Università degli Studi di Salerno

Facoltà di Informatica

PROGRAMMAZIONE CONCORRENTE, PARALLELA E SU CLOUD

Progetto di corso N-Body

Professori

Prof. Vittorio Scarano

Dott. Carmine Spagnuolo

Studente

Antonio Donnarumma

Anno Accademico 2020-2021

Indice

Introduzione

N-body è un problema di tipo fisico che consiste nella predizione dei movimenti di corpi celestiali in relazione all'influenza della gravità. Generalmente l'input del problema consiste nella posizione, velocità e tempo di ogni corpo celestiale, di cui si tenta di predire la posizione attraverso l'interazione con tutti gli altri corpi. Data la natura del problema è quindi possibile effettuarne una simulazione attraverso appositi algoritmi. Uno dei più famosi è quello di Barnes-Hut, con tempo di computazione pari a O(n logn), tuttavia, ai fini dell'esame si è scelto di utilizzare l'algoritmo con tempo quadratico rispetto all'input. Questo è stato opportunamente parallelizzato mediante la libreria MPI e testato su un cluster AWS di tipo t2.xlarge composto da 8 macchine, usando al più 2 vCPUs per ogni elaboratore.

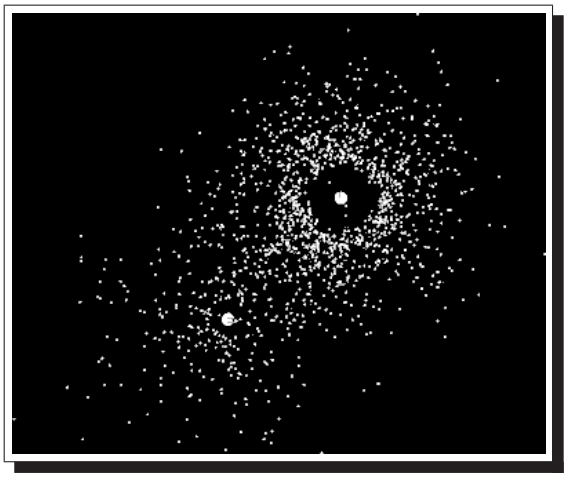

Figura 1: L'immagine di una simulazione di n-body.

La soluzione proposta

Come detto in precedenza, la soluzione proposta consiste nella parallelizzazione, mediante l'utilizzo di MPI, dell'algoritmo sequenziale fornitoci e di verificarne l'ottimizzazione (in termini di tempo di computazione) che si ottiene dall'utilizzo di più processi.

1.1 Funzionamento

L'algoritmo parallelo che risolve la problematica si può dividere in 3 macro aree:

Inizializzazione: questa fase consiste nell'inizializzazione delle variabili che saranno poi utilizzate nelle fasi successive. Sostanzialmente, dopo l'inizializzazione di MPI, vengono calcolati **da ogni processo** il numero di elementi associati ad ognuno di essi. Vengono calcolati: il numero di processi, il proprio rank, viene inizializzata la struttura body e così via. Una possibile soluzione avrebbe visto una comunicazione di tipo broadcast dal master per comunicare agli altri processi il numero di elementi per ogni singolo processo ma è stato ritenuto un inutile overhead visto che tutti gli altri processi sarebbero rimasti in attesa della comunicazione senza effettuare alcun calcolo. Ciò che invece viene fatto **solo** dal processo master è l'inizializzazione degli N-body che sono poi inviati, dividendoli a seconda del numero di elementi calcolati per ogni processo, ad ognuno degli altri processi (comprendendo sé stesso) mediante una comunicazione di tipo MPI_Scatterv.

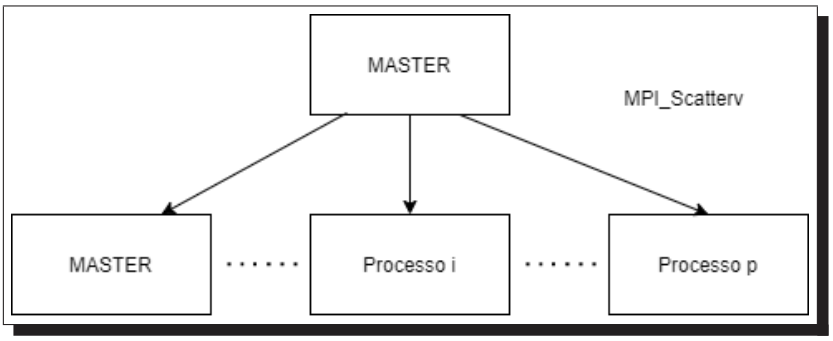

Figura 1.1: Il master suddivide gli N-body.

Il calcolo: in questa fase vengono effettuati tutti i calcoli necessari per risolvere il problema. Una particolare attenzione è stata fatta alle comunicazioni necessarie ad ogni passo dell'iterazione. Infatti, data la natura del problema è necessario che ad ogni passo ogni processo comunichi con gli altri i propri bodies e li riceva a sua volta dagli altri processi per poi calcolarne la forza. Questa fase è stata scomposta: nella prima parte dell'iterazione sono inviati i bodies, tramite una comunicazione di tipo broadcast non bloccante (MPI_Ibcast), agli altri processi e vengono postate (sempre tramite comunicazione non bloccanti) le altre richieste agli altri processi per i loro bodies. A questo punto, nel mentre si attendono le risposte, ogni processo inizia a calcolare la forza dei propri bodies per poi integrarla con quelli ricevuti dagli altri processi e terminanto quindi l'i-esimo passo.

