dispositivo Lenovo IdeaPad 5

Programma	Runtime	Speed-up	Efficienza
<code>seq_ext_1</code>	139.26667118	1.000000000	1.00
<code>omp_ext_3</code> (1)	142.753837109	0.975572174	0.97
<code>omp_ext_3</code> (2)	101.364573956	1.373918577	0.68
<code>omp_ext_3</code> (4)	56.463528633	2.466488981	0.61
<code>omp_ext_3</code> (8)	51.738933086	2.691719038	0.33
<code>mpi_ext_3</code> (1)	127.15266371	1.095271362	1.09
<code>mpi_ext_3</code> (2)	74.12223911	1.878878362	0.93
<code>mpi_ext_3</code> (4)	50.02179480	2.784119837	0.69
<code>mpi_ext_3</code> (8)	43.783582211	3.180796640	0.39

Figure 2.23: Statistiche per dispositivo MacBook Air M1

## Chapter 3

# Simulazione tramite algoritmo di Barnes-Hut

### 3.1 Descrizione dell'algoritmo

L'**algoritmo di Barnes-Hut** è uno dei metodi di approssimazione più utilizzato per problema degli  $n$  corpi. I punti nello spazio vengono considerati come un **piano cartesiano tridimensionale** i cui assi  $x, y$  e  $z$  rappresentano rispettivamente la larghezza (Ovest-Est), l'altezza (Sopra-Sotto) e la profondità (Nord-Sud) dei corpi

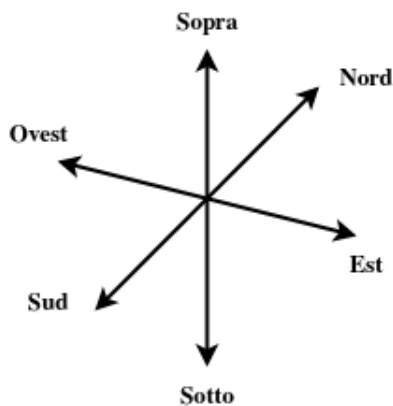


Figure 3.1: Suddivisione dello spazio nelle otto direzioni spaziali

Lo **spazio occupato dai corpi** viene considerato come un cubo  $\ell \times \ell \times \ell$ , dove  $\ell$  è il doppio della distanza massima tra l'origine dello spazio (ossia il punto  $(0,0,0)$ ) e gli  $n$  corpi.

Tale cubo viene **suddiviso in 8 sotto-cubi** di dimensione  $\frac{\ell}{2} \times \frac{\ell}{2} \times \frac{\ell}{2}$ . Tali cubi vengono indicati come una combinazione di **tre direzioni spaziali**:

- **Upper** e **Lower**, ossia tutti i cubi situati rispettivamente nella bisezione *superiore* e *inferiore* della coordinata  $y$
- **North** e **South**, ossia tutti i cubi situati rispettivamente nella bisezione *frontale* e *posteriore* della coordinata  $z$
- **West** e **East**, ossia tutti i cubi situati rispettivamente nella bisezione *sinistra* e *destra* della coordinata  $x$

In totale, dunque, i sotto-cubi sono identificati dai seguenti acronimi: UNW, UNE, USW, USE, LNW, LNE, LSW, LSE

Successivamente, se uno dei sotto-cubi contiene al suo interno più di un corpo, tale sotto-cubo viene **sezionato nuovamente**, ripetendo l'intero processo ricorsivamente finché ogni sotto-cubo non contiene un solo corpo.

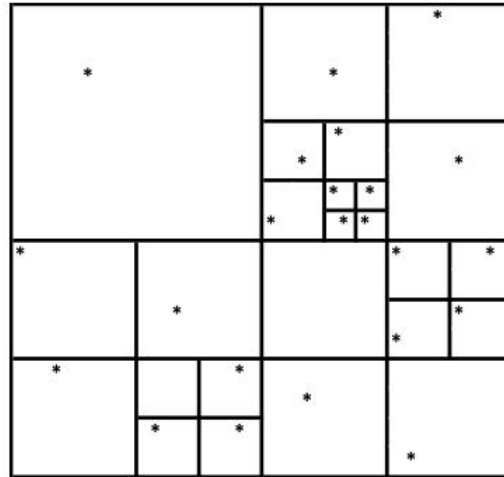


Figure 3.2: Esempio bidimensionale della suddivisione



Una volta suddiviso lo spazio, l'algoritmo procede calcolando gli **"pseudo-corpi"** di ogni sotto-cubo: se all'interno di un sotto-cubo vi sono  $k$  corpi, la posizione e la massa dello pseudo-corpo associato a tale sotto-cubo corrisponderanno rispettivamente al centro della massa e alla massa totale ricavate tramite i  $k$  corpi.

Ad esempio, se un sotto-cubo contiene i corpi  $b_1, \dots, b_4$ , il centro della massa del suo pseudo-corpo corrisponderà a:

$$\vec{c} = \frac{m_1 \vec{p}_1 + \dots + m_4 \vec{p}_4}{m_1 + \dots + m_4}$$

In particolare, notiamo che se un sotto-cubo contiene un solo corpo, allora il suo pseudo-corpo **coincide** con tale unico corpo.

Una volta calcolati tutti gli pseudo-corpi, l'algoritmo procede calcolando le interazioni tra ogni corpo. In particolare, tramite un valore  $\theta$  dettato a priori, tali interazioni possono essere **approssimate** con l'interazione con uno pseudo-corpo:

1. Sia  $b_i$  il corpo di cui attualmente si sta calcolando l'interazione
2. Partendo dal cubo originale, l'algoritmo calcola il rapporto  $\frac{s}{d}$ , dove  $s$  è il **lato** del cubo e  $d$  è la **distanza vettoriale** tra  $b_i$  e lo pseudo-corpo  $w$  associato a tale cubo
3. Se  $\frac{s}{d} < \theta$ , allora l'algoritmo approssimerà l'interazione tra  $b_i$  e tutti i corpi contenuti nel cubo calcolando direttamente l'interazione tra  $b_i$  e  $w$ . Altrimenti, l'algoritmo procederà ricorsivamente nei sotto-cubi.

