N-body

Progetto di Programmazione Parallela Concorrente e su Cloud

Nome: Antonello Cognome: Luppolo

Sommario

Introduzione

In generale il problema N-body consiste nel prevedere il moto (posizione e velocità) di un gruppo di *n* oggetti che interagiscono indipendentemente l'uno dall'altro, sotto l'influenza di forze fisiche, come ad esempio la forza di gravità. Molti fenomeni fisici, astrofisici e chimici, possono essere simulati con un sistema di particelle in cui ogni particella interagisce con le altre secondo leggi fisiche. Per studiare questi fenomeni vengono utilizzati programmi che simulano il comportamento delle particelle. Tali programmi in:

- **input** prendono un insieme di particelle, dove per ognuna viene specificata la sua posizione nello spazio e la sua velocità all'inizio della simulazione;
- **output** restituiscono le posizioni e velocità di ciascuna particella alla fine della simulazione.

Per simulare il comportamento di n corpi sono stati implementati diversi algoritmi con diversi gradi di complessità e approssimazione. Tra questi prendiamo in considerazione il codice realizzato da Mark Harris, reperibile al seguente link: mini-nbody. Si tratta di un programma scritto in linguaggio C che:

- esegue la simulazione su un numero di particelle che può essere specificato dall'utente tramite riga di comando, oppure su un numero predefinito di particelle, nel caso in cui non venisse fornito alcun input.
- tramite un algoritmo pseudocasuale vengono inizializzati i campi di ogni particella

- la simulazione prevede 10 iterazioni dove ad ognuna, in modo sequenziale ogni particella viene fatta interagire con tutte le altre per calcolare la forza e la posizione che assumerà all'iterazione successiva;

La soluzione fornita da questo programma è quadratica nel numero di particelle. È possibile ottimizzare tale codice utilizzando la libreria OpenMPI, in modo da parallelizzare il calcolo.

Descrizione soluzione proposta

Di seguito viene mostrata una versione ottimizzata del programma sopra citato che fa uso della libreria OpenMPI, mediante la quale si va a distribuire il calcolo della forza delle particelle tra n processi nel seguente modo:

- Innanzitutto i processi comunicano tra loro attraverso il communicator MPI_COMM_WORLD. All'interno del gruppo associato a questo communicator i processi vengono distinti in base al loro rank in due "categorie":
    - MASTER: un solo processo appartiene a questa categoria e viene identificato con rank = 0;
    - SLAVE: tutti gli altri invece appartengono a quest'altra categoria ed hanno rank > 0;
- tutti i processi contribuiscono al calcolo della forza delle particelle.
- La simulazione viene fatta su un numero preciso di particelle se tale valore viene specificato dall'utente tramite riga di comando. Altrimenti viene fatta su un numero predefinito di particelle, indicato all'interno del programma.
- Ciascun processo, una volta a conoscenza del numero di particelle e processi, esegue il calcolo per la distribuzione del carico di lavoro. Cioè calcola sia la propria porzione di particelle sia quella degli altri processi e dopodiché inizializza le sue particelle. In seguito procede con la simulazione, la quale prevede 10 iterazioni, dove in ciascuna ogni processo:
    1. tramite invocazioni della funzione collettiva non bloccante MPI_Ibcast invia solo le coordinate(valori : x, y, z) delle sue particelle a tutti i processi e riceve le altre da essi. Vengono scambiate solo le coordinate poiché ciascun processo per il calcolo della velocità (valori vx, vy, vz) e della forza di ciascuna sua particella ha bisogno solo delle coordinate di tutte le altre.
    2. Durante la fase di comunicazione inizia il calcolo della forza relativo alla sua porzione di particelle. Ovvero fa interagire ciascuna sua particella con tutte le altre al suo interno.
    3. Resta in attesa di ricevere una o più porzioni tramite la funzione MPI_Waitsome. Non appena riceve una o più porzioni riprende il calcolo della forza, utilizzando le particelle appena ricevute, facendole interagire con le proprie. Ripete il passo 3 fin quando ci sono ancora richieste di comunicazione da completare.
    4. Terminate le fasi di ricezione e calcolo della forza, i processi SLAVE prima di aggiornare i loro valori inviano le velocità calcolate delle proprie particelle al MASTER, tramite la funzione MPI_Gatherv. Stessa cosa per il MASTER, ovvero prima di aggiornare i valori delle sue particelle attende la ricezione dei risultati delle velocità da parte degli SLAVE. Solo il MASTER ha bisogno di tutti i valori delle particelle poiché esso deve collezionarli tutti per poi scriverli su un file. Subito dopo, il processo può aggiornare i valori(coordinate e velocità) della propria porzione di particelle. Tali risultati sono necessari per il prossimo passo della simulazione.

Dettagli implementativi

Rappresentazione delle particelle

Le particelle nel programma sequenziale vengono rappresentate tramite un'unica struttura:

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

Mentre nella versione parallela tramite due *struct*:

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

La prima contiene le informazioni riguardanti le coordinate nello spazio e l'altra invece i