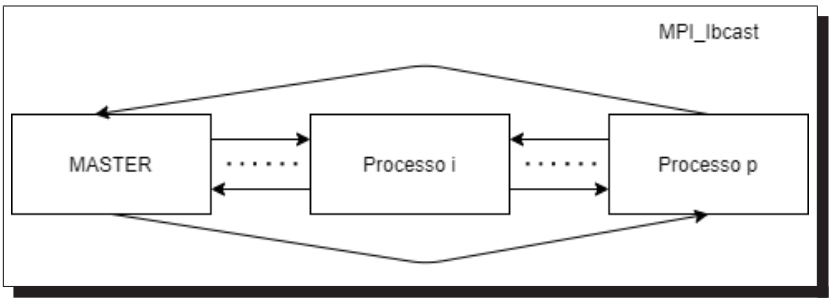

Figura 1.2: I processi si scambiano i valori dei bodies.

Raccolta dei risultati: al termine delle N iterazioni, ogni processo invia i propri bodies al processo master, che li raccoglie, scrive un file txt contenenti i valori di ogni singolo body e visualizza il tempo di esecuzione dell'algoritmo.

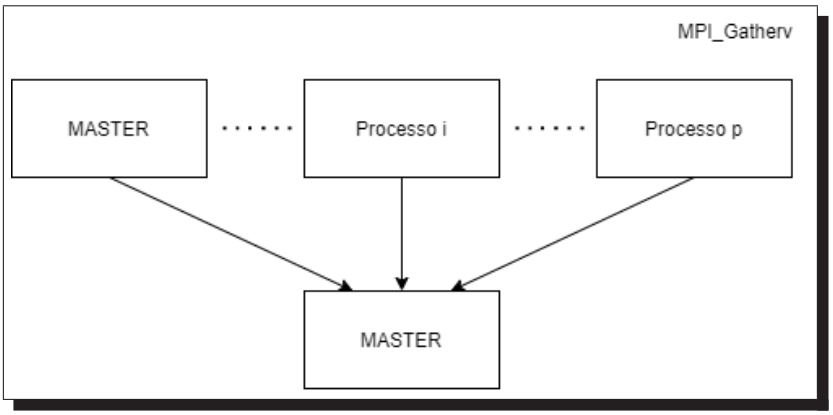

Figura 1.3: Il master raccoglie gli N-body.

L'Implementazione

Vediamo adesso alcuni dettagli relativi all'implementazione della soluzione proposta attraverso una descrizione delle funzioni presenti in ognuna delle tre aree individuate in precedenza.

2.1 Inizializzazione

Analizziamo le funzioni per la fase di inizializzazione.

2.1.1 createBodyStruct

```
1  void createBodyStruct(MPI_Datatype *bodyType)
2  {
3      MPI_Datatype oldtypes[1];
4      int blockcounts[1];
5      MPI_Aint offsets[1];
6
7      offsets[0] = 0;
8      blockcounts[0] = 6;
9      oldtypes[0] = MPI_FLOAT;
10
11     MPI_Type_create_struct(1, blockcounts, offsets, oldtypes,
           bodyType);
12     MPI_Type_commit(bodyType);
13 }
```

Questa funzione permette la creazione del tipo Body e di registrarlo come utilizzabile all'interno delle comunicazioni MPI.

2.1.2 initVariables

```
1  void initVariables(int nBodies, int *myRank, int *processes,
       int **sendCounts, int **displacements, Body **recv, int
       *myNumberOfElements, int minSize)
2  {
3      MPI_Comm_size(MPI_COMM_WORLD, processes);
4      MPI_Comm_rank(MPI_COMM_WORLD, myRank);
5
6      *sendCounts = (int *)malloc(*processes * sizeof(int));
7      *displacements = (int *)malloc(*processes * sizeof(int));
8      memset(*sendCounts, 0, *processes * sizeof(int));
9
10     initDisplsAndSendCounts(*processes, nBodies,
           *displacements, *sendCounts, minSize);
11     *myNumberOfElements = sendCounts[0][*myRank];
12     *recv = (Body *)malloc(*myNumberOfElements * sizeof(Body));
13 }
```

Questa funzione inizializza le variabili che saranno poi utilizzate nelle fasi successive. La funzione initDisplsAndSendCounts suddivide gli elementi da assegnare ad ogni processo. Particolare attenzione va fatta per la variabile minSize che viene passata in input al programma (di default vale 100) e definisce il numero **minimo** di elementi da assegnare ad ogni processo; se il numero di bodies fornito in input è minore rispetto a minSize * numeroProcessi allora non tutti i processi saranno utilizzati per l'esecuzione (gli viene assegnato il sendCount pari a zero). Questa ottimizzazione è stata fatta in previsione di possibili lanci con un numero elevato di processi, tale che l'overhead generato dalle comunicazioni supera il vantaggio della parallelizzazione rendendo l'algoritmo inefficiente. Al momento è un parametro di input, successivamente sarebbe utile che questo valore venga calcolato dinamicamente così da adattare il programma a seconda degli input.