Tali approssimazioni permettono di **risparmiare** una grande mole di calcoli, riducendo l'andamento asintotico del programma a  $O(n \log n)$ . È opportuno notare, inoltre, che il valore scelto per  $\theta$  risulti fondamentale:

- Se  $\theta = 0.0$ , l'algoritmo non approssimerà mai il calcolo, facendo degenerare l'intero algoritmo in una versione più complessa e lenta dell'implementazione estensiva della simulazione
- Se  $\theta = 1.0$ , le prestazioni dell'algoritmo risultano estremamente migliori rispetto all'implementazione estensiva, producendo tuttavia un'errore di approssimazione rilevante, ma tuttavia abbastanza trascurabile

- Se  $\theta \geq 1.25$ , l'errore generato dall'approssimazione risulta eccessivamente alto

In particolare, è necessario notare che, indipendentemente dal valore  $\theta$  scelto, l'errore risulta essere **inversamente proporzionale** al numero  $n$  di corpi coinvolti nella simulazione.

Dopo un'attenta analisi, è stato scelto di utilizzare il **valore**  $\theta = 0.5$ , in quanto l'incremento delle performance risulti non molto distante da quello ottenuto con il valore  $\theta = 1.0$ , presentando tuttavia un'errore pari a 0.0001% nella maggior parte delle simulazioni effettuate.

## 3.2 Implementazione sequenziale

### 3.2.1 Struttura dati e funzioni principali

Tramite la descrizione della sezione precedente, l'implementazione sequenziale dell'algoritmo viene interamente ridotta all'implementazione di una **struttura dati** che possa permettere di svolgere le varie operazioni efficientemente.

In particolare, per rappresentare la suddivisione ricorsiva dello spazio occupato dai corpi, risulta intuitivo l'uso di un **oct-tree**, ossia un albero i cui nodi possono possedere fino ad 8 *figli*. Tali figli vengono memorizzati tramite un array di 8 puntatori ad altri nodi, dove ogni indice rappresenta uno specifico figlio:

0	1	2	3	4	5	6	7
UNW	UNE	USW	USE	LNW	LNE	LSW	LSE

Oltre ai puntatori ai figli, ogni nodo dovrà memorizzare anche informazioni inerenti ai **confini** dello spazio rappresentato dal sotto-cubo (*bounding box*), informazioni inerenti al proprio **pseudo-corpo** associato e informazioni inerenti all'attuale **modalità**, la quale descrive lo stato attuale del sotto-cubo da esso rappresentato, ossia: *vuoto*, *singolo corpo* o *suddiviso*.

```

typedef struct bounding_box {

    /* Cube data */
    double min_x, mid_x, max_x,
           min_y, mid_y, max_y,
           min_z, mid_z, max_z,
           side;
} BoundingBox;

typedef struct barnes_hut_octree {

    /* != NULL only if status = 1*/
    Body* body;

    /* Space section limits */
    BoundingBox box;

    // 0 = Empty, 1 = Single body, 2 = Subdivided
    u8 status;

    /* Pseudo-body data */
    double total_mass;
    Vec3D center_of_mass;

    /*
       Each index represents a sub-section:

       | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
       -----
       | UNW | UNE | USW | USE | LNW | LNE | LSW | LSE |
    */
    struct barnes_hut_octree* sub_sections[8];
} BHOctree;

```

Figure 3.3: Screenshot del frammento di codice di interesse di `bh_octree_1.h`

Per **generare** l'albero, è necessario calcolare quale sia la *distanza massima* tra i corpi e il punto  $(0, 0, 0)$ , in modo da poter creare il primo nodo dell'albero corrispondente al cubo iniziale, per poi successivamente *inserire* uno ad uno tutti i corpi ed infine calcolare gli *pseudo-corpi*.

```

init_half_side = 0;

/* Find maximum distance from origin in every coordinate */
for(size_t i = 0; i < num_bodies; ++i){
    if(fabs(bodies[i].pos.x) > init_half_side)    init_half_side = fabs(bodies[i].pos.x);
    if(fabs(bodies[i].pos.y) > init_half_side)    init_half_side = fabs(bodies[i].pos.y);
    if(fabs(bodies[i].pos.z) > init_half_side)    init_half_side = fabs(bodies[i].pos.z);
}

/* Create initial cube */
box = init_box(-init_half_side, init_half_side, -init_half_side,
               init_half_side, -init_half_side, init_half_side,
               2*init_half_side);

bho = init_bhootree(box);

for(size_t i = 0; i < num_bodies; ++i){
    insert_body(bho, &bodies[i]);
}

compute_pseudo_bodies(bho);

return bho;

```

Figure 3.4: Screenshot del frammento di codice della funzione `generate_bodies_bho()` presente in `bh_octree_1.c`

Di fondamentale importanza risulta essere la procedura di **inserimento dei corpi**. Trattandosi della funzione principale per la costruzione dell'albero, è necessario che essa risulti essere il più efficiente possibile:

- Se il nodo in cui sta venendo inserito il corpo è nello stato **vuoto**, esso verrà impostato sullo stato *singolo corpo*, conservando il puntatore al corpo inserito
- Se il nodo in cui sta venendo inserito il corpo è nello stato **singolo corpo**, esso verrà impostato sullo stato *suddiviso*, per poi re-inserire in se stesso il corpo precedentemente conservato e il nuovo corpo

- Se il nodo in cui sta venendo inserito il corpo è nello stato **suddiviso**, verrà calcolata la sotto-sezione in cui inserire il corpo, ossia l'indice del figlio su cui propagare l'inserimento. Nel caso in cui il figlio non sia stato ancora creato, verrà calcolato il *bounding box* corrispondente per poi creare suddetto figlio ed effettuare l'inserimento

```
switch(bho->status){
    case EMPTY:
        bho->status = SINGLE_BODY;
        bho->body = body;
        break;

    case SINGLE_BODY:
        // Re-insert bodies treating the section as subdivided
        bho->status = SUBDIVIDED;
        tmp = bho->body;
        bho->body = NULL;
        insert_body(bho, tmp);
        insert_body(bho, body);
        break;

    default:
        /* Compute sub-section in which the body will be inserted */
        box = bho->box;
        U = (body->pos.y < box.mid_y) ? 1 : 0; // Upper/Down section
        N = (body->pos.z < box.mid_z) ? 1 : 0; // North/South section
        W = (body->pos.x < box.mid_x) ? 1 : 0; // West/East section
        section = U * 4 + N * 2 + W * 1;

        if(bho->sub_sections[section] != NULL){
            insert_body(bho->sub_sections[section], body);
        }
        else{
            sub_bho = init_bhoctree(init_subbox(box, U, N, W));
            bho->sub_sections[section] = sub_bho;
            insert_body(sub_bho, body);
        }
}
```

Figure 3.5: Screenshot del frammento di codice della funzione `insert_body()` presente in `bh_octree_1.c`

Per quanto riguarda il **calcolo degli pseudo-corpi**, le proprietà algebriche delle formule mostrate nella sezione precedente permettono di ottimizzare notevolmente tale calcolo poiché lo pseudo-corpo di un nodo può essere calcolato tramite gli pseudo-corpi dei suoi figli, implicando che l'intero calcolo possa essere svolto tramite una **visita post-order**.

