N-Body

Università degli studi di Salerno

Laurea Magistrale in Informatica

Programmazione Parallela, Concorrente e su Cloud

1. Introduzione

Nel problema N-body, abbiamo bisogno di trovare le posizioni e le velocità di un gruppo di particelle che interagiscono in un periodo di tempo. Ad esempio, come un astrofisico potrebbe voler conoscere le posizioni e le velocità di un gruppo di stelle. Allo stesso modo, un chimico potrebbe voler conoscere le posizioni e le velocità di un gruppo di molecole o atomi.

Un N-body solver è un programma che trova la soluzione ad un problema N-body attraverso la simulazione del comportamento delle particelle. Gli input al problema sono la massa, la posizione e la velocità di ogni particella all'inizio della simulazione. Gli output tipicamente sono la posizione e la velocità di ogni particella in una sequenza di tempo specificata dall'utente, oppure semplicemente la posizione e la velocità di ogni particella alla fine del tempo specificato dall'utente.

2. Descrizione della soluzione

Nella soluzione proposta il programma possiede un numero di iterazioni da effettuare e di particelle da computare. Alle particelle sono assegnate in modo casuale le posizioni e le velocità iniziali. Il master invia agli slave l'array di particelle in modo che tutti processi possano calcolare le variazioni delle grandezza. Successivamente il master raccoglie i risultati, aggiorna l'array e nell'iterazione successiva invierà l'array aggiornato. Infine restituisce all'utente i risultati ottenuti.

3. Dettagli dell'implementazione

Le particelle, descritte attraverso una struttura Body, sono raccolte in un array di particelle. Ad ogni iterazione, il master, con l'operazione di MPI_Bcast, invia ad ogni processo l'array di particelle aggiornato. Ogni processo, attraverso due array di interi definiti precedentemente, conosce quante particelle computare e da dove partire, quindi esegue il calcolo solo sulla porzione di sua competenza. Successivamente, attraverso l'operazione MPI_Gatherv, il master raccoglie i risultati ottenuti aggiornando l'array di particelle. All'iterazione successiva il master invierà l'array aggiornato. Viene utilizzata la MPI_Gatherv poiché le particelle potrebbe non essere perfettamente divisibili tra i processi. Infine il master si occuperà di stampare le posizioni e velocità finali di ogni slave ed il tempo impiegato per l'esecuzione. Per parallelizzare l'algoritmo Nbody, il carico di lavoro viene equamente distribuito tra i processi coinvolti nel cluster, rendendo più equa possibile a distribuzione anche in caso il numero di particelle non sia perfettamente divisibile tra i processi. Per i benchmark sono state utilizzate istanze AWS di tipo t2.large.

4. Codice

3.1 Variabili

Viene definita una struttura dati Body utilizzata per rappresentare la particella. La struttura è composta dai tre dati riguardanti la posizione nello spazio ed i tre dati riguardanti la velocità.

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

Viene dichiarato un tipo di dato MPI_body da utilizzare come buffer per inviare i valori di body. Viene dichiarato anche un array di tipo body per gestire le comunicazioni tra slave e master.

```
int rank;                //rank del processo int process;
//numero di processi int iteration = 20;      //numero di
iterazioni int bodies;               //numero di particelle
inserito dall'utente const float dt = 0.1f;    //time step
MPI_Datatype MPIbody;     //tipo di dato MPI
Body *collection;         //array di particelle
```

3.2 Funzioni

La funzione bodyForce si occupa di calcolare le variazioni di velocità e posizione di ogni particelle confrontandola con le particelle vicine. Prende in input l'array di particelle da computare, da dove iniziare a computazione ed il numero di particelle.

```
void bodyForce(Body *p, int start, int lenght)
{
    for (int i = start; i < start + lenght; i++)
    {
        float Fx = 0.0f;
        float Fy = 0.0f;
        float Fz = 0.0f;
```

```
        for (int j = 0; j < bodies; j++)
        {
            float dx = p[j].x - p[i].x;
            float dy = p[j].y - p[i].y;
            float dz = p[j].z - p[i].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[i].vx += dt * Fx;
        p[i].vy += dt * Fy;
        p[i].vz += dt * Fz;
    }

    for (int l = start; l < start + lenght; l++)
    {
        p[l].x += p[l].vx * dt;
        p[l].y += p[l].vy * dt;
        p[l].z += p[l].vz * dt;
    }
}
```

La funzione randomizeBodies si occupa di generare le particelle in modo casuale all'interno dell'array collection. Prende in input l'array di particelle e la quantità di particelle da generare.

