Introduzione

Nel problema N-Body l'obiettivo è calcolare la posizioni e la velocità di un insieme di particelle che interagiscono tra di loro. Ad esempio, un astrofisico potrebbe avere la necessità di calcolare la posizione e la velocità di un gruppo di stelle oppure un chimico potrebbe voler calcolare la posizione e la velocità di un insieme di molecole o atomi.

In generale, un N-Body solver è un programma che trova le soluzioni ad un problema N-Body simulando il comportamento delle particelle. L'input al problema comprende la massa, la posizione e la velocità di ogni particella all'inizio della simulazione, l'output è tipicamente composto dalla posizione e dalla velocità di ogni particella in un determinato istante di tempo.

Le soluzioni proposte

Ogni soluzione è stata implementata a partire da un risolutore sequenziale. (presente a questo link https://github.com/harrism/mini-nbody/blob/master/nbody.c)
L'obiettivo è quello di migliorare le prestazioni della versione sequenziale parallelizzando il codice tramite l'utilizzo di MPI.

Per la risoluzione del problema sono stati implementati 2 N-Body solver, con caratteristiche differenti, che verranno di seguito descritti nel dettaglio.

Per calcolare la velocità di ogni particella è necessario calcolare le distanze e le forze che intercorrono tra quella particella a tutte le altre, dopodiché vengono calcolate le nuove posizioni di ogni particella sulla base delle forze che vengono aggiornate ad ogni iterazione.

Alla base di tutte le soluzioni proposte c'è l'idea che ogni processo si occuperà di una frazione dei body presi in input, in particolare ogni processo calcolerà le posizioni e le velocità di n/p body, con n che è il numero di body in input e p il numero di processi.

Soluzione 1

La prima soluzione è la soluzione più semplice in cui, avendo n body e p processi, ogni processo si occuperà di calcolare le posizioni e le velocità di n/p body in input.
Per fare ciò ad ogni iterazione il processo Master invia l'array di body a tutti i processi attraverso una broadcast e ogni processo calcola le forze da applicare alla propria porzione di particelle usando la funzione bodyForce:

```
void bodyForce(Body *p, float dt, int offset, int n, int allBodies) {
  #pragma omp parallel for schedule(dynamic)
  for (int i = 0; i < n; i++) {
    int myIndex = i + offset;
    float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f;
    for (int j = 0; j < allBodies; j++) {
      float dx = p[j].x - p[myIndex].x;
      float dy = p[j].y - p[myIndex].y;
      float dz = p[j].z - p[myIndex].z;
      float distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;
      float invDist = 1.0f / sqrtf(distSqr);
      float invDist3 = invDist * invDist * invDist;
```

```
      Fx += dx * invDist3; Fy += dy * invDist3; Fz += dz * invDist3;
    }
    p[myIndex].vx += dt*Fx; p[myIndex].vy += dt*Fy; p[myIndex].vz += dt*Fz;
  }
}
```

Una volta calcolate le forze, ogni nodo aggiorna le posizioni di ogni particella e invia i risultati al processo Master attraverso una gatherv.

Al termine delle iterazioni il processo Master avrà i risultati di tutta la computazione.

Soluzione 2

La seconda soluzione si basa sull'idea di sovrapporre la computazione delle posizioni e delle velocità con la comunicazione tra i processi.
In questo caso quindi vengono effettuate p broadcast non bloccanti, nelle quali ogni processo invia la propria porzione di input a tutti gli altri processi. In questo modo, quando una delle broadcast termina, si possono calcolare le distanze con gli altri body e quindi aggiornare le velocità.

Nella prima parte dell'algoritmo avviene l'inizializzazione delle variabili e dell'array di input tramite la funzione randomizeBodies:

```
void randomizeBodies(float *data, int n) {
  for (int i = 0; i < n; i++) {
    data[i] = 2.0f * (rand() / (float)RAND_MAX) - 1.0f;
  }
}
```

Per semplicità ogni processo inizializza l'array utilizzando la funzione, ma sarebbe stata la stessa cosa se l'input fosse stato letto da un file.
La prima iterazione dell'algoritmo viene eseguita al di fuori del for, in quanto tutti i processi hanno tutti i body.

