Introduzione

Nel problema N-Body, ci troviamo di fronte alla sfida di determinare le posizioni e le velocità di una serie di particelle interagenti nel corso del tempo. Immaginiamo, ad esempio, un astrofisico che desideri conoscere le posizioni e le velocità di un gruppo di stelle, oppure un chimico interessato alle posizioni e alle velocità di una collezione di molecole o atomi.

Per affrontare questo compito ci si serve di un risolutore, un programma in grado di trovare la soluzione al problema simulando il comportamento delle particelle. L'input richiesto comprende la massa, la posizione e la velocità di ciascuna particella all'inizio della simulazione, mentre l'output tipicamente fornisce le posizioni e le velocità di ogni particella in una sequenza di tempi o al termine di un determinato intervallo di tempo specificati dall'utente.

Ai fini del progetto si è scelto di implementare il risolutore servendosi della soluzione banale quadratica rispetto al numero di particelle. Tuttavia si noti che esistono anche altre soluzioni più efficienti, ad esempio quella che utilizza l'algoritmo di Barnes-Hut.

Il risolutore è stato testato sulla piattaforma Google Cloud su un cluster di 4 istanze e2-standard-8. Ciascuna istanza offre 32GB di RAM e 8vCPU; di queste, solo 4 sono realmente core fisici, per un totale di 16 core effettivi per l'intero cluster.

Le soluzioni proposte

Il risolutore è stato implementato 2 volte. Entrambe le versioni parallelizzano la versione sequenziale del risolutore fornitoci contestualmente all'assegnazione del progetto. La differenza tra le 2 soluzioni risiede nelle modalità di comunicazione e sincronizzazione tra processi.

Soluzione 1

In questa soluzione si è seguito l'approccio più semplice e lineare possibile.

Ad ogni iterazione, l'intero array contente le informazioni su tutti i bodies del problema viene mandato in broadcast a tutti i processi che partecipano alla computazione della soluzione.

Ciascuno di essi è in grado di conoscere quale è la parte di questo array che deve computare. Per ogni body appartenente alla propria parte di array, il processo aggiorna le velocities.

Le informazioni così calcolate vengono raccolte dal processo master, che infine aggiorna la posizione di ciascun body.

Soluzione 2

La Soluzione 2 è più creativa.

Inizialmente il processo master suddivide l'array di body in tante parti quanti sono i processi che partecipano alla computazione. Ad ogni processo viene inviata una parte di array.

Ad ogni iterazione, ciascun processo:

- invia i propri bodies in broadcast a tutti gli altri processi;
- calcola le velocities dei propri bodies considerando solo le forze che intercorrono tra di essi;
- riceve da ogni processo n i bodies del processo n;
- ad ogni ricezione dal processo n, aggiorna le velocities dei propri bodies considerando solo le forze che intercorrono tra essi e i bodies del processo n;
- completate tutte le ricezioni e le relative computazioni delle velocities, aggiorna le posizioni dei propri bodies.

Al termine delle n iterazioni (numero deciso dall'utente), il processo master raccoglie tutti i bodies per la stampa dei risultati.

Dettagli implementativi

L'implementazione delle due soluzioni è stata scritta in C, servendosi della libreria OpenMPI per parallelizzare le operazioni. Essa è un'implementazione open source di MPI sviluppata e mantenuta da un consorzio di partner di ricerca e industriali. MPI è uno standard di message-passing standardizzato e portabile, progettato per funzionare su architetture di calcolo parallelo. Esso definisce la sintassi e la semantica di librerie utilizzate per scrivere programmi di message-passing portabili in C e altri linguaggi di programmazione.

Alcuni snippets / funzioni vengono utilizzati da entrambe le soluzioni. Vediamole:

Body structure

```
typedef struct
{
    float x, y, z, vx, vy, vz;
} Body;
```

È la struttura che rappresenta il singolo Body. Ciascun Body è formato dalle *posizioni* (.x, .y, .z) e dalle *velocities* (.vx, .vy, .vz).

int compute_sendreceivecount_withoffset

```
int compute_sendreceivecount_withoffset(int sendbufsize, int
send_recv_count[], int offsets[], int size)
{

    int elementsPerProcess = sendbufsize / (size);
    int remainder = sendbufsize % (size);

    // Inizializzo array
    send_recv_count[0] = elementsPerProcess;
    offsets[0] = 0;
    for (int i = 1; i < size; i++)
    {
        send_recv_count[i] = elementsPerProcess;
```

```
        if (remainder > 0)
        {
            send_recv_count[i]++;
            remainder--;
        }
        offsets[i] = offsets[i - 1] + send_recv_count[i - 1];
    }
}
```

Questa funzione calcola il numero di oggetti da computare per ciascun processo. Il calcolo viene eseguito distribuendo in maniera equa lo stesso numero di oggetti su ciascun processo. Se ciò non è possibile (cioè se la divisione nOggetti/nProcessi non ha resto 0) i rimanenti oggetti vengono distribuiti in maniera equa sui primi m processi, con m pari al resto della divisione di cui sopra.