**Esempio:**

- Supponiamo che la sezione  $A$  contenga al suo interno le sezioni  $B, C, D$
- Supponiamo inoltre che  $A$  contenga i corpi  $b_1, b_2, b_3, b_4$ , che  $B$  contenga i corpi  $b_2, b_3$ , che  $C$  contenga il corpo  $b_4$  e che  $D$  contenga il corpo  $b_1$
- Il centro della massa e la massa totale di  $D$ , ossia  $\vec{c}_D$  e  $m_D$ , corrispondono a  $\vec{p}_1$  e  $m_1$  poiché  $b_1$  risulta essere l'unico corpo al suo interno
- Il centro della massa e la massa totale di  $C$ , ossia  $\vec{c}_C$  e  $m_C$ , corrispondono a  $\vec{p}_4$  e  $m_4$  poiché  $b_4$  risulta essere l'unico corpo al suo interno
- Il centro della massa e la massa totale di  $B$ , ossia  $\vec{c}_B$  e  $m_B$ , corrispondono a:

$$\vec{c}_B = \frac{m_2 \vec{p}_2 + m_3 \vec{p}_3}{m_2 + m_3}$$

$$m_B = m_2 + m_3$$

- Il centro della massa e la massa totale di  $A$ , ossia  $\vec{c}_A$  e  $m_A$ , corrispondono a:

$$\vec{c}_A = \frac{m_1 \vec{p}_1 + m_2 \vec{p}_2 + m_3 \vec{p}_3 + m_4 \vec{p}_4}{m_1 + m_2 + m_3 + m_4}$$

$$m_A = m_1 + m_2 + m_3 + m_4$$

- Notiamo quindi che:

$$\vec{c}_A = \frac{m_1 \vec{p}_1 + m_2 \vec{p}_2 + m_3 \vec{p}_3 + m_4 \vec{p}_4}{m_1 + m_2 + m_3 + m_4} = \frac{m_D \vec{c}_D + m_B \vec{c}_B + m_C \vec{c}_C}{m_D + m_B + m_C}$$

$$m_A = m_1 + m_2 + m_3 + m_4 = m_D + m_B + m_C$$

```

switch(bho->status){
    case EMPTY:
        bho->total_mass = 0;
        bho->center_of_mass.x = 0;
        bho->center_of_mass.y = 0;
        bho->center_of_mass.z = 0;

        break;

    case SINGLE_BODY:
        bho->total_mass = bho->body->mass;
        bho->center_of_mass = bho->body->pos;

        break;

    default:
        total_mass = 0;
        center_of_mass.x = 0;
        center_of_mass.y = 0;
        center_of_mass.z = 0;

        for (int i = 0; i < 8; ++i){
            sub_bho = bho->sub_sections[i];

            if(sub_bho){ // if != NULL
                compute_pseudo_bodies(sub_bho);

                total_mass += sub_bho->total_mass;
                center_of_mass.x += sub_bho->center_of_mass.x * sub_bho->total_mass;
                center_of_mass.y += sub_bho->center_of_mass.y * sub_bho->total_mass;
                center_of_mass.z += sub_bho->center_of_mass.z * sub_bho->total_mass;
            }
        }

        bho->total_mass = total_mass;

        if(total_mass == 0){
            bho->center_of_mass.x = 0;
            bho->center_of_mass.y = 0;
            bho->center_of_mass.z = 0;
        }
        else{
            bho->center_of_mass.x = center_of_mass.x / total_mass;
            bho->center_of_mass.y = center_of_mass.y / total_mass;
            bho->center_of_mass.z = center_of_mass.z / total_mass;
        }

        break;
}

```

Figure 3.6: Screenshot del frammento di codice della funzione `compute_pseudo_bodies()` presente in `bh_octree_1.c`

Una volta costruito l'albero e calcolati gli pseudo-corpi, l'algoritmo procede con il calcolare l'**aggiornamento delle velocità**. Dato il corpo  $b_i$  attualmente considerato, viene computato se approssimare il calcolo o no tramite il rateo di  $\frac{s}{d}$  e il valore  $\theta$  tramite una **visita post-order**.

```
new_vel.x = 0;
new_vel.y = 0;
new_vel.z = 0;

if(!bho || bho->status == EMPTY || (bho->status == SINGLE_BODY && bho->body == body))
    return new_vel;

/* If the node contains a single body, that body is the center of mass */
dist = vector_distance(&body->pos, &bho->center_of_mass);

if(bho->status != SINGLE_BODY ){
    ratio = bho->box.side / dist;

    /* Do not approximate */
    if(ratio >= theta){
        for(int i = 0; i < 8; ++i){
            if(!bho->sub_sections[i] || bho->sub_sections[i]->status == EMPTY)
                continue;

            sub_vel = compute_velocity_increment(bho->sub_sections[i], body, G, theta);

            new_vel.x += sub_vel.x;
            new_vel.y += sub_vel.y;
            new_vel.z += sub_vel.z;
        }

        return new_vel;
    }
}

/* Approximate */

scalar = G * bho->total_mass * (pow(dist, -3.0));
new_vel.x += scalar * (bho->center_of_mass.x - body->pos.x);
new_vel.y += scalar * (bho->center_of_mass.y - body->pos.y);
new_vel.z += scalar * (bho->center_of_mass.z - body->pos.z);

return new_vel;
```

Figure 3.7: Screenshot del frammento di codice della funzione `compute_velocity_increment()` presente in `bh_octree_1.c`