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

La funzione printBodies riceve in input l'array contente le particelle e viene utilizzata per stampare le posizioni e le velocità delle particelle.

```
void printBodies(Body *body)
{
  for (int i = 0; i < bodies; i++)
  {
•    printf("\n %d\n", i);
•    printf("x= %f\ty= %f\tz= %f\tvx= %f\tvy= %f\tvz= %f\t\n\n", body[i].x,
body[i].y, body[i].z, body[i].vx, body[i].vy, body[i].vz);
•    fflush(stdout);
  }
}
```

3.3 Main

Inizialmente viene inizializzato l'ambiente MPI. Attraverso Viene creato un tipo contiguo e, in modo da poter utilizzarlo nelle comunicazioni, viene effettuata la MPI_Type_commit.

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &process);
MPI_Type_contiguous(6, MPI_FLOAT, &MPIbody);
MPI_Type_commit(&MPIbody);
```

Successivamente ottengo il numero di particelle inserito dall'utente, viene allocata la memoria degli array e vengono definite le variabili da utilizzare. Vengono dichiarati due array di interi dspl ed sc per dividere in modo equo le particelle e gestire, nel caso vi sia, il resto della divisione tra il numero di particelle ed i processi e ne vengono allocate le memorie. In seguito viene allocata la memoria dell'array di particelle utilizzato per la comunicazione.

```
bodies = atoi(argv[1]);                     //ottengo il numero di particelle
collection = malloc(sizeof(Body) * bodies); //alloco l'insiemi delle
particelle int part = bodies / process;               //calcolo il numero di
particelle da                                                  //assegnare
ad ogni processo int rest = bodies % process;               //calcolo il
resto int sum = 0;                                 //sum viene utilizzato per
la suddivisione in                                           //caso di resto
int *dspl, *sc;                             //array di interi per gestire il
resto
dspl = malloc(sizeof(int) * process);       //alloco l'array che descrive lo
spostamento sc = malloc(sizeof(int) * process);         //alloco l'array che
descrive quanti elementi                                       //inviare
double times;                               //utilizzato per calcolare il
tempo di                                                 //esecuzione

times = MPI_Wtime();                        //calcolo il tempo di esecuzione
```

Successivamente vengono calcolate le particelle destinate ad ogni processo.

```
for (int i = 0; i < process; i++)
{
• sc[i] = part; //ad ogni processo viene assegnato lo stesso numero di
particelle
• if (rest > 0) //finchè c'è resto alcuni processi riceveranno una particella in
più
• {
•   sc[i]++;
•   rest--;
• }
• dspl[i] = sum; //sum tiene il conto del passo a cui il processo di
suddivisione è giunto e lo assegna ad ogni processo
• sum += sc[i];  //sum viene aggiornato in base a quante particelle il processo
deve calcolare
}
```

Vengono inizializzate le particelle.

```
randomizeBodies(collection, bodies);
```

A questo punto, ad ogni iterazione, con MPI_Bcast il master diffonde l'array di particelle. Ogni processo, compreso il master, esegue la bodyForce sulla sua parte. A questo punto i risultati vengono raccolti tramite una MPI_Gatherv poiché i processi potrebbero aver computato un numero variabile di particelle.

```
for (int iter = 0; iter < iteration; iter++) //ad ogni iterazione i processi si
occupano                                          //di calcolare le nuove
posizioni e velocità
{
• MPI_Bcast(collection, bodies, MPIbody, 0, MPI_COMM_WORLD); //invio l'array

• bodyForce(collection, dspl[rank], sc[rank]); //ogni processo calcola la sua
porzione

• MPI_Barrier(MPI_COMM_WORLD); //aspetto che tutti terminano

• MPI_Gatherv(&collection[dspl[rank]], sc[rank], MPIbody, collection, sc, dspl,
MPIbody, 0, MPI_COMM_WORLD);                          //vengono raccolti i
risultati
}
```

Una volta terminata la ricezione, il master calcola il tempo di esecuzione. Attraverso printBodies stampa l'array delle particelle computate ed il tempo impiegato.

```
if (rank == 0) //il master stampa le posizioni finali ed il tempo impiegato
{
  double timee = MPI_Wtime();
  double end = timee - times;
  printBodies(collection);
  printf("\nTime: %f\n", end);
  fflush(stdout);
}
```

Infine vengono rilasciate le memorie allocate e viene terminata l'esecuzione.