Nel codice, *sendbufsize* è il numero totale di oggetti da processare e *size* è il numero di processi che partecipano al calcolo. I due array *send_recv_count* e *offsets* vengono utilizzati per memorizzare l'output della funzione:

- send_recv_count[i] conterrà il numero di oggetti da processare dal processo con *rank* i;
- offsets[i] conterrà l'indice del primo oggetto da processare dal processo con *rank* i nell'array contenente tutti gli oggetti; viene utilizzato come parametro *displs* nelle chiamate a funzioni di comunicazione collettiva messe a disposizione da MPI.

randomizeBodies

```
void randomizeBodies(Body p[], int n)
{
    for (int i = 0; i < n; i++)
    {
        p[i].x = 2.0f * (rand() / (float)RAND_MAX) - 1.0f;
        p[i].y = 2.0f * (rand() / (float)RAND_MAX) - 1.0f;
        p[i].z = 2.0f * (rand() / (float)RAND_MAX) - 1.0f;
        p[i].vx = 2.0f * (rand() / (float)RAND_MAX) - 1.0f;
        p[i].vy = 2.0f * (rand() / (float)RAND_MAX) - 1.0f;
        p[i].vz = 2.0f * (rand() / (float)RAND_MAX) - 1.0f;
    }
}
```

Questa funzione inizializza in maniera pseudo-randomica n bodies all'interno dell'array p. È importante sottolineare che ad esecuzioni diverse del programma corrispondono uguali inizializzazioni dei primi n bodies (ovvero, eseguendo il programma con n bodies e successivamente con n + x bodies, i primi n bodies della seconda esecuzione sono inizializzati con gli stessi valori degli n bodies della prima esecuzione). Ciò è vero perchè la funzione rand() genera numeri a partire da un seed, e fissando il seed la sequenza di numeri generati rimane costante. Nel nostro caso, il seed utilizzato dalla funzione rand() è quello di default, cioè 1.

bodiesPosition

```
void bodiesPosition(Body p[], int nBodies, float dt)
{
    for (int i = 0; i < nBodies; i++)
    { // integrate position
        p[i].x += p[i].vx * dt;
        p[i].y += p[i].vy * dt;
        p[i].z += p[i].vz * dt;
    }
}
```

Questa funzione aggiorna le posizioni di *nBodies* bodies all'interno dell'array p. Viene chiamata al termine di ciascuna iterazione, ma:

- Nella Soluzione 1, viene chiamata dal processo master che aggiorna le posizioni di tutti i bodies del problema;
- Nella Soluzione 2, viene chiamata da ciascun processo che aggiorna le posizioni solo dei propri bodies.

main

Gran parte della funzione main è uguale nelle due soluzioni.

```
// Init MPI
MPI_Init(&argc, &argv);

// Compute size and rank
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Si inizializza OpenMPI e e si salvano il numero di processi che partecipano alla computazione (*size*) e il rank del processo locale (*rank*).

```
// Take parameters from input
int nBodies = argc > 1 ? atoi(argv[1]) : DEFAULT_NBODIES;
int nIters = argc > 2 ? atoi(argv[2]) : DEFAULT_NITERS;
const char *resultsFileName = argc > 3 ? argv[3] : DEFAULT_FILENAME;
```

Si legge dagli argomenti da riga di comando alcuni parametri utili per l'esecuzione:

- *nBodies*: numero di bodies in input al problema
- *nIters*: numero di iterazioni da computare; ciascuna iterazione calcola le velocities e le posizioni dei bodies ad un intervallo di 0.01s rispetto all'iterazione precedente;
- *resultsFileName*: il nome del file all'interno del quale scrivere i risultati finali dell'esecuzione.

```
// Compute sendcount and offsets
int sendrecvcount[size];
int offsets[size];
compute_sendreceivecount_withoffset(nBodies, sendrecvcount, offsets,
size);
```

Si calcola per ciascun processo il numero di bodies da computare e l'indice di partenza nell'array di bodies completo.

```
// Create new MPI_Datatype for Body
MPI_Datatype MPI_Body;
MPI_Type_contiguous(6, MPI_FLOAT, &MPI_Body);
MPI_Type_commit(&MPI_Body);
```

Si crea una nuovo MPI_Datatype che rappresenta la struttura Body. La scelta di creare un MPI_Datatype ha permesso di utilizzare le funzioni di comunicazione collettiva di MPI con maggior semplicità logica e sintattica.