Una volta definite quelle che sono le funzioni principali, l'intero algoritmo può essere ridotto all'applicazione in sequenza di suddette **tre procedure** ad ogni passo della simulazione, **distruggendo** l'albero dell'iterazione precedente, **ricalcolando** un nuovo albero tramite le nuove posizioni dei corpi e infine **computando** le nuove posizioni.

```
for (n = 0; n < num_steps; ++n){

    bho = generate_bodies_bho(bodies, num_bodies);

    for (i = 0; i < num_bodies; ++i) {
        sum_vel = compute_velocity_increment(bho, &bodies[i], G, theta);
        bodies[i].vel.x += sum_vel.x * TIME_DELTA;
        bodies[i].vel.y += sum_vel.y * TIME_DELTA;
        bodies[i].vel.z += sum_vel.z * TIME_DELTA;
    }

    for (i = 0; i < num_bodies; ++i) {
        bodies[i].pos.x += bodies[i].vel.x * TIME_DELTA;
        bodies[i].pos.y += bodies[i].vel.y * TIME_DELTA;
        bodies[i].pos.z += bodies[i].vel.z * TIME_DELTA;
    }

    destroy_bhoctree(bho);
```

Figure 3.8: Screenshot del frammento di codice di interesse di `seq_bh_1.c`

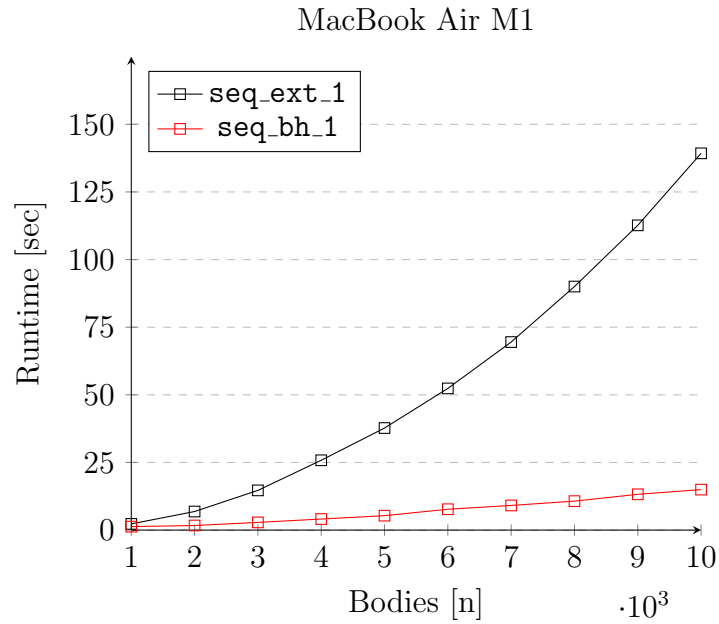
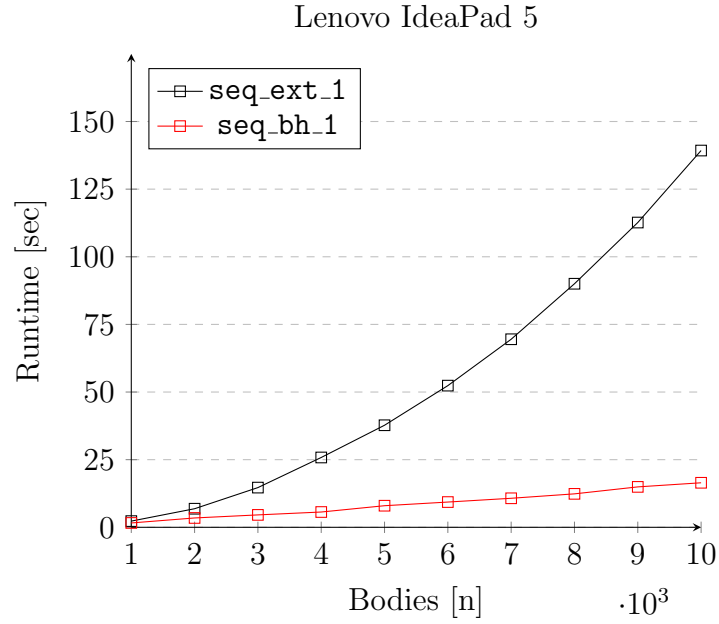


Figure 3.9: Confronto delle performance di `seq_ext_1` e `seq_bh_1` con  $\theta = 0.5$  su 100 passi di simulazione da 1000 a 10000 corpi. I numeri tra parentesi nella legenda del grafico indicano il numero di thread attivi

### 3.2.2 Ottimizzazioni della struttura dati

Nella versione `seq_bh_1.c`, potrebbe verificarsi che, a seguito dell'aggiornamento della sua posizione, un corpo **esca fuori** dal *bounding box* del proprio nodo. In tal caso, risulta necessario che l'albero venga **distrutto** e **ricalcolato** al fine di poter aggiornare dove siano situati i corpi al suo interno.

Tuttavia, tale operazione richiede la deallocazione e riallocazione dell'intero albero ad ogni passo della simulazione, generando un elevato rallentamento del programma, senza contare l'aggiuntivo rallentamento dato dal ricalcolo dell'intero albero, il quale spesso risulta *non necessario* nel caso in cui nessun corpo sia uscito dal *bounding box* del proprio nodo.

Di conseguenza, è necessario ottimizzare l'intera struttura dati in modo che essa possa essere **aggiornata** senza essere interamente ricalcolata. Per raggiungere tale obiettivo, viene utilizzato un array di appoggio **leaves**, il quale memorizza per ogni corpo il puntatore al nodo foglia in cui è situato il corpo stesso. Tramite tale array aggiuntivo, è possibile definire la seguente procedura di aggiornamento:

1. Dato il corpo  $b_i$ , viene **verificato** se tale corpo sia uscito al di fuori del *bounding box* del proprio nodo foglia
2. Se ciò risulta falso, il controllo procede con il prossimo corpo
3. Se ciò risulta vero, l'albero viene percorso **dal basso verso l'alto** a partire dalla foglia contenente il nodo, procedendo finché non viene trovato un nodo sul cammino dalla foglia alla radice il cui *bounding box* **contenga ancora il corpo**. Una volta trovato tale nodo, viene effettuato l'**inserimento** del corpo in tale nodo.
4. In particolare, se neanche il *bounding box* della radice dell'albero contiene il corpo, dunque se il corpo è uscito fuori da quello che precedentemente era lo spazio massimo, l'intero albero verrà **completamente rigenerato**, come nella versione `seq_bh_1`, espandendo così lo spazio massimo occupato dai corpi