2.1.3 initProcesses

```
1  void initProcesses(Body *recv, int *sendCounts, int
       *displacements, MPI_Datatype bodyType, int nBodies, int
       myRank, int myNumberOfElements)
2  {
3      Body *p = NULL;
4      if (myRank == MASTER)
5      {
6          int bytes = nBodies * sizeof(Body);
7          float *buf = (float *)malloc(bytes);
8          p = (Body *)buf;
9          randomizeBodies(buf, 6 * nBodies);
10     }
11
12     MPI_Scatterv(p, sendCounts, displacements, bodyType,
13                 recv, myNumberOfElements, bodyType, MASTER,
                       MPI_COMM_WORLD);
14     free(p);
15 }
```

Questa funzione permette la generazione da parte del master di tutti i bodies e la loro suddivisione per ognuno dei processi coinvolti in accordo con i valori ottenuti dalla funzione vista in precedenza.

2.2 Il calcolo

Analizziamo le funzioni per la fase di calcolo, parte principale del programma.

2.2.1 runAllIterations

```
1  void runAllIterations(Body *recv, int *sendCounts, MPI_Datatype
       bodyType, int nBodies, int myNumberOfElements, int myRank,
       int processes, int nIters)
2  {
```

```
3       const float dt = 0.01f;
4       int currentIteration = 0;
5       while (currentIteration < nIters)
6       {
7           Body *otherProcessesBody = (Body *)malloc((nBodies -
                myNumberOfElements) * sizeof(Body));
8           MPI_Request *requests = (MPI_Request *)malloc(processes
                * sizeof(MPI_Request));
9
10          runSingleIteration(otherProcessesBody, recv, bodyType,
                requests, sendCounts, myRank, processes,
                myNumberOfElements, dt);
11
12          free(otherProcessesBody);
13          free(requests);
14          currentIteration++;
15      }
16  }
```

Questa funzione è il corpo della elaborazione. Sostanzialmente esegue il singolo step per N (fornito in input con 10 come valore di default) iterazioni ed inizializza l'array di richieste necessarie all'i-esimo step e l'array di bodies che sarà riempito con i valori forniti dagli altri processi.

2.2.2 getAndSendBodiesToOtherProcesses

```
1  void getAndSendBodiesToOtherProcesses(Body
       *otherBodiesBasicAddress, MPI_Datatype bodyType,
       MPI_Request *requests, int myRank, int processes, int
       *sendCounts, int myNumberOfElements, Body *recv)
2  {
3      for (int i = 0; i < processes; i++)
4      {
5          if (i != myRank)
6          {
7              MPI_Ibcast(otherBodiesBasicAddress, sendCounts[i],
                   bodyType, i, MPI_COMM_WORLD, &requests[i]);
8              otherBodiesBasicAddress += sendCounts[i];
9          }
10         else
11         {
12             MPI_Ibcast(recv, myNumberOfElements, bodyType,
                   myRank, MPI_COMM_WORLD, &requests[i]);
13         }
14     }
15 }
```

Questa funzione effettua le comunicazioni necessarie ad ogni passo attraverso delle broadcast non bloccanti. Sostanzialmente, considerando che ogni processo ha bisogno delle particelle degli altri processi, ognuno di essi invia le proprie e posta una broadcast per ricevere le altre. Una valida alternativa era rappresentata dalla MPI_Allgatherv, tuttavia, si è scelto di non utilizzarla visto che, ad ogni iterazione, si sarebbe dovuto attendere la ricezione di tutti i bodies da tutti gli altri processi per poi poter effettuare il calcolo (cosa che con la soluzione proposta non avviene, visto che, appena una broadcast viene completata si procede con l'applicazione della forza).

2.2.3 applyBodyForce

```
1  void applyBodyForce(Body *myBody, int myBodySize, Body
       *otherBodies, int otherBodiesSize, float dt)
2  {
3      for (int i = 0; i < myBodySize; i++)
4      {
5          float Fx = 0.0f;
6          float Fy = 0.0f;
7          float Fz = 0.0f;
8
9          for (int j = 0; j < otherBodiesSize; j++)
10         {
11             float dx = otherBodies[j].x - myBody[i].x;
12             float dy = otherBodies[j].y - myBody[i].y;
13             float dz = otherBodies[j].z - myBody[i].z;
14             float distSqr = dx * dx + dy * dy + dz * dz +
                   SOFTENING;
15             float invDist = 1.0f / sqrtf(distSqr);
16             float invDist3 = invDist * invDist * invDist;
17
18             Fx += dx * invDist3;
19             Fy += dy * invDist3;
20             Fz += dz * invDist3;
21         }
22
23         myBody[i].vx += dt * Fx;
24         myBody[i].vy += dt * Fy;
25         myBody[i].vz += dt * Fz;
26     }
27 }
```

Questa funzione permette di applicare la forza dei bodies "otherBodies" sui bodies passati come primo parametro (myBody), ossia i body locali dell'i-esimo processo. É utilizzata P volte, una quando viene applicata la forza dei bodies locali ed altre p-1 volte per ogni ricezione dei bodies degli altri processi.