```
// Init bodies pos & vel
if (rank == 0)
{
    randomizeBodies(p, nBodies);
}
```

Si inizializza l'array di Body *p* dal processo master.

computePositions

La grande differenza implementativa tra le due soluzioni risiede nella funzione *computePositions*.

Soluzione 1

```
void computePositions(Body p[], int nBodies, int nIters, MPI_Datatype
datatype, float dt, int offsets[], int sendrecvcount[], int rank)
{
    for (int iter = 0; iter < nIters; iter++)
    {

        // Bcast complete array to slaves
        MPI_Bcast(p, nBodies, datatype, 0, MPI_COMM_WORLD);

        // Debug
        debugPrint("After Bcast", sendrecvcount[rank], p, rank);

        // Compute forces, each slave just his part
        bodiesForce(p, nBodies, offsets[rank], sendrecvcount[rank], dt);
```

```
        // Gather results from slaves to master
        MPI_Gatherv(&p[offsets[rank]], sendrecvcount[rank], datatype, p,
sendrecvcount, offsets, datatype, 0, MPI_COMM_WORLD);

        // Debug
        debugPrint("After bodiesForce", nBodies, p, rank);

        // If master
        if (rank == 0)
        {
            bodiesPosition(p, nBodies, dt);
        }
    }
}
```

In questa versione di *computePositions* ad ogni iterazione viene eseguito il Broadcast dal master a tutti gli altri processi di tutto l'array completo di bodies. Ogni processo chiama poi la *bodiesForce*, che aggiorna le velocities *solo dei suoi bodies* ma avendo a disposizione *tutti* i bodies. I bodies così aggiornati vengono ripresi attraverso la Gather dal processo master. Esso poi aggiorna le *posizioni* di tutti i bodies.

Soluzione 2

```
void computePositions(Body p[], Body myBodies[], int nLocalBodies, int
nIters, MPI_Datatype datatype, float dt, int offsets[], int
sendrecvcount[], int rank, int size)
{
    for (int iter = 0; iter < nIters; iter++)
    {

        MPI_Request requests[size];

        // Non-blocking bcast to others slaves +
        // Receive from other slaves
        for (int i = 0; i < size; i++)
        {
            void *buffer;
            int count;

            // I love one-liners
            (i == rank) ? (buffer = myBodies, count = nLocalBodies) :
(buffer = &p[offsets[i]], count = sendrecvcount[i]);

            // 1 will be to send myBodies, size - 1 will be to receive
other processes' myBodies
            MPI_Ibcast(buffer, count, datatype, i, MPI_COMM_WORLD,
&requests[i]);
        }

        // Compute forces, each slave just his part
        bodiesForce(myBodies, nLocalBodies, myBodies, nLocalBodies, dt);
```

```
        debugPrint("After local bodiesForce", nLocalBodies, myBodies,
rank);

        // Compute forces with local bodies from other processes as they
arrive
        for (int i = 0; i < size; i++)
        {

            // It will be equal to the rank of the process for which the
req is completed
            int index;
            MPI_Status status;
            MPI_Waitany(size, requests, &index, &status);

            if (index != rank)
            {
                // Compute forces on myBodies with forces by other
processes' myBodies
                bodiesForce(&p[offsets[index]], sendrecvcount[index],
myBodies, nLocalBodies, dt);
            }
        }

        // Compute myBodies position
        bodiesPosition(myBodies, nLocalBodies, dt);
    }
}
```

In questa versione di *computePositions* ciascun processo effettua la Broadcast in modalità *non bloccante* verso gli altri processi, inviando il proprio array di Bodies. Questo perchè ciascun processo ha a disposizione solo la propria parte di bodies e non è a conoscenza dell'array completo di bodies (riempito e valido solo nel processo master). Inoltre, ciascun processo aspetta di ricevere, sempre in modalità non bloccante, la corrsipondente Broadcast dagli altri processi. Nel frattempo, calcola le velocities dei suoi bodies tenendo in considerazione solo le forze che intercorrono tra di essi. In seguito, ad ogni ricezione dei bodies dagli altri processi, aggiorna le velocities dei propri bodies considerando le forze che intercorrono tra essi e i bodies appena ricevuti. Infine, completate tutte le ricezioni e le relative computazioni delle velocities, aggiorna le posizioni dei propri bodies.

Istruzioni per l'esecuzione

Per poter eseguire uno dei due risultori procedere in questo modo:

Compilazione

```
mpicc ./parallelX.c ./mycollective/mycollective.c -o parallelX.out -lm
```

N.B. Sostuire *X* con la versione della soluzione (*1* o *2*).

N.B. Il flag *-lm* serve a linkare la libreria math.h.

Esecuzione