Per realizzare tale procedura efficientemente, è necessario che ogni nodo memorizzi anche un puntatore al proprio **nodo padre** e l'**indice del proprio sotto-settore** in cui tale nodo è situato all'interno dell'array dei figli del

padre. Ad esempio, se il nodo *A* corrisponde al figlio 4 del proprio padre, allora tale indice verrà impostato su 4. In tal modo, è possibile evitare di dover iterare su tutti i figli del padre in cerca di quale sia il sotto-settore in cui è situato il nodo stesso.

```
typedef struct barnes_hut_octree {
    /* != NULL only if status = 1*/
    Body* body;

    /* Space section limits */
    BoundingBox box;

    // 0 = Empty, 1 = Single body, 2 = Subdivided
    u8 status;

    /* Pseudo-body data */
    double total_mass;
    Vec3D center_of_mass;

    /*
       Each index represents a sub-section:

       | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
       -----
       | UNW | UNE | USW | USE | LNW | LNE | LSW | LSE |
    */
    struct barnes_hut_octree* sub_sections[8];

    /* Optimized-update data*/
    struct barnes_hut_octree** leaf;
    struct barnes_hut_octree* parent;
    unsigned int parent_sub_index;
} BHOctree;
```

Figure 3.10: Screenshot del frammento di codice di interesse di `bh_octree_2.c`

```

new_vel.x = 0;
new_vel.y = 0;
new_vel.z = 0;

if(!bho || bho->status == EMPTY || (bho->status == SINGLE_BODY && bho->body == body))
    return new_vel;

/* If the node contains a single body, that body is the center of mass */
dist = vector_distance(&body->pos, &bho->center_of_mass);

if(bho->status != SINGLE_BODY ){
    ratio = bho->box.side / dist;

    /* Do not approximate */
    if(ratio >= theta){
        for(int i = 0; i < 8; ++i){
            if(!bho->sub_sections[i] || bho->sub_sections[i]->status == EMPTY)
                continue;

            sub_vel = compute_velocity_increment(bho->sub_sections[i], body, G, theta);

            new_vel.x += sub_vel.x;
            new_vel.y += sub_vel.y;
            new_vel.z += sub_vel.z;
        }

        return new_vel;
    }
}

/* Approximate */

scalar = G * bho->total_mass * (pow(dist, -3.0));
new_vel.x += scalar * (bho->center_of_mass.x - body->pos.x);
new_vel.y += scalar * (bho->center_of_mass.y - body->pos.y);
new_vel.z += scalar * (bho->center_of_mass.z - body->pos.z);

return new_vel;

```

Figure 3.11: Screenshot del frammento di codice della funzione `update_bodies_bho()` presente in `bh_octree_2.c`

```

leaves = malloc(sizeof(BHOctree*) * num_bodies);
bho = generate_bodies_bho(bodies, leaves, num_bodies);

for (k = 0; k < num_steps; ++k) {

    for (i = 0; i < num_bodies; ++i) {
        sum_vel = compute_velocity_increment(bho, &bodies[i], G, theta);
        bodies[i].vel.x += sum_vel.x * TIME_DELTA;
        bodies[i].vel.y += sum_vel.y * TIME_DELTA;
        bodies[i].vel.z += sum_vel.z * TIME_DELTA;
    }

    for (i = 0; i < num_bodies; ++i) {
        bodies[i].pos.x += bodies[i].vel.x * TIME_DELTA;
        bodies[i].pos.y += bodies[i].vel.y * TIME_DELTA;
        bodies[i].pos.z += bodies[i].vel.z * TIME_DELTA;
    }

    bho = update_bodies_bho(bho, bodies, leaves, num_bodies);

    if (k % interval == 0) {
        for (size_t i = 0; i < num_bodies; i++) {
            fprintf(out_fp, "%lf,%lf,%lf,", bodies[i].pos.x, bodies[i].pos.y, bodies[i].pos.z);
        }

        fprintf(out_fp, "\n");
    }
}

destroy_bhooctree(bho);

```

Figure 3.12: Screenshot del frammento di codice di interesse di `seq_bh_2.c`

Nel caso di molti corpi, tali parametri aggiuntivi incidono molto sulla **memoria totale occupata** dal programma nel suo intero poiché, dati  $n$  corpi, l'albero può avere un massimo di  $8n$  nodi istanziati contemporaneamente. Tuttavia, tale memoria aggiuntiva risulta trascurabile rispetto **netto miglioramento** delle performance del programma.

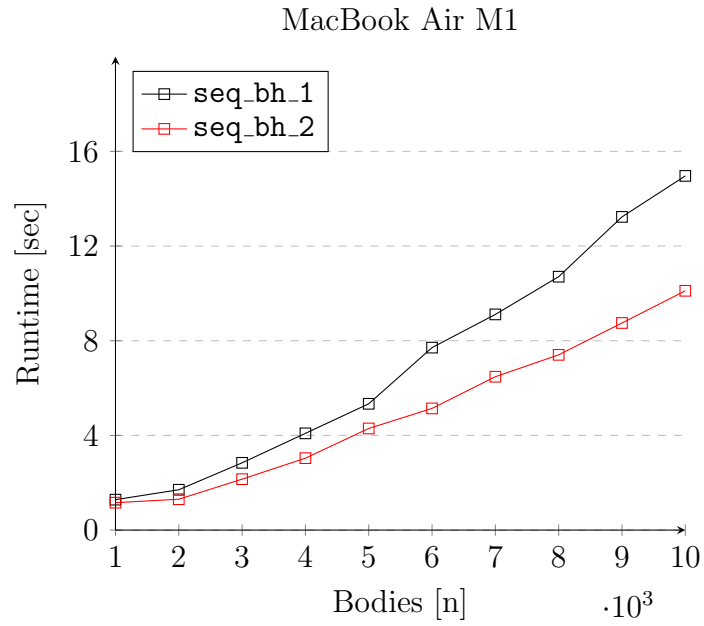
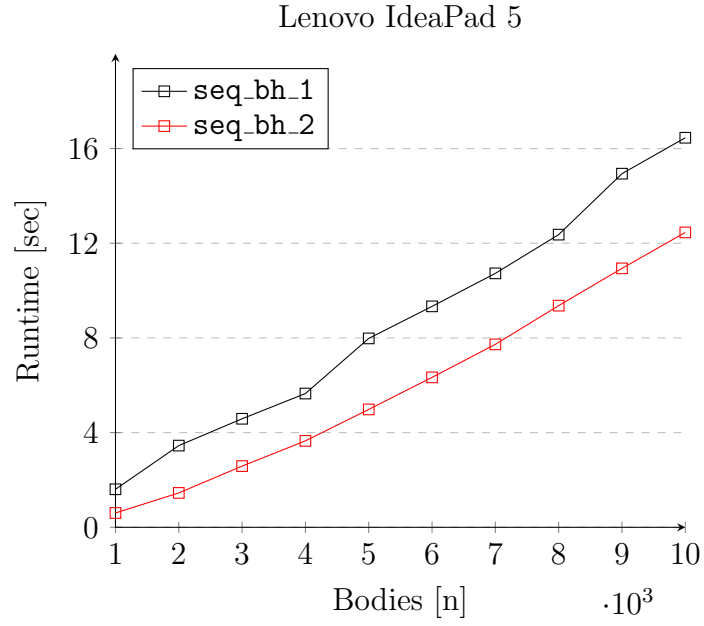