2.2.4 waitForOtherBodiesAndApplybodyForce

```
1  void waitForOtherBodiesAndApplybodyForce(MPI_Request *requests,
       Body *otherProcessesBodies, Body *recv, int *sendCounts,
       int myRank, int processes, int myNumberOfElements, float dt)
2  {
3      int i = 0;
4      while (i != processes)
5      {
6          int index;
7          MPI_Status status;
8          MPI_Waitany(processes, requests, &index, &status);
9          i++;
10         if (index != myRank)
11         {
12             Body *actualBody =
                   getBodyAddressForActualIndex(otherProcessesBodies,
                   myRank, index, sendCounts);
13             applyBodyForce(recv, myNumberOfElements, actualBody,
                   sendCounts[index], dt);
14         }
15     }
16 }
```

Questa funzione attende il completamento di una operazione di MPI_Ibcast, calcola (attraverso la funzione getBodyAddressForActualIndex) l'indice da dove leggere gli elementi ricevuti dal processo con valore pari ad index (valorizzato dalla operazione MPI_Waitany) ed applica la forza degli elementi ricevuti agli elementi locali.

2.2.5 integrateBodyPosition

```
1  void integrateBodyPosition(Body *recv, int myNumberOfElements,
       float dt)
2  {
3      for (int i = 0; i < myNumberOfElements; i++)
4      {
5          recv[i].x += recv[i].vx * dt;
6          recv[i].y += recv[i].vy * dt;
7          recv[i].z += recv[i].vz * dt;
8      }
9  }
```

Questa funzione viene chiamata al termine di un singolo passo (una volta calcolate le varie forze) e modifica le posizioni dei singoli bodies in relazione alla velocità precedentemente calcolata.

2.2.6 collectIterationsResult

```
1  void collectIterationsResult(Body **processedBodies, Body
       *recv, int *sendCounts, int *displacements, MPI_Datatype
       bodyType, int nBodies, int myRank, int myNumberOfElements)
2  {
3      *processedBodies = NULL;
4
5      if (myRank == MASTER)
6      {
7          *processedBodies = (Body *)malloc(nBodies *
               sizeof(Body));
8      }
9
10     MPI_Gatherv(recv, myNumberOfElements, bodyType,
           *processedBodies, sendCounts, displacements, bodyType,
           MASTER, MPI_COMM_WORLD);
11 }
```

Questa funzione viene chiamata al termine di tutte le iterazioni e semplicemente colleziona, all'interno del processo master, il risultato di tutte le elaborazioni dei vari processi. Questi risultati sono poi passati ad una funzione che ne salva il contenuto all'interno di un file txt e visualizza il tempo totale della esecuzione.

Istruzioni per il lancio

Per poter eseguire il progetto è necessario lanciare due comandi:

```
1    mpicc nbody.c -o nbody -lm
2     mpirun -np {numero processi} nbody {numero bodies} {numero
       iterazioni} {dimensione minSize}
```

Dopo aver lanciato il comando di compilazione, prima di eseguire il comando di run è necessario definire quattro parametri:

- Numero processi: definisce il numero di processi con cui lanciare l'esecuzione.

- Numero bodies: definisce il numero di corpi che verranno generati casualmente.

- Numero iterazioni: definisce il numero di passi dell'esecuzione corrente.

- Dimensione minSize: definisce la dimensione minima per poter considerare un dato processo utilizzabile per la esecuzione corrente. Spiegato qui: initVariables.

Correttezza

Al fine di testare la correttezza dell'algoritmo proposto sono state implementate due soluzioni: una sequenziale ed una parallela con uso di MPI. Entrambe le soluzioni, al termine della loro esecuzione scrivono su un file di tipo testuale i loro risultati ed è proprio grazie a questo che è stato possibile garantire la correttezza dell'algoritmo parallelo che, al variare del numero di processi, restituisce sempre il medesimo risultato (al netto di alcune approssimazioni fatte da MPI durante le comunicazioni).

4.1 Algoritmo sequenziale

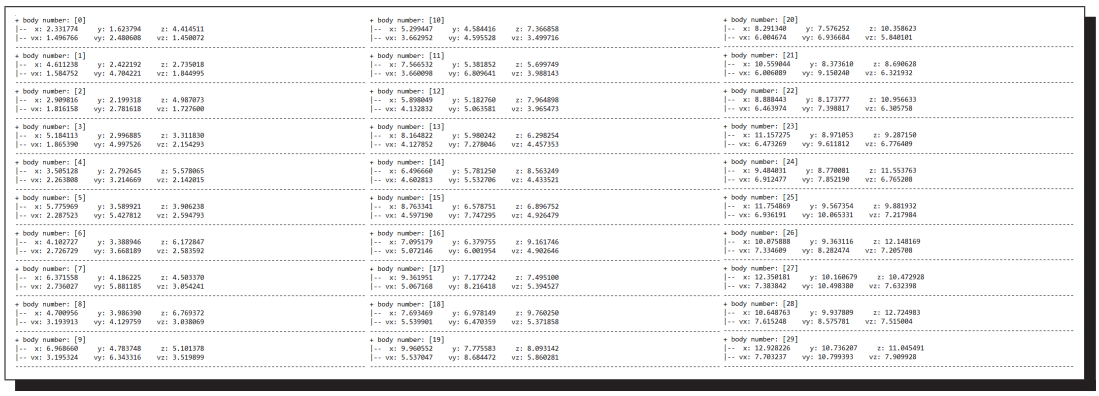

Figura 4.1: Simulazione con algoritmo sequenziale

4.2 Algoritmo parallelo

Per l'algoritmo parallelo sono state effettuate tre prove per la correttezza: singolo processo, due processi e quattro processi. Di seguito i files prodotti dalle loro esecuzioni.