```
mpirun --allow-run-as-root -np {np} parallelX.out {nBodies} {nIters}
{resultsFileName}
```

Parametri per l'esecuzione

Parametro	Descrizione	Default
np	numero di processi che partecipano alla computazione	NA
nBodies	numero di bodies che faranno parte del problema	10000
nIters	numero di iterazioni per le quali calcolare velocities e posizioni dei bodies	5
resultsFileName	il path del file che verrà creato per salvare il risultato dell'esecuzione	"../results.txt"

Correttezza

Per verificare la correttezza delle implementazioni proposte sono state eseguite le due soluzioni in modalità sequenziale. In seguito sono stati confrontati i risultati ottenuti con quelli delle esecuzioni in modalità concorrente, a 2 e 4 processi. Tali esperimenti sono stati eseguiti sull'immagine docker messa a disposizione contestualmente al corso di PCPC.

È possibile affermare che le soluzioni proposte sono *corrette* in quanto, fissando il numero di bodies e le proprietà iniziali di ciascuno, esse restituiscono risultati analoghi a prescindere dal numero di processi utilizzati per risolvere l'istanza del problema.

Di seguito alcuni snippets dei file di output delle esecuzioni su 2000 bodies, 10 iterazioni e 1 - 2 - 4 processi. I file completi sono disponibili in repo, nella cartella results/correctness.

Soluzione 1

1 processo

```
Total time: 4.556783s

p[0].x: -4.227  p[0].y: 7.640   p[0].z: -0.369
p[0].vx: -58.966    p[0].vy: 107.754    p[0].vz: -6.206

p[1].x: 4.723   p[1].y: -1.230  p[1].z: 2.788
p[1].vx: 65.296 p[1].vy: -16.801    p[1].vz: 39.495

p[2].x: 4.839   p[2].y: -0.219  p[2].z: -0.277
p[2].vx: 66.897 p[2].vy: -3.576 p[2].vz: -5.116
```

```
p[3].x: 2.107   p[3].y: -2.154  p[3].z: 2.179
p[3].vx: 31.029 p[3].vy: -38.062    p[3].vz: 29.727

p[4].x: 4.880   p[4].y: 2.188   p[4].z: 9.385
p[4].vx: 70.928 p[4].vy: 32.306 p[4].vz: 135.201
```

2 processi

```
Total time: 2.353013s

p[0].x: -4.227  p[0].y: 7.640   p[0].z: -0.369
p[0].vx: -58.966    p[0].vy: 107.754    p[0].vz: -6.206

p[1].x: 4.723   p[1].y: -1.230  p[1].z: 2.788
p[1].vx: 65.296 p[1].vy: -16.801    p[1].vz: 39.495

p[2].x: 4.839   p[2].y: -0.219  p[2].z: -0.277
p[2].vx: 66.897 p[2].vy: -3.576 p[2].vz: -5.116

p[3].x: 2.107   p[3].y: -2.154  p[3].z: 2.179
p[3].vx: 31.029 p[3].vy: -38.062    p[3].vz: 29.727

p[4].x: 4.880   p[4].y: 2.188   p[4].z: 9.385
p[4].vx: 70.928 p[4].vy: 32.306 p[4].vz: 135.201
```

4 processi

```
Total time: 1.378734s

p[0].x: -4.227  p[0].y: 7.640   p[0].z: -0.369
p[0].vx: -58.966    p[0].vy: 107.754    p[0].vz: -6.206

p[1].x: 4.723   p[1].y: -1.230  p[1].z: 2.788
p[1].vx: 65.296 p[1].vy: -16.801    p[1].vz: 39.495

p[2].x: 4.839   p[2].y: -0.219  p[2].z: -0.277
p[2].vx: 66.897 p[2].vy: -3.576 p[2].vz: -5.116

p[3].x: 2.107   p[3].y: -2.154  p[3].z: 2.179
p[3].vx: 31.029 p[3].vy: -38.062    p[3].vz: 29.727

p[4].x: 4.880   p[4].y: 2.188   p[4].z: 9.385
p[4].vx: 70.928 p[4].vy: 32.306 p[4].vz: 135.201
```

Soluzione 2

1 processo

```
Total time: 4.561903s

p[0].x: -4.227  p[0].y: 7.640   p[0].z: -0.369
p[0].vx: -58.966    p[0].vy: 107.754    p[0].vz: -6.206

p[1].x: 4.723   p[1].y: -1.230  p[1].z: 2.788
p[1].vx: 65.296 p[1].vy: -16.801    p[1].vz: 39.495

p[2].x: 4.839   p[2].y: -0.219  p[2].z: -0.277
p[2].vx: 66.897 p[2].vy: -3.576 p[2].vz: -5.116

p[3].x: 2.107   p[3].y: -2.154  p[3].z: 2.179
p[3].vx: 31.029 p[3].vy: -38.062    p[3].vz: 29.727

p[4].x: 4.880   p[4].y: 2.188   p[4].z: 9.385
p[4].vx: 70.928 p[4].vy: 32.306 p[4].vz: 135.201
```

2 processi