Di seguito è riportato il core della soluzione:

```
for (int iter = 2; iter <= nIters; iter++) {
        // salvo la mia porzione di array in un buffer da inviare agli altri processi
        memcpy(myBodies, p+offset, sizeof(Body)*dim);

        // invio e ricevo dagli altri processi
        sendAndReceive(world_size, myrank, myBodies, displacements, receive_counts,
bodyDataType, requests, p);

        // calcolo le forze da applicare
        myBodyForces(p, dt, displacements[myrank], receive_counts[myrank]);

        // quando ricevo i body da un processo calcolo le forze da applicare tra i miei body
e quelli appena ricevuti
        for(int i = 0; i<world_size; i++){
            MPI_Waitany(world_size, requests, &index, &status);
            if(index != myrank)
                otherBodyForces(p, dt, displacements[myrank], receive_counts[myrank],
displacements[index], receive_counts[index], p);
```

```
        }

        for (int i = 0 ; i < dim; i++) {
            p[i + offset].x += p[i + offset].vx*dt;
            p[i + offset].y += p[i + offset].vy*dt;
            p[i + offset].z += p[i + offset].vz*dt;
        }
    }
```

Ad ogni iterazione la partizione di body del processo i-esimo viene salvata all'interno di un buffer *"myBodies"*, attraverso la funzione *memcpy*, per poter poi essere inviata attraverso funzione sendAndReceive.
Quest'operazione è fondamentale in quanto, quando si utilizzano funzioni di comunicazione non bloccante come MPI_Ibcast, i buffer di invio e quelli di ricezione non devono essere manipolati fino al termine dell'invio.

La sandAndReceive effettua le operazioni di broadcast non bloccanti tra tutti i processi, inviando a tutti gli altri processi la propria partizione dell'input, e ricevendo da tutti questi le restanti parti dell'input.

```
void sendAndReceive(int world_size, int myrank, Body *myBodies, int displacements[], int
receive_counts[], MPI_Datatype bodyDataType, MPI_Request requests[], Body *rcv){
    for(int i = 0; i<world_size; i++){
        if(i == myrank)
            MPI_Ibcast(myBodies, receive_counts[myrank], bodyDataType, i, MPI_COMM_WORLD,
&requests[myrank]);
        else
            MPI_Ibcast(rcv + displacements[i], receive_counts[i], bodyDataType, i,
MPI_COMM_WORLD, &requests[i]);
    }
}
```

Subito dopo si può iniziare ad effettuare il calcolo sulla propria porzione di body utilizzando la funzione myBodyForces:

```
void myBodyForces(Body *p, float dt, int offset, int n){
    for (int i = 0; i < n; i++) {
        int myIndex = i + offset;
        float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f;

        for (int j = offset; j < n + offset; j++) {
            float dx = p[j].x - p[myIndex].x;
            float dy = p[j].y - p[myIndex].y;
            float dz = p[j].z - p[myIndex].z;
            float distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;
            float invDist = 1.0f / sqrtf(distSqr);
            float invDist3 = invDist * invDist * invDist;

            Fx += dx * invDist3; Fy += dy * invDist3; Fz += dz * invDist3;
        }
        p[myIndex].vx += dt*Fx; p[myIndex].vy += dt*Fy; p[myIndex].vz += dt*Fz;
    }
}
```

Una volta fatto ciò si aspetta che una qualsiasi delle broadcast finisca, e si calcolano le forze e le velocità delle particelle relative alla porzione di input appena ricevuta utilizzando la funzione otherBodyForces:

```
void otherBodyForces(Body *p, float dt, int offset, int dim, int otherOffset, int otherDim,
Body *others){
    for (int i = 0; i < dim; i++) {
        int myIndex = i + offset;
        float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f;

        for (int j = otherOffset; j < otherDim + otherOffset; j++) {
            float dx = others[j].x - p[myIndex].x;
            float dy = others[j].y - p[myIndex].y;
            float dz = others[j].z - p[myIndex].z;
            float distSqr = dx*dx + dy*dy + dz*dz + SOFTENING;
            float invDist = 1.0f / sqrtf(distSqr);
            float invDist3 = invDist * invDist * invDist;

            Fx += dx * invDist3; Fy += dy * invDist3; Fz += dz * invDist3;
        }
        p[myIndex].vx += dt*Fx; p[myIndex].vy += dt*Fy; p[myIndex].vz += dt*Fz;
    }
}
```

Una volta terminate tutte le broadcast si procede con l'aggiornamento delle posizioni sulla base delle velocità appena calcolate.
Al termine delle iterazioni si inviano i risultati al processo Master utilizzando una gatherv.