4.2.1 1 processo

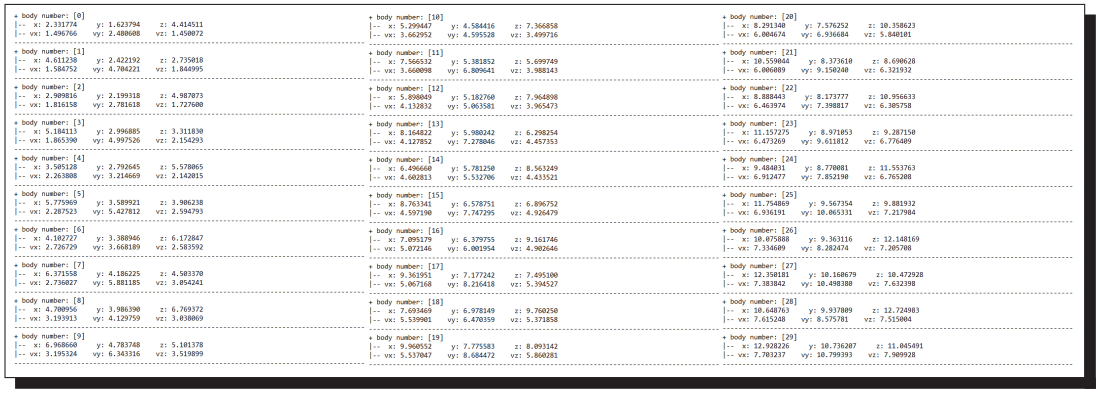

Figura 4.2: Simulazione con algoritmo parallelo e un processo

4.2.2 2 processi

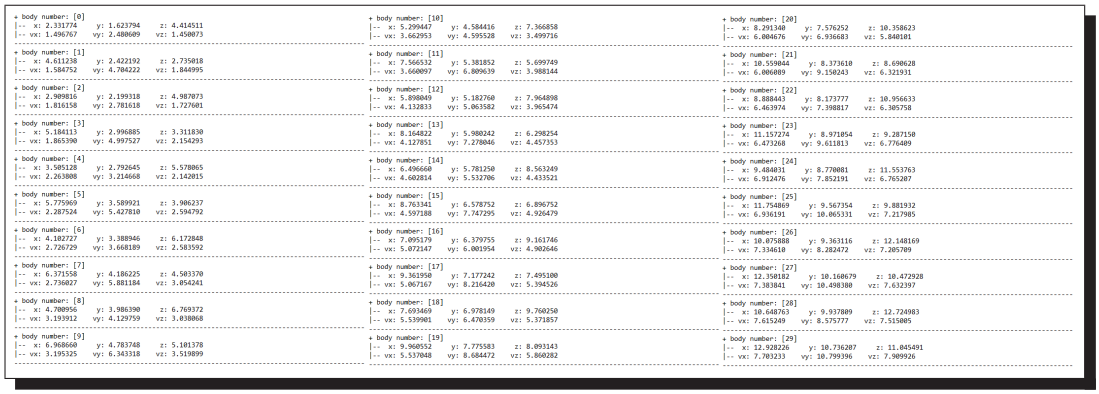

Figura 4.3: Simulazione con algoritmo parallelo e due processi

4.2.3 4 processi

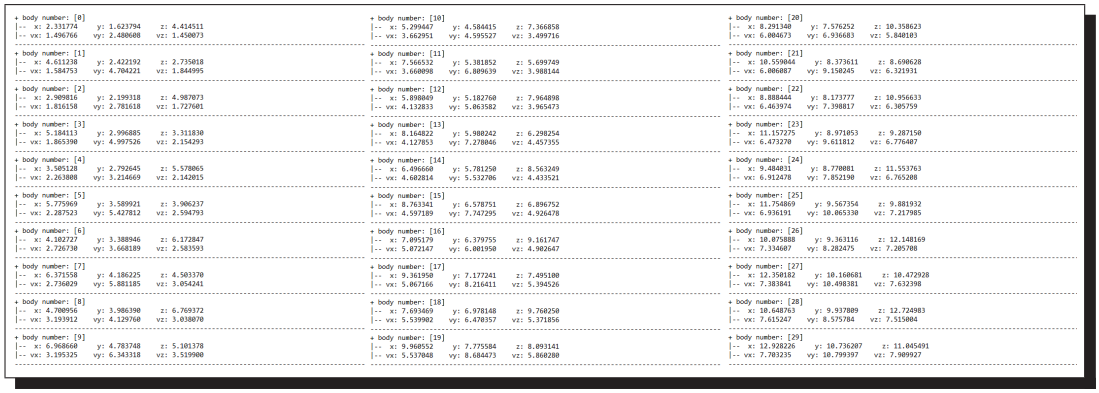

Figura 4.4: Simulazione con algoritmo parallelo e quattro processi

Benchmarks

Il benchmark della soluzione proposta è stato effettuato su un cluster di macchine t2.xlarge di AWS e sono state prese in considerazione sia la scalabilità forte che la scalabilità debole. Sono stati effettuati un totale di 12 esperimenti, 9 per quanto concerne la scalabilità forte al variare del numero di iterazioni (10, 25 e 50) e numero di bodies forniti in input (10k, 25k e 50k). I restanti 3 esperimenti riguardano la scalabilità debole che ha visto come valore iniziale 1.25k, 2.5k e 5k bodies con un numero di iterazioni fisso, pari a 25. Vediamo i risultati ottenuti.

5.1 Scalabilità forte

Di seguito i grafici ottenuti per l'analisi sulla scalabilità forte dell'algoritmo.

5.1.1 10 iterazioni

Simulazione con 10k bodies e 10 iterazioni

Processi	1	2	3	4	5	6	7	8
Tempo	19,63	9,81	7,42	5,50	4,44	3,70	3,15	2,75
Processi	9	10	11	12	13	14	15	16