```
Total time: 2.401930s

p[0].x: -4.227  p[0].y: 7.640   p[0].z: -0.369
p[0].vx: -58.966    p[0].vy: 107.754    p[0].vz: -6.206

p[1].x: 4.723   p[1].y: -1.230  p[1].z: 2.788
p[1].vx: 65.296 p[1].vy: -16.801    p[1].vz: 39.496

p[2].x: 4.839   p[2].y: -0.218  p[2].z: -0.276
p[2].vx: 66.898 p[2].vy: -3.576 p[2].vz: -5.110

p[3].x: 2.107   p[3].y: -2.153  p[3].z: 2.179
p[3].vx: 31.030 p[3].vy: -38.039    p[3].vz: 29.723

p[4].x: 4.880   p[4].y: 2.188   p[4].z: 9.385
p[4].vx: 70.928 p[4].vy: 32.306 p[4].vz: 135.200
```

4 processi

```
Total time: 1.332988s

p[0].x: -4.227  p[0].y: 7.640   p[0].z: -0.369
p[0].vx: -58.967    p[0].vy: 107.753    p[0].vz: -6.207

p[1].x: 4.723   p[1].y: -1.230  p[1].z: 2.788
p[1].vx: 65.296 p[1].vy: -16.801    p[1].vz: 39.495
```

```
p[2].x: 4.839   p[2].y: -0.219  p[2].z: -0.277
p[2].vx: 66.896 p[2].vy: -3.576 p[2].vz: -5.118

p[3].x: 2.107   p[3].y: -2.153  p[3].z: 2.179
p[3].vx: 31.033 p[3].vy: -38.039    p[3].vz: 29.724