Figure 3.13: Confronto delle performance di `seq_bh_1` e `seq_bh_2` con  $\theta = 0.5$  su 100 passi di simulazione da 1000 a 10000 corpi. I numeri tra parentesi nella legenda del grafico indicano il numero di thread attivi

## 3.3 Parallelizzazione tramite OpenMP

### 3.3.1 Analisi, limitazioni riscontrate e implementazione

Come per l'implementazione estensiva, in particolare la versione `omp_ext_3`, il modo più diretto per poter parallelizzare l'algoritmo tramite OpenMP risulta essere tramite la **suddivisione a priori** dei corpi con cui i thread vanno a calcolare le varie interazioni.

Tuttavia, a differenza dell'implementazione estensiva, l'algoritmo di Barnes-Hut presenta una problematica aggiuntiva, ossia la **gestione dell'albero**: trattandosi di una struttura dati condivisa, risulta evidente pensare a problematiche relative all'**inserimento simultaneo dei nodi**, sia durante la creazione che durante l'aggiornamento.

Per gestire tale problema, sono state considerate le seguenti soluzioni:

1. Implementare un **lock sulla radice** dell'albero, in modo che solo un thread per volta possa accedere all'intera struttura dati.
2. Implementare un **lock su ogni nodo** dell'albero, in modo che solo un thread per volta possa accedere al nodo stesso
3. Implementare un **optimistic locking** prima di effettuare modifiche ai nodi durante un inserimento
4. Far sì che **solo un processo** si occupi di generare ed aggiornare l'albero mentre tutti gli altri attendono

Dopo un'attenta analisi e prove di implementazione, è stato scelto di implementare l'**ultima soluzione**, in quanto le altre sono state ritenute come particolarmente **inefficienti** per i seguenti motivi:

1. Utilizzando un **lock sulla radice**, ogni thread dovrebbe attendere che il thread precedente termini tutti i propri inserimenti, rendendo difatti l'intera operazione del tutto identica all'ultima soluzione, tuttavia con un overhead aggiunto dovuto alla coordinazione tra i processi e alla gestione del lock



2. Utilizzando un **lock su ogni nodo**, le performance di ogni operazione verrebbero drasticamente peggiorate in quanto ogni inserimento può propagarsi sull'intero albero, implicando che ogni inserimento debba richiedere e cedere continuamente il lock di ogni nodo attraversato nel cammino percorso
3. Utilizzando l'**optimistic locking**, le performance di ogni operazione verrebbero drasticamente peggiorate in quanto ogni inserimento può generare modifiche a catena sui vari nodi attraversati, implicando che sia necessario effettuare la doppia scansione prevista dall'optimistic locking per ognuna di tali modifiche, rendendo il costo di un singolo inserimento pari a  $O(n^2)$

```

leaves = malloc(sizeof(BHOctree*) * num_bodies);
bho = generate_bodies_bho(bodies, leaves, num_bodies);

local_n = num_bodies / thread_count;
remainder = num_bodies % thread_count;

#pragma omp parallel num_threads(thread_count) \
    default(none) \
    shared(bodies, bho, leaves, num_bodies, num_steps, out_fp, \
        G, theta, interval, local_n, remainder) \
    private(i, k, sum_vel, my_rank, my_slice_size, my_slice_start, my_slice_end)
{
    my_rank = omp_get_thread_num();
    my_slice_size = local_n;
    my_slice_start = my_rank * my_slice_size;

    if(my_rank < remainder){
        my_slice_size++;
        my_slice_start += my_rank;
    }
    else my_slice_start += remainder;

    my_slice_end = my_slice_start + my_slice_size - 1;

    for (k = 0; k < num_steps; ++k){
        for (i = my_slice_start; i <= my_slice_end; ++i) {
            sum_vel = compute_velocity_increment(bho, &bodies[i], G, theta);
            bodies[i].vel.x += sum_vel.x * TIME_DELTA;
            bodies[i].vel.y += sum_vel.y * TIME_DELTA;
            bodies[i].vel.z += sum_vel.z * TIME_DELTA;
        }

        #pragma omp barrier
    }
}

```

```

    for (i = my_slice_start; i <= my_slice_end; ++i) {
        bodies[i].pos.x += bodies[i].vel.x * TIME_DELTA;
        bodies[i].pos.y += bodies[i].vel.y * TIME_DELTA;
        bodies[i].pos.z += bodies[i].vel.z * TIME_DELTA;
    }

    #pragma omp barrier

    #pragma omp single
    bho = update_bodies_bho(bho, bodies, leaves, num_bodies);

    if (k % interval == 0){
        #pragma omp single
        {
            for (size_t i = 0; i < num_bodies; i++){
                fprintf(out_fp, "%lf,%lf,%lf,", bodies[i].pos.x, bodies[i].pos.y, bodies[i].pos.z);
            }

            fprintf(out_fp, "\n");
        }
    }
}

destroy_bhoctree(bho);