Tempo	2,45	2,20	2,00	1,83	1,70	1,58	1,47	1,40

Simulazione con 25k bodies e 10 iterazioni

Processi	1	2	3	4	5	6	7	8
Tempo	122,91	61,48	46,52	34,54	27,60	23,22	20,69	18,18
Processi	9	10	11	12	13	14	15	16
Tempo	15,90	13,77	12,57	11,55	10,65	9,93	9,26	8,71

Simulazione con 50k bodies e 10 iterazioni

Processi	1	2	3	4	5	6	7	8
Tempo	491,73	245,78	185,85	138,04	110,43	94,35	81,35	78,02
Processi	9	10	11	12	13	14	15	16
Tempo	63,70	57,79	56,15	50,75	47,84	47,38	43,54	43,17

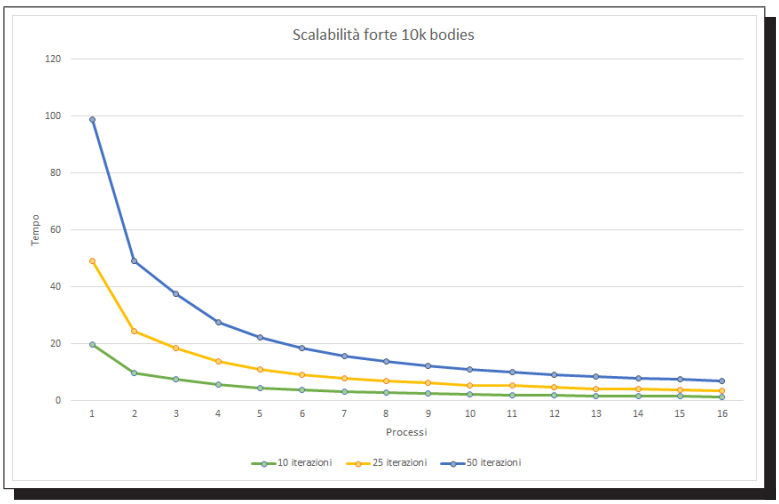

Figura 5.1: Simulazione con 10k, 25k e 50k bodies e 10 iterazioni

5.1.2 25 iterazioni

Simulazione con 10k bodies e 25 iterazioni

Processi	1	2	3	4	5	6	7	8
Tempo	49,09	24,53	18,42	13,80	11,08	9,24	7,89	6,92
Processi	9	10	11	12	13	14	15	16
Tempo	6,16	5,49	5,28	4,63	4,24	3,98	3,69	3,46

Simulazione con 25k bodies e 25 iterazioni

Processi	1	2	3	4	5	6	7	8
Tempo	307,42	153,67	115,13	86,34	68,95	57,84	49,53	44,55
Processi	9	10	11	12	13	14	15	16
Tempo	39,31	34,6	31,27	28,76	26,55	24,73	23,10	21,56

Simulazione con 50k bodies e 25 iterazioni

Processi	1	2	3	4	5	6	7	8
Tempo	1232,99	616,71	461,17	345,26	275,36	231,97	199,89	177,83
Processi	9	10	11	12	13	14	15	16
Tempo	158,65	143,18	138,74	124,26	115,28	103,94	96,68	91,16