p[4].x: 4.880   p[4].y: 2.188   p[4].z: 9.385
p[4].vx: 70.928 p[4].vy: 32.306 p[4].vz: 135.200
```

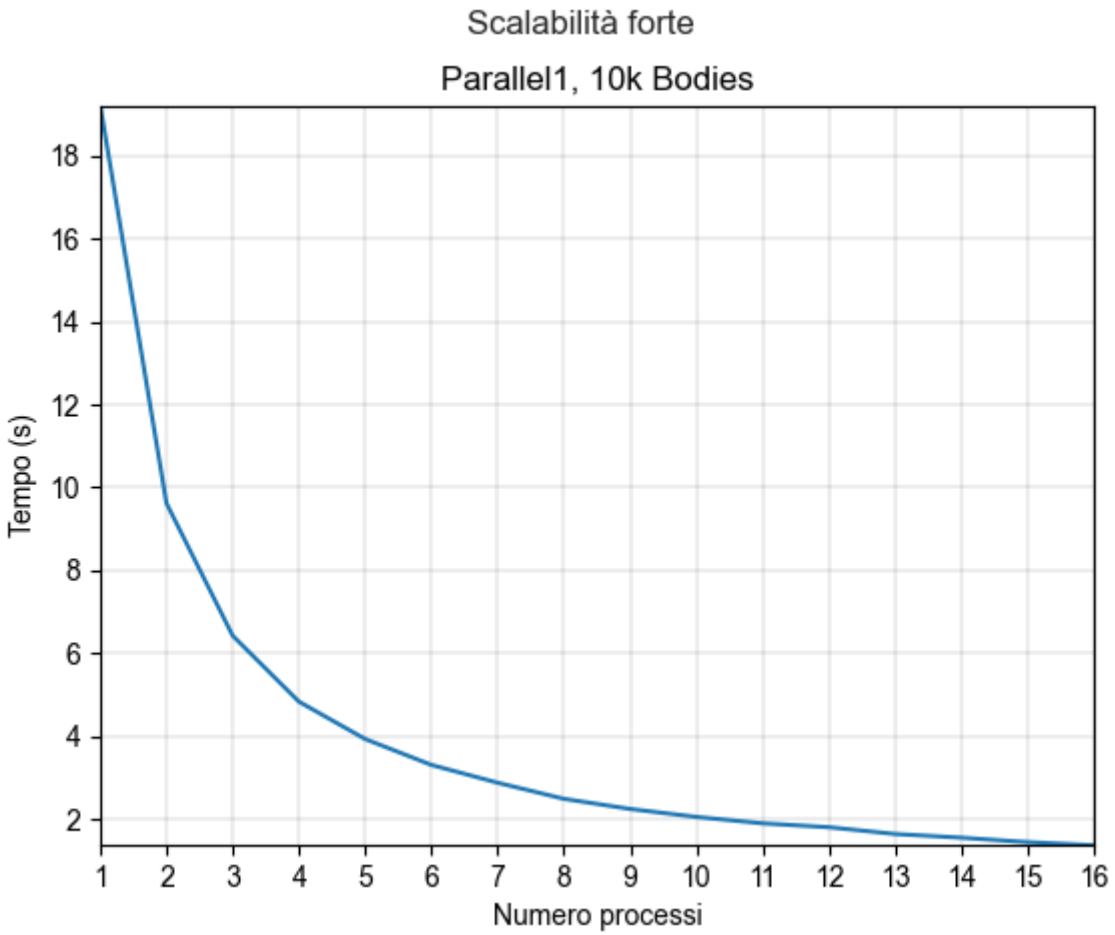
Benchmarks

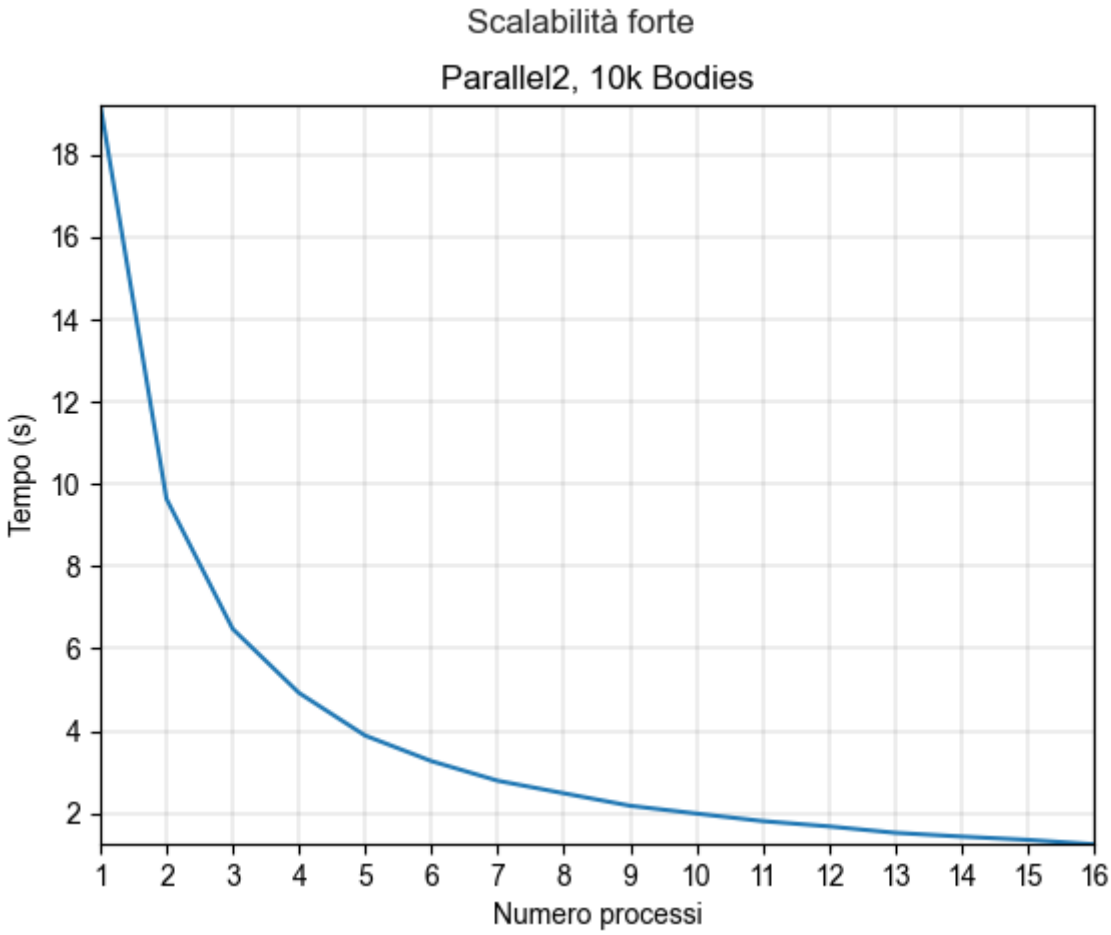
Per effettuare i benchmarks è stato utilizzato un cluster di 4 istanze Compute Engine di Google Cloud di tipo e2-standard-8. Ciascuna istanza offre 32GB di RAM e 8vCPU; di queste, solo 4 sono realmente core fisici, per un totale di 16 core effettivi per l'intero cluster.

Sono state testate sia la scalabilità forte che la scalabilità debole *per entrambe le soluzioni*. Di seguito sono riportati alcuni grafici che raffigurano i risultati ottenuti. Per maggior precisione e più dettagli di ciascuna istanza del problema, si rimanda alla cartella contente i file di output.

Scalabilità forte

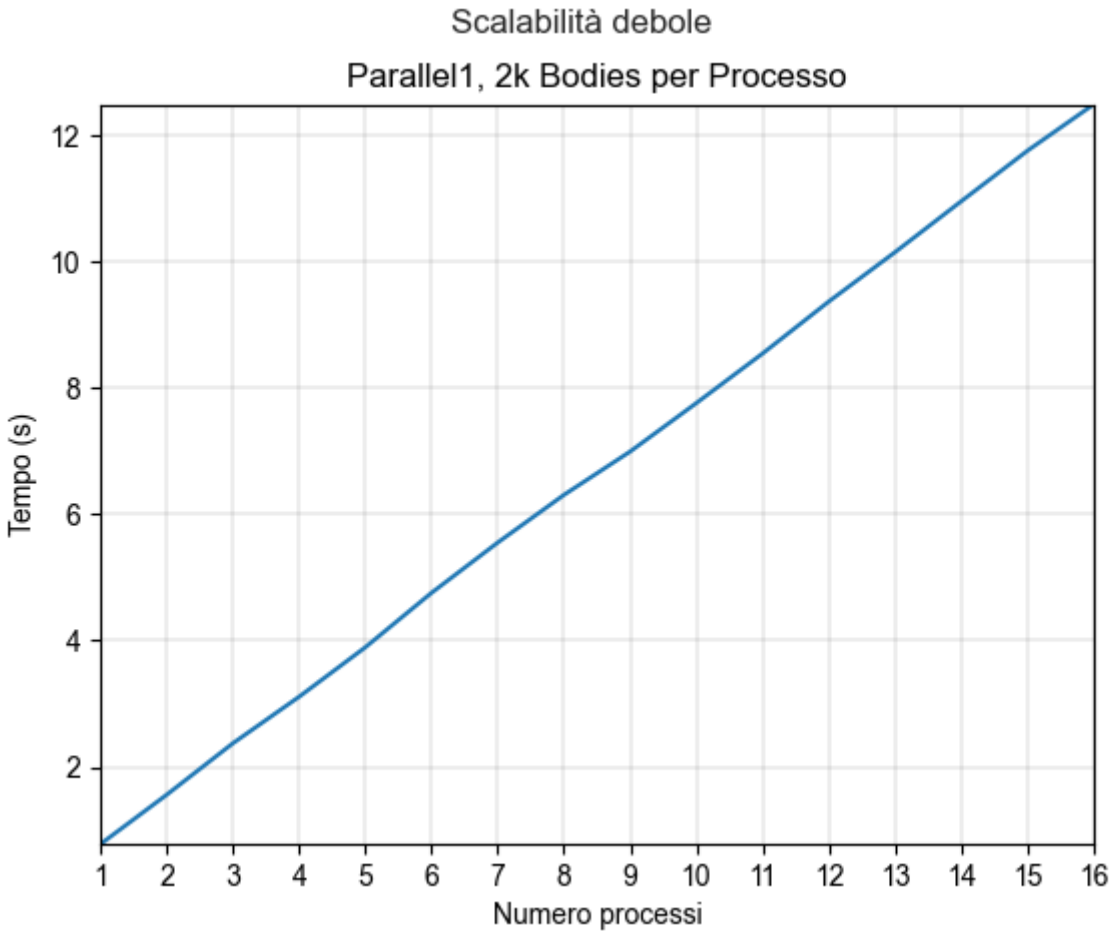
Per la scalabilità forte è stata eseguita un'istanza del problema con 10k bodies e 10 iterazioni, variando di volta in volta il numero di processi partecipanti al computo.

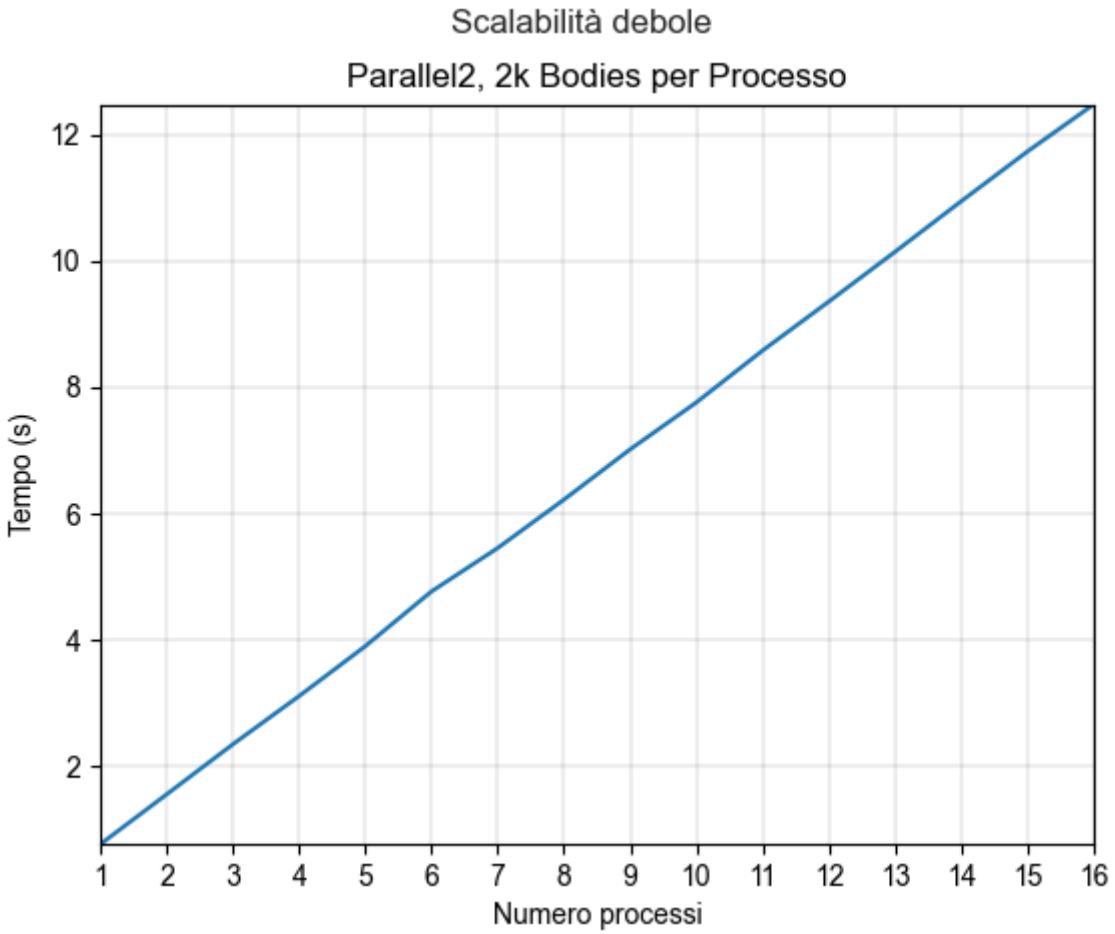




Scalabilità debole

Per la scalabilità debole si è scelto di eseguire un'istanza del problema con 2k bodies per ciascun processo e 10 iterazioni. In totale quindi il numero di bodies è variato da 2k a 32k.