```
MPI_Type_free(&MPIbody); //libera la memoria allocata dal tipo contiguo
free(collection);        //dealloco l'array delle particelle
free(sc);                //dealloco gli array dei contatori
free(dspl);
MPI_Finalize();         //termino l'esecuzione di MPI
return 0;
```

Per la compilazione del sorgente utilizzare: **mpicc**

nbody.c -lm -std=c99 -o nbody

Per l'esecuzione:

mpirun -np numero_processi nbody numero_particelle

5. Benchmark

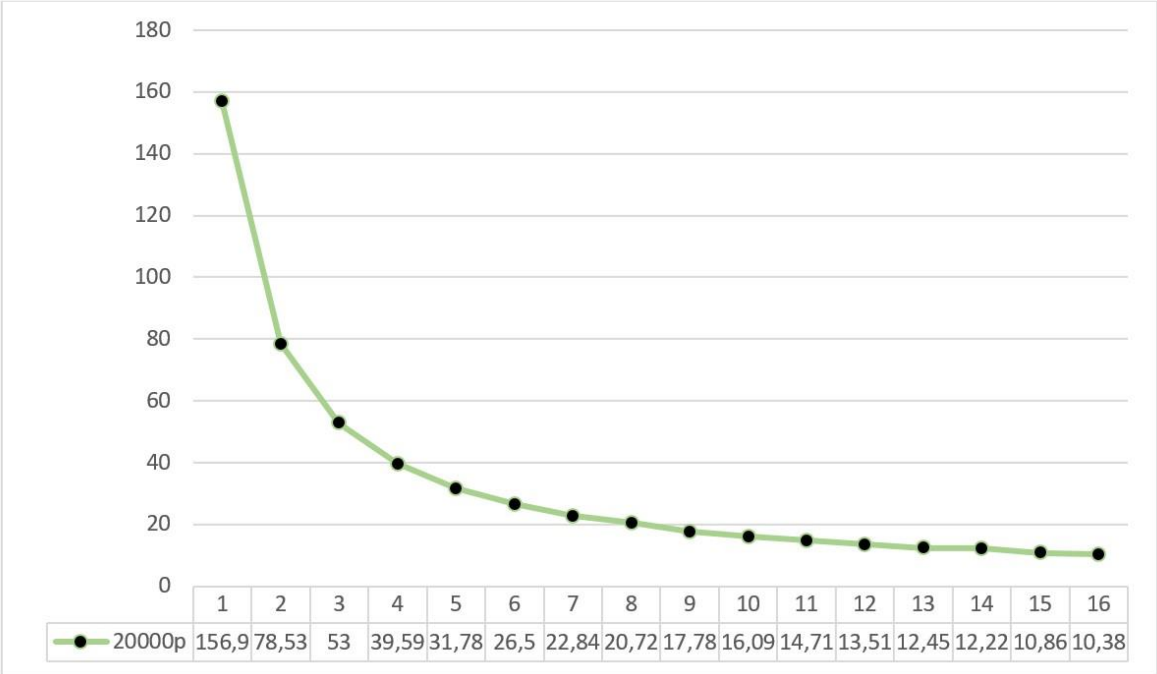
I benchmark sono stati effettuati su 8 istanze t2.large (2vCPU) di AWS. I test sono stati effettuati variando il numero di particelle, ma con un numero di iterazioni fissato a 20. I test condotti riguardano sia il variare del numero di processori, sia del numero di particelle.

Configurazione:

- 8 istanze EC2 t2.large Ubuntu Server 18.04 LTS (HVM), SSD Volume Type 16
- Processori

5.1 Strong scaling

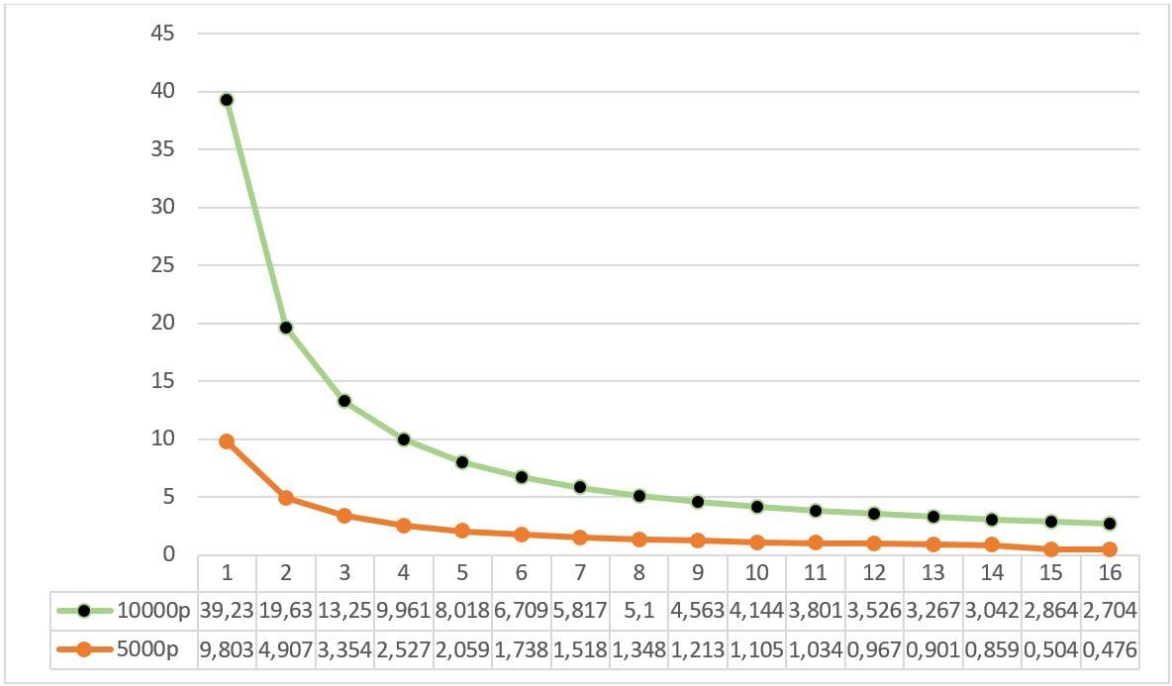
In questa fase di test, il numero di particelle è rimasto costante, ma è variato il numero di processori. Sono state inserite 20000 particelle ed i i risultati sono stati ottenuti variando il numero di processori da 1 a 16.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
20000p	156,9	78,53	53	39,59	31,78	26,5	22,84	20,72	17,78	16,09	14,71	13,51	12,45	12,22	10,86	10,38

5.2 Weak scaling

In questa fase di test si è tenuto conto sia della variazioni del carico di lavoro su un numero costante di processori, sia della variazione degli stessi. Sono state considerate 10000 e 5000 particelle. I test sono stati effettuati su configurazioni da 1 a 16 processori.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
10000p	39,23	19,63	13,25	9,961	8,018	6,709	5,817	5,1	4,563	4,144	3,801	3,526	3,267	3,042	2,864	2,704
5000p	9,803	4,907	3,354	2,527	2,059	1,738	1,518	1,348	1,213	1,105	1,034	0,967	0,901	0,859	0,504	0,476

6. Conclusione