Istruzioni per l'esecuzione

Per poter eseguire la soluzione 1 sarà necessario lanciare i seguenti comandi:

```
make nbody1

mpirun --mca btl_vader_single_copy_mechanism none --allow-run-as-root -np {numero processi}
nbody1 {numero di body} {numero di iterazioni} {file di output}
```

Per poter eseguire la soluzione 2 sarà necessario lanciare i seguenti comandi:

```
make nbody2

mpirun --mca btl_vader_single_copy_mechanism none --allow-run-as-root -np {numero processi}
nbody2 {numero di body} {numero di iterazioni} {file di output}
```

In entrambi i casi:

- numero processi: il numero di processi con cui si vuole lanciare la soluzione
- numero di body: la taglia dell'array in input al problema
- numero di iterazioni: numero di iterazioni da eseguire
- file di output: nome del file in cui salvare i risultati dell'esecuzione

Correttezza

Per la valutazione della correttezza è stato confrontato l'output restituito dalle soluzioni con l'output restituito dalla versione sequenziale dell'algoritmo.
Entrambe le soluzioni risultano essere corrette, al netto di alcune approssimazioni nella soluzione 2 dovute alla separazione delle operazioni matematiche nel calcolo delle velocità e delle forze.
I file dei risultati delle esecuzioni per la correttezza sono salvati all'interno della directory */correctness*.

Benchmark

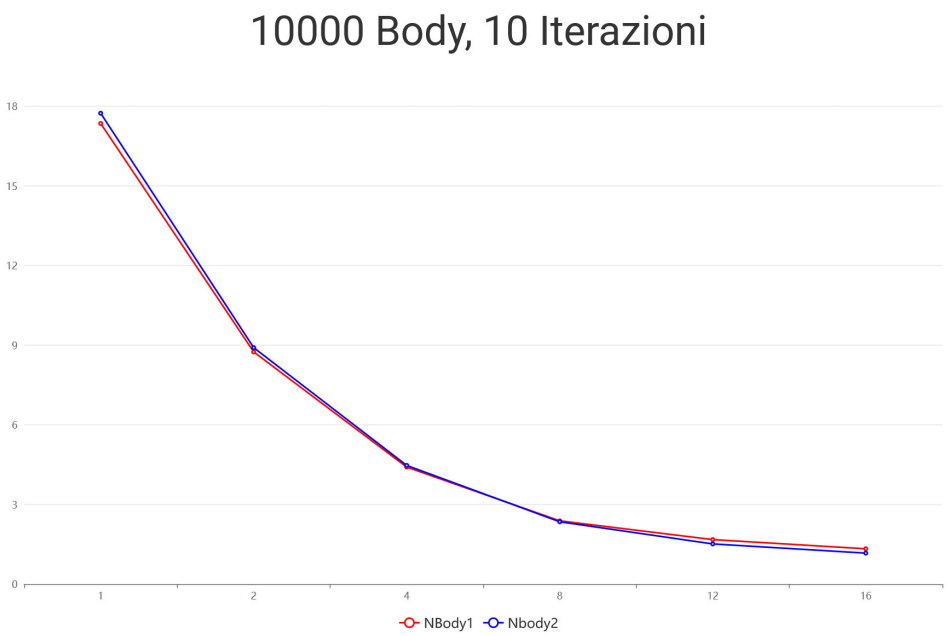
Per la valutazione delle prestazioni delle soluzioni sviluppate si è utilizzato un cluster di 4 macchine e2-standard-8 con 8vCPU 32gb RAM fornite da Google Cloud Platform. Nonostante ciò, ogni macchina ha soltanto 4 core fisici per un totale di 16 core nel cluster.

Scalabilità Forte

È stata valutata la scalabilità forte effettuando dei test con taglia dell'input pari a 10000, 30000 e 50000 body eseguiti su 1, 2, 4, 8, 12 ,16 processi.