```

Figure 3.14: Screenshot del frammento di codice di interesse di `omp_bh.c`

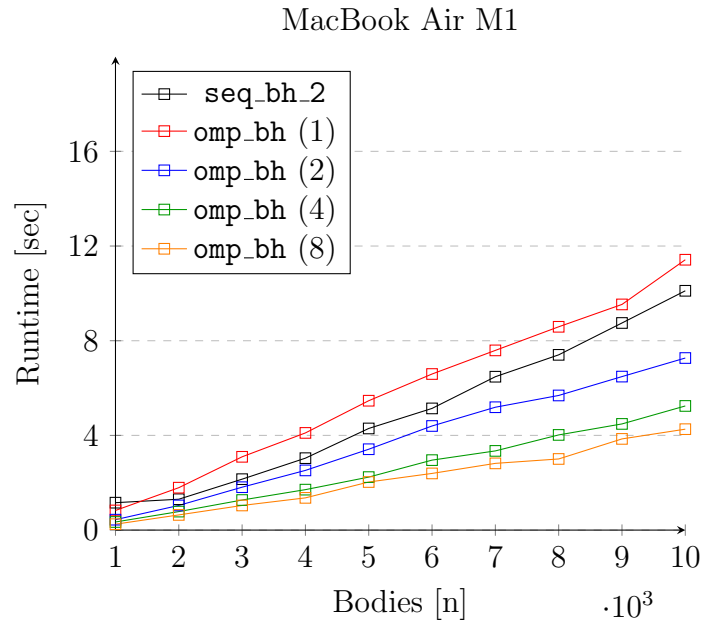
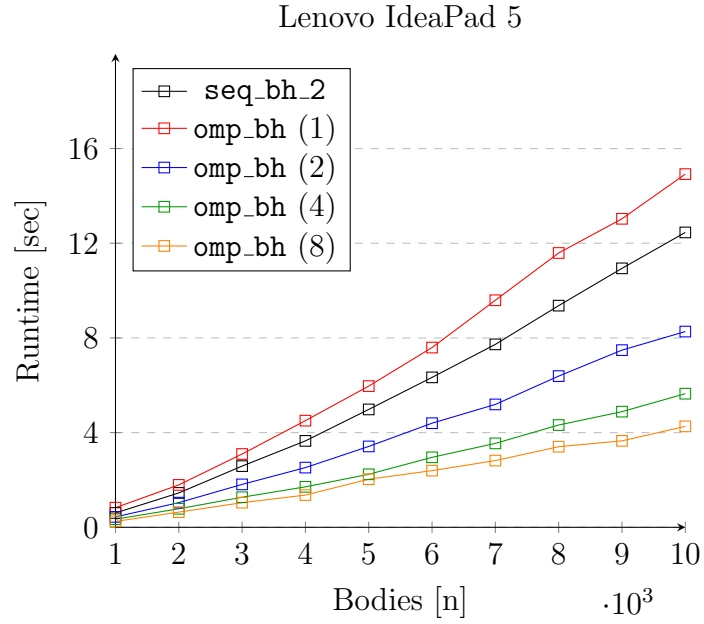


Figure 3.15: Confronto delle performance di `seq_bh_2` e `omp_bh` con  $\theta = 0.5$  su 100 passi di simulazione da 1000 a 10000 corpi. I numeri tra parentesi nella legenda del grafico indicano il numero di thread attivi

## 3.4 Parallelizzazione tramite MPI

### 3.4.1 Analisi, limitazioni riscontrate e implementazione

Per parallelizzare l'algoritmo di Barnes-Hut utilizzando la libreria MPI si è optato per un approccio molto simile a quello della versione estensiva analizzando cosa era sconveniente e limitante fra quello che era possibile parallelizzare:

1. Creazione e aggiornamento dell'albero
2. Calcolo della velocità di ogni corpo
3. Corpi da calcolare

Un modo per parallelizzare la **creazione e l'aggiornamento** dell'albero sarebbe potuto essere quello di distribuire tra i processi **branch** diversi dell'albero per fare in modo che ognuno calcolasse la nuova disposizione dei nodi. Data però la natura sbilanciata dell'albero si avrebbero dei rami ricchi di nodi ed altri molto meno, dunque si avrebbero molti processi in wait che aspettano i processi più lenti per condividere la propria parte di albero ed eseguire il **mounting** dell'albero completo. In oltre ci sarebbe l'overhead aggiunto dalla divisione dei nodi e dei branch e soprattutto dalle comunicazioni aggiuntive necessarie. Dunque si è optato per lasciare che ogni processo modifichi per intero il proprio albero.

Si è scelto di parallelizzare solo sul numero di corpi di cui calcolare la posizione per ridurre al minimo l'overhead delle molteplici comunicazioni che sono limitate, come per la versione estensiva, alle nuove posizioni calcolate. Ogni processo costruisce dunque il proprio albero e lo aggiorna con i valori delle posizioni condivise.

```

/*
|   each process saves only the speeds of its own bodies in a smaller local vector
*/
Vec3D localv[my_slice_size];
for (size_t i = 0; i < my_slice_size; i++)
{
    localv[i] = bodies[i + my_slice_start].vel;
}

if (num_steps <= 500)
    interval = 1;
else
    interval = num_steps / 500;

leaves = malloc(sizeof(BHOctree*) * num_bodies);
bho = generate_bodies_bho(bodies, leaves, num_bodies);

for (n = 0; n < num_steps; ++n)
{
    for (i = my_slice_start; i < my_slice_end; ++i)
    {
        sum_vel = compute_velocity_increment(bho, &bodies[i], G, theta);
        localv[i - my_slice_start].x += sum_vel.x * TIME_DELTA;
        localv[i - my_slice_start].y += sum_vel.y * TIME_DELTA;
        localv[i - my_slice_start].z += sum_vel.z * TIME_DELTA;
    }

    for (i = my_slice_start; i < my_slice_end; ++i)
    {
        bodies[i].pos.x += localv[i - my_slice_start].x * TIME_DELTA;
        bodies[i].pos.y += localv[i - my_slice_start].y * TIME_DELTA;
        bodies[i].pos.z += localv[i - my_slice_start].z * TIME_DELTA;
    }

    MPI_Allgather(bodies+my_slice_start, my_slice_size, MPI_Body,
        bodies, count, disp, MPI_Body, MPI_COMM_WORLD);

    update_bodies_bho(bho, bodies, leaves, num_bodies);
}