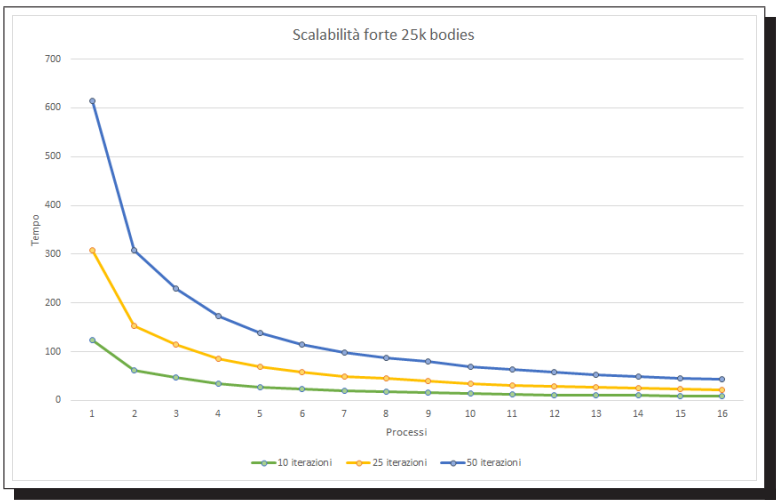

Figura 5.2: Simulazione con 10k, 25k e 50k bodies e 25 iterazioni

5.1.3 50 iterazioni

Simulazione con 10k bodies e 50 iterazioni								
Processi	1	2	3	4	5	6	7	8
Tempo	98,71	49,20	37,49	27,66	22,16	18,50	15,72	13,79
Processi	9	10	11	12	13	14	15	16
Tempo	12,24	11,09	10,03	9,18	8,55	7,93	7,42	6,96

Simulazione con 25k bodies e 50 iterazioni								
Processi	1	2	3	4	5	6	7	8
Tempo	615,06	307,44	229,47	172,70	138,21	115,37	99,16	87,35
Processi	9	10	11	12	13	14	15	16
Tempo	79,46	69,14	63,21	57,56	53,30	49,29	46,09	43,23

Simulazione con 50k bodies e 50 iterazioni								
Processi	1	2	3	4	5	6	7	8
Tempo	2466,57	1232,97	917,13	690,1	552,07	462,69	397,06	351,97
Processi	9	10	11	12	13	14	15	16
Tempo	312,99	284,11	258,71	238,95	218,9	207,9	193,15	182,37

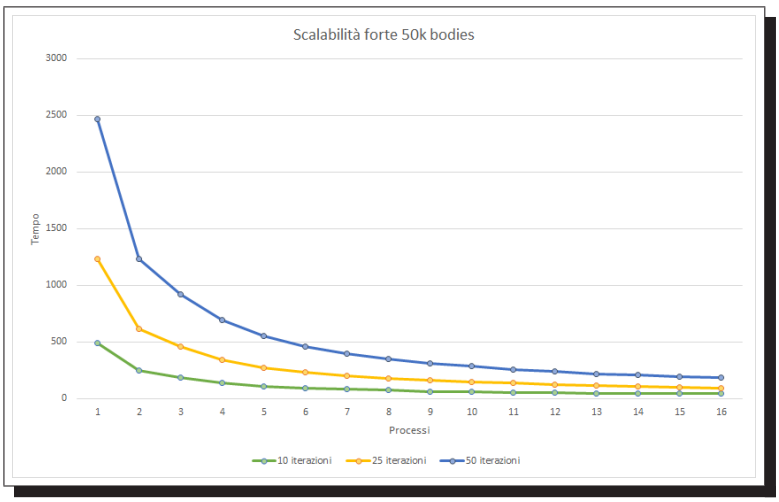

Figura 5.3: Simulazione con 10k, 25k e 50k bodies e 50 iterazioni

5.2 Scalabilità debole

Di seguito i grafici ottenuti per l'analisi sulla scalabilità debole dell'algoritmo.

Simulazione con 1.25k bodies per processo e 25 iterazioni

Processi	1	2	3	4	5	6	7	8
Bodies	1.25k	2.5k	3.75k	5k	6.25k	7.5k	8.75k	10k
Tempo	0,77	1,53	2,57	3,43	4,3	5,16	6,02	6,89
Processi	9	10	11	12	13	14	15	16
Bodies	11.25k	12.5k	13.75k	15k	16.25k	17,5k	18.75k	20k
Tempo	7,77	8,63	9,45	10,37	11,16	12,03	12,97	13,75

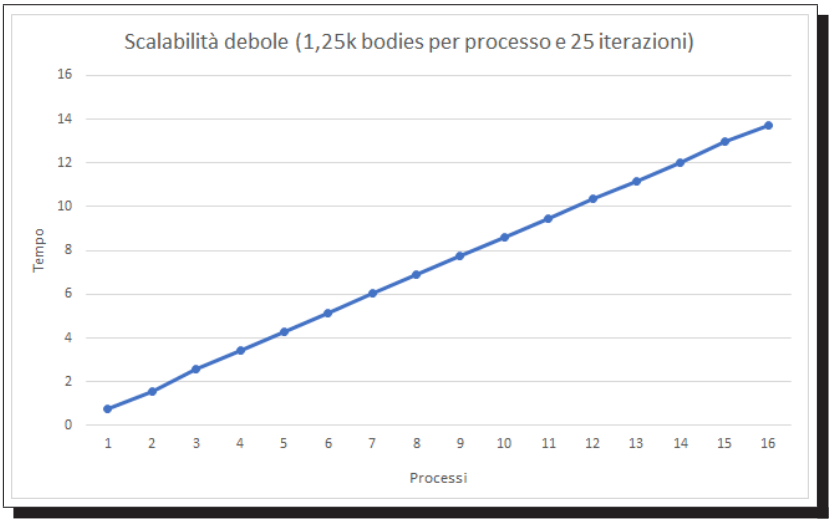

Figura 5.4: Simulazione con k iniziale pari a 1.25k bodies e 25 iterazioni

Simulazione con 2.5k bodies per processo e 25 iterazioni

Processi	1	2	3	4	5	6	7	8
Bodies	2.5k	5k	7.5k	10k	12.5k	15k	17.5k	20k
Tempo	3,08	6,14	10,29	13,82	17,24	20,76	24,1	27,68
Processi	9	10	11	12	13	14	15	16
Bodies	22.5k	25k	27.5k	30k	32.5k	35k	37.5k	40k
Tempo	31,1	34,43	38,09	41,38	44,7	48,43	51,84	55,35

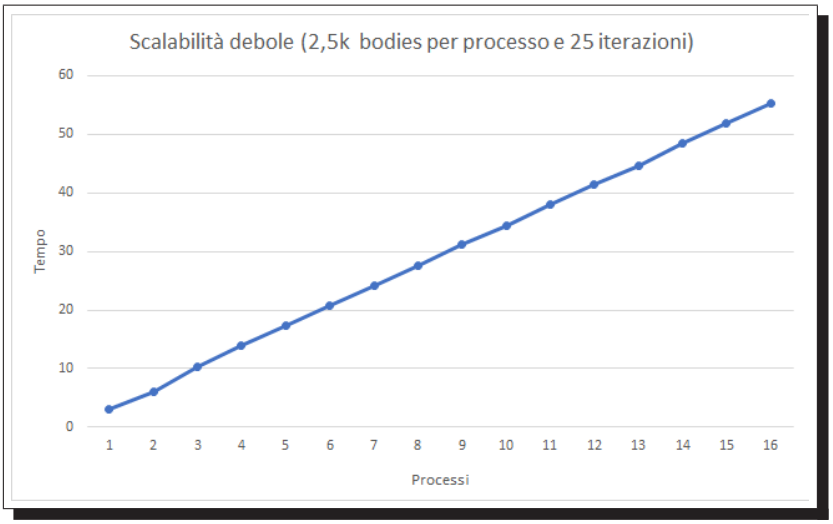

Figura 5.5: Simulazione con k iniziale pari a 2.5k bodies e 25 iterazioni

Simulazione con 5k bodies per processo e 25 iterazioni

Processi	1	2	3	4	5	6	7	8
Bodies	5k	10k	15k	20k	25k	30k	35k	40k
Tempo	12,29	24,57	41,78	55,21	68,99	82,67	96,09	112,38
Processi	9	10	11	12	13	14	15	16
Bodies	45k	50k	55k	60k	65k	70k	75k	80k
Tempo	130,98	143,49	156,02	171,95	198,68	218,45	247,16	261,19

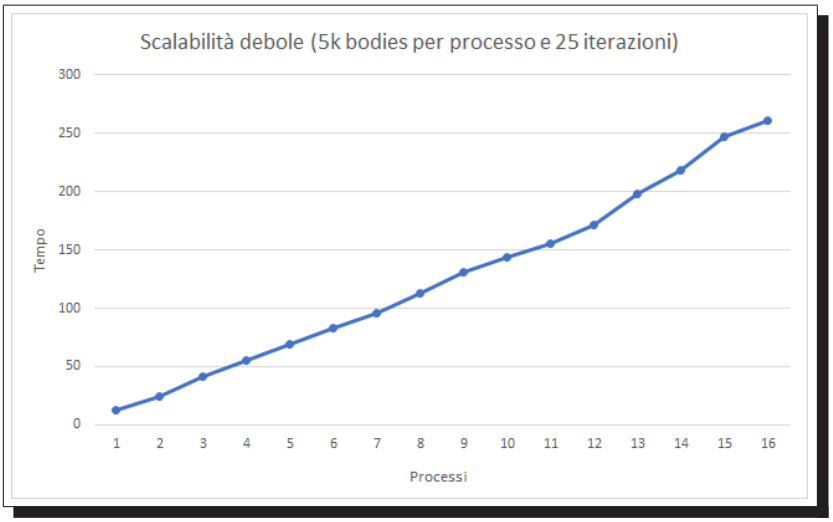

Figura 5.6: Simulazione con k iniziale pari a 5k bodies e 25 iterazioni

Conclusioni

I risultati ottenuti sono molto soddisfacenti, infatti, lo speedup ottenuto dall'utilizzo della parallelizzazione è sempre in salita (anche se di poco) come è possibile notare dai grafici della scalabilità forte e debole. Tuttavia una nota importante è dovuta all'ambiente utilizzato per effettuare i benchmarks, ossia un cluster t2.xlarge composto da 8 macchine (con due vCPUs per elaboratore) che ha certamente influenzato in positivo i risultati, riducendo di molto l'overhead delle comunicazioni. Infatti, su prove effettuate in locale, lo speedup dell'algoritmo diminuisce quando, a parità di bodies, il numero di processi cresce. Una ottimizzazione è possibile tramite l'utilizzo del valore minSize che permette di non utilizzare i processi in eccesso, tuttavia, al momento viene passato come valore in input al programma ma, in una possibile implementazione futura, come detto in precedenza, dovrebbe essere ottenibile dinamicamente in modo da rendere l'algoritmo sempre performante.