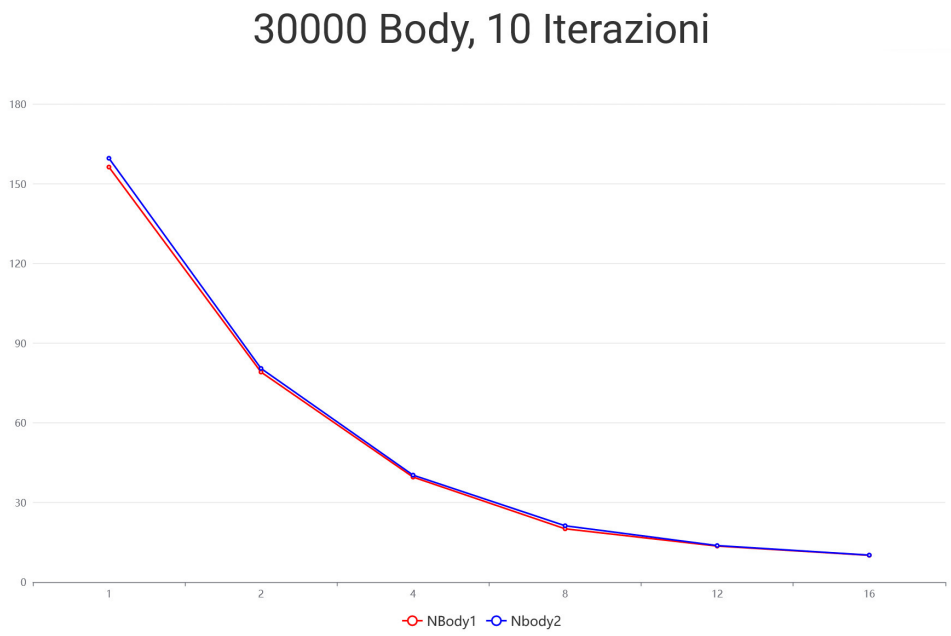
Esecuzione su 10000 body

	1	2	4	8	12	16
Nbody1	17.352	8.746	4.408	2.384	1.677	1.331
Nbody2	17.737	8.907	4.471	2.348	1.514	1.168



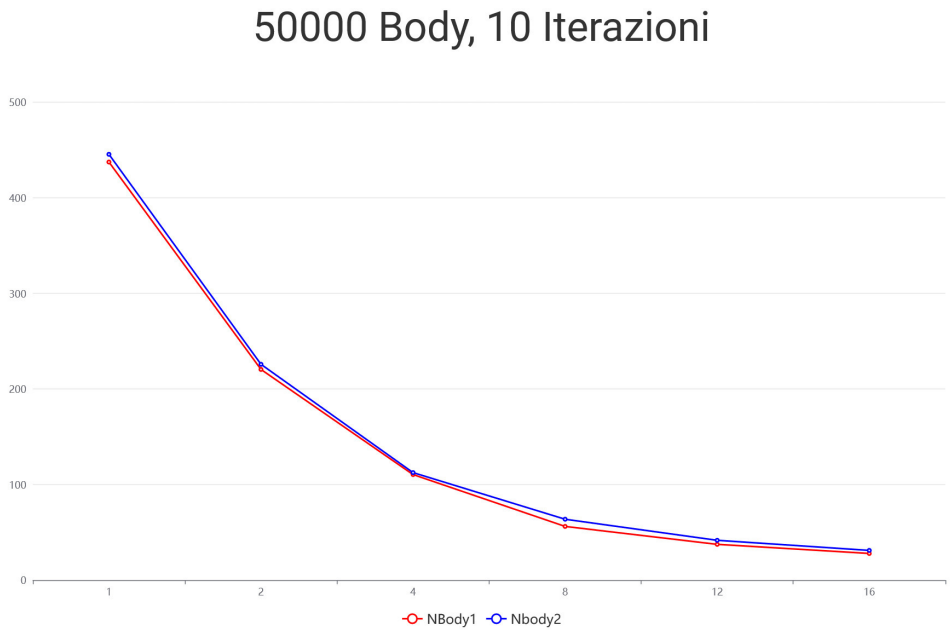
Esecuzione su 30000 body

	1	2	4	8	12	16
Nbody1	156.340	79.127	39.563	20.085	13.577	10.103
Nbody2	159.652	80.492	40.304	21.230	13.770	10.176