Scalabilità debole
Parallel1, 2k Bodies per Processo
Tempo (s)
Numero processi
2
4
6
8
10
12
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16


Scalabilità debole
Parallel2, 2k Bodies per Processo
Tempo (s)
Numero processi
2
4
6
8
10
12
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

Conclusioni

In linea generale è possibile affermare che la distribuzione del carico di lavoro su più processi e la loro parallelizzazione ha permesso di raggiungere risultati soddisfacenti. I benchmarks di scalabilità forte hanno evidenziato che il tempo di esecuzione dei due risolutori diminuisce sempre all'aumentare del numero di processi che partecipano alla computazione. È probabile che esso possa diminuire ancora aumentando il numero di processi oltre l'attuale soglia di 16. Allo stato attuale e considerando solo i run i cui output sono in repo, lo speedup massimo ottenuto è pari a 15.5, con la Soluzione 2.

I risultati ottenuti dai benchmarks di scalabilità debole sono invece meno soddisfacenti. Il tempo di esecuzione all'aumentare dei processi (e quindi del numero di bodies totali) è sempre aumentato, mentre l'obiettivo (corrispondente all'ottimo desiderato, quasi impossibile da ottenere nel contesto in cui si è lavorato) era che rimanesse costante.

Ciò che sorprende maggiormente è che entrambe le versioni del risolutore hanno ottenuto performance praticamente identiche. I tempi di esecuzione sono molto vicini (variano di decimi di secondo) e gli speedup ottenuti sono analoghi (a parità di numero di bodies e di processi), come è possibile verificare dai file di output e dai grafici poco sopra. Ciò accade nonostante le differenze nelle modalità di comunicazione e sincronizzazione tra processi siano non banali. In particolare l'utilizzo di comunicazione in modalità non bloccante e la taglia minore di ciascun messaggio scambiato (ogni processo invia / riceve solo una parte del totale dei bodies) suggerirebbero che la Soluzione 2 sia più veloce. Invece, l'overhead introdotto dall'elevato numero di messaggi scambiati nella Soluzione 2 (ogni processo invia e riceve a / da *tutti* gli altri processi) compensa la taglia più grande dei messaggi scambiati dalla Soluzione 1. In tale implementazione il numero di messaggi è minore, poichè ad ogni iterazione ogni processo invia e riceve solo al / dal master, ma ciascun messaggio è più grande in quanto viene scambiato l'array completo di bodies.