```

Figure 3.16: Screenshot del frammento di codice di interesse di `mpi_bh.c`

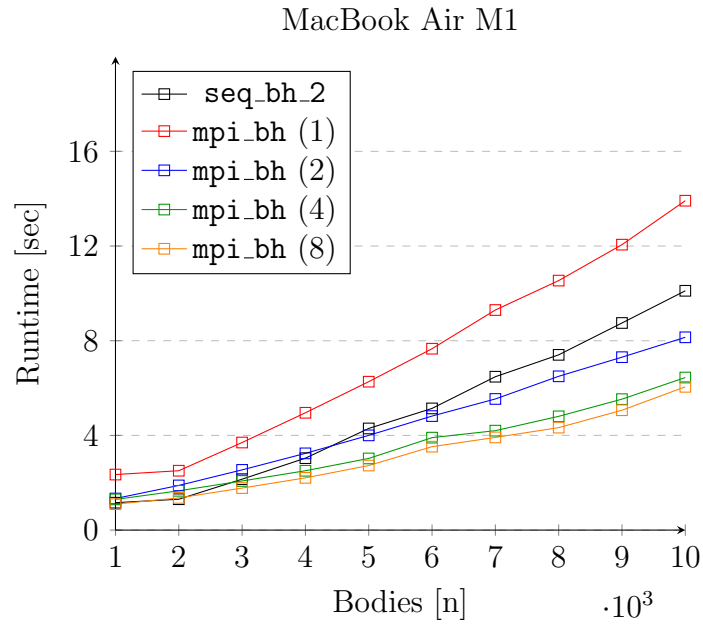
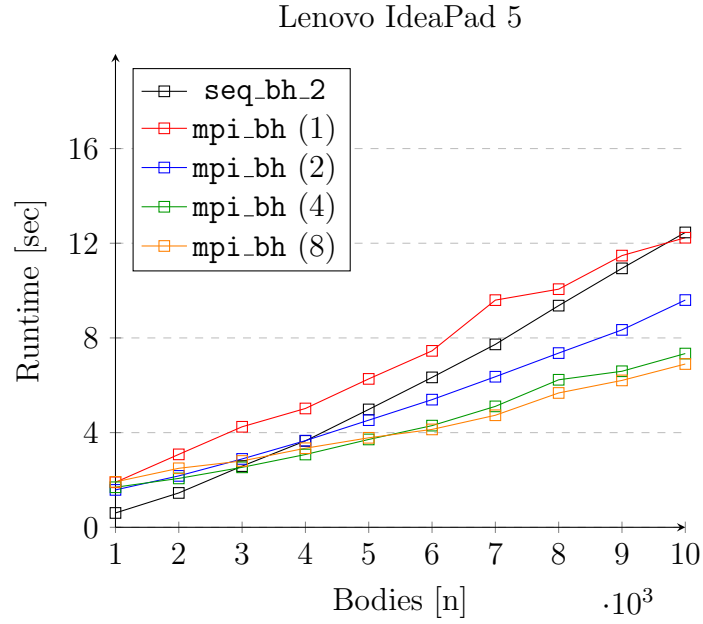


Figure 3.17: Confronto delle performance di `seq_bh_2` e `mpi_bh` con  $\theta = 0.5$  su 100 passi di simulazione da 1000 a 10000 corpi. I numeri tra parentesi nella legenda del grafico indicano il numero di thread attivi

### 3.5 Analisi dello speed-up e dell'efficienza

Tramite i dati inerenti al runtime delle varie versioni parallelizzate e non mostrati nelle sezioni precedenti, consideriamo lo **speed-up** e l'**efficienza** ottenute da ognuna di tali versioni rispetto all'implementazione sequenziale `seq_bh_2` nel caso con 10000 corpi e  $\theta = 0.5$ .

Programma	Runtime	Speed-up	Efficienza
<code>seq_bh_2</code>	12.456793785	1.000000000	1.00
<code>omp_bh (1)</code>	14.921123028	0.834842911	0.83
<code>omp_bh (2)</code>	8.269041061	1.506437529	0.75
<code>omp_bh (4)</code>	5.644170761	2.207019297	0.55
<code>omp_bh (8)</code>	4.263443470	2.921768254	0.36
<code>mpi_bh (1)</code>	12.233047009	1.018290355	1.01
<code>mpi_bh (2)</code>	9.599754333	1.297615892	0.64
<code>mpi_bh (4)</code>	7.339652538	1.697191212	0.42
<code>mpi_bh (8)</code>	6.897924423	1.805875655	0.22

Figure 3.18: Statistiche per dispositivo Lenovo IdeaPad 5

Programma	Runtime	Speed-up	Efficienza
<code>seq_bh_2</code>	10.109156609	1.000000000	1.00
<code>omp_bh (1)</code>	11.421123028	0.885128072	0.88
<code>omp_bh (2)</code>	7.269041061	1.390713923	0.69
<code>omp_bh (4)</code>	5.244170761	1.927694019	0.48
<code>omp_bh (8)</code>	4.263443470	2.371124815	0.30
<code>mpi_bh (1)</code>	13.914565563	0.726516151	0.72
<code>mpi_bh (2)</code>	8.144491196	1.241226292	0.62
<code>mpi_bh (4)</code>	6.444715023	1.568596373	0.39
<code>mpi_bh (8)</code>	6.052484035	1.670249199	0.21

Figure 3.19: Statistiche per dispositivo MacBook Air M1

## Chapter 4

### Conclusioni

Come mostrato dalle analisi precedenti, entrambi i due algoritmi risultano *sfavorevoli* all'uso della **memoria distribuita** per via della necessità di effettuare una comunicazione ad ogni passo della simulazione. L'overhead imposto da tali comunicazioni risulta trascurabile solo nel caso in cui vengano trasmesse enormi quantità di dati in una singola comunicazione.

Viceversa, l'uso della **memoria condivisa** fornisce risultati più *omogenei* tra loro. Tuttavia, anche tale modalità presenta le sue complicazioni, in particolare per quanto riguarda l'alta difficoltà nel **parallelizzare la struttura dati** utilizzata dall'algoritmo Barnes-Hut. Un'eventuale implementazione che sia in grado di parallelizzare correttamente tale struttura dati con l'introduzione di un minimo overhead per eventuali attese durante inserimenti simultanei porterebbe l'algoritmo ad aver performance più elevate.

Per quanto riguarda l'analisi sullo *speed-up* e l'*efficienza*, essi risultano tendenzialmente massimi durante l'utilizzo di **4 processori** contemporaneamente. In particolare, tali statistiche risultano particolarmente elevate nelle varie implementazioni con **metodo esaustivo**.

Il codice sorgente relativo a questo documento può essere trovato al seguente link: <https://github.com/Exyss/n-body-parallelization>