Per quanto riguarda lo strong scaling, avendo un input fissato (in questo caso il numero di particelle), all'aumentare dei processori si ha un ridimensionamento del carico sul singolo processore. Idealmente l'aumentare del numero di processori dovrebbe portare ad un'accelerazione sul tempo di esecuzione data da una costante n, ma bisogna tener conto anche del tempo che viene speso in fase di comunicazione. Quindi, nello strong scaling, l'obiettivo è di ridurre il tempo di esecuzione attraverso l'utilizzo di una macchina sempre più potente. Nei test che sono stati effettuati, si è partiti da 1 processore fino all'esecuzione su 16 processori. Il problema nbody si presta in modo ottimo alla strong scaling, infatti è stato evidenziato come su un input di 20000 particelle vi sia un significativo aumento del tempo di esecuzione da 1 ad 12 processori Successivamente il miglioramento è stato più discreto.

Per quanto riguarda il weak scaling, la variazione dell'input combinata con la variazione dei processori si traduce in un carico di lavoro costante per ogni processore. Nel caso di nbody il problema non è ben predisposto a questo tipo di scalabilità. Questo poiché, essendo che ogni particella viene computata in riferimento a tutte le restanti particelle, nonostante la suddivisione tra processori resti costante, ogni processore dovrà computare per ogni particella assegnata un maggior numero di variazioni. Di conseguenza il carico di lavoro per processore aumenterà all'aumentare dell'input. Ad esempio, dai grafici ottenuti, è possibile notare come per un input di 20000 particelle e 4 processori il tempo di esecuzione sia di 39,6 secondi circa. Considerano l'esecuzione di 10000 particelle su 2 processo il tempo di esecuzione si riduce a 19,6 secondi, mentre per un processore su 5000 particelle vi è un ulteriore calo ed è uguale a 9,8 secondi.