Esecuzione su 50000 body

	1	2	4	8	12	16
Nbody1	437.437	220.423	110.445	56.190	37.437	27.911
Nbody2	445.525	225.802	112.507	63.709	41.667	31.107

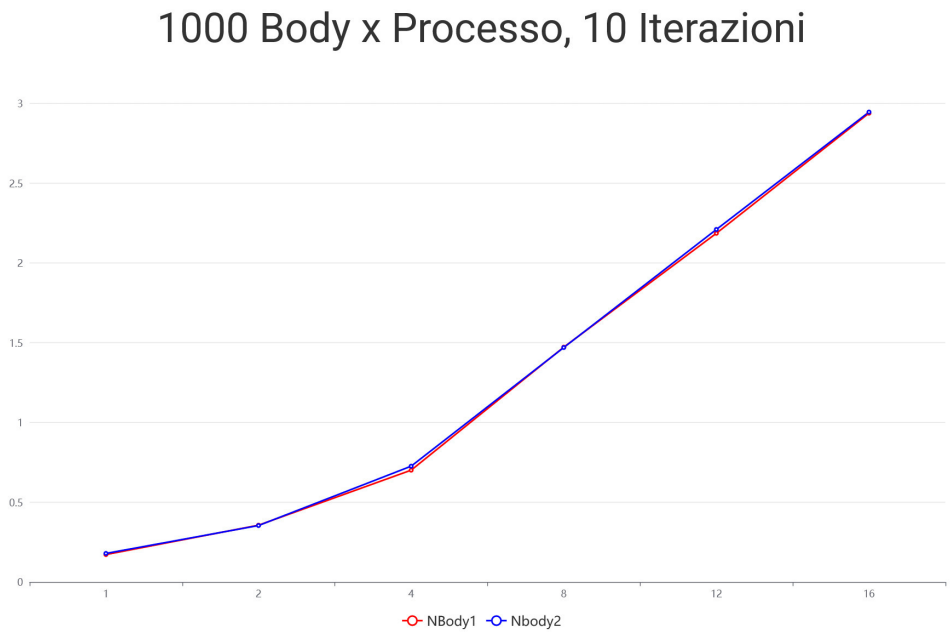


Scalabilità Debole

Per quanto riguarda la scalabilità debole sono stati effettuati test con taglia dell'input pari a 1000, 2000 e 4000 body per processo, su 1, 2, 4, 8, 12, 16 processi.

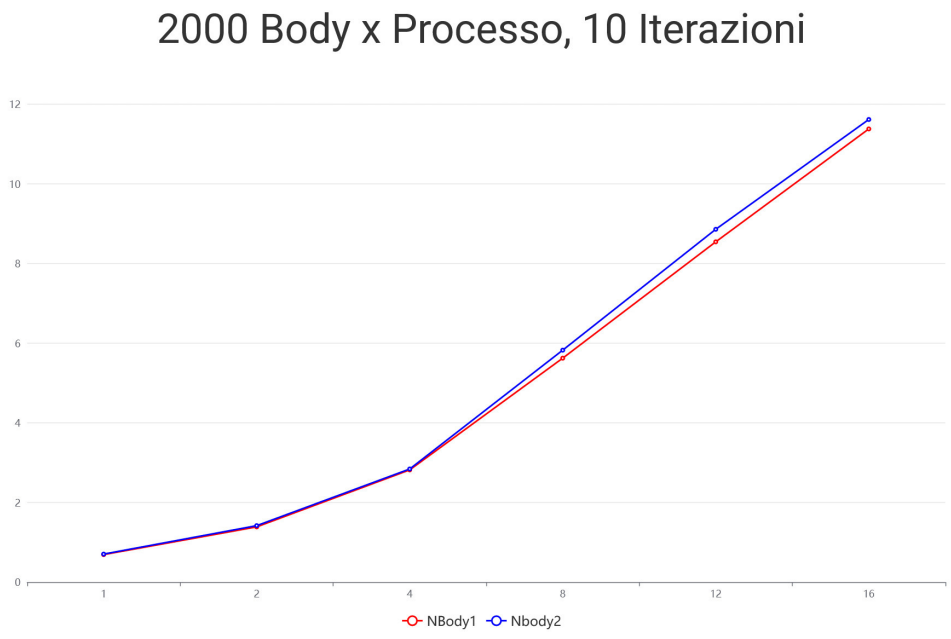
Esecuzione su 1000 body per processo

	1	2	4	8	12	16
Nbody1	0.172	0.356	0.701	1.472	2.186	2.939
Nbody2	0.179	0.354	0.726	1.470	2.210	2.946



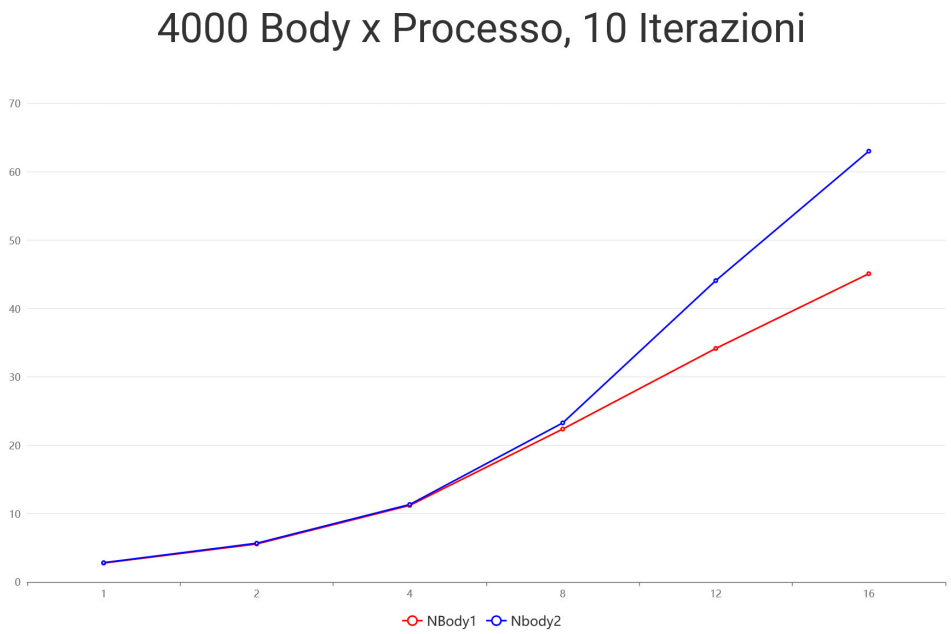
Esecuzione su 2000 body per processo

	1	2	4	8	12	16
Nbody1	0.691	1.387	2.818	5.624	8.545	11.380
Nbody2	0.704	1.418	2.841	5.825	8.860	11.617



Esecuzione su 4000 body per processo

	1	2	4	8	12	16
Nbody1	2.766	5.562	11.199	22.370	34.163	45.094
Nbody2	2.823	5.669	11.334	23.274	44.087	63.017



Conclusioni

Possiamo affermare che entrambe le soluzioni proposte risultano essere soddisfacenti, infatti, all'aumentare del numero di processi lo speedup continua a crescere in entrambe le soluzioni, raggiungendo anche il picco di 15.6 a 16 processi.
Tuttavia, mi aspettavo che la soluzione 2 fosse più performante, soprattutto su grandi taglie dell'input. I risultati di scalabilità forte, invece, mostrano che all'aumentare della taglia la soluzione 1 performa meglio rispetto alla soluzione 2.
Ad esempio con taglia dell'input pari a 50000 body la soluzione 1 è nettamente più veloce della soluzione 2.

Anche dal punto di vista della scalabilità debole i risultati mostrano una tendenza della soluzione 1 ad essere più performante all'aumentare della taglia dell'input.

Alla luce di questi risultati è probabile che l'overhead di comunicazione aggiunto nella soluzione 2 vada ad incrementare il costo generale della soluzione, nonostante la sovrapposizione di computazione e comunicazione. Inoltre, la necessità di utilizzare la funzione *memcpy*, per salvare la porzione di body all'interno di un buffer prima di inviarla a tutti gli altri processi, rallenta ulteriormente la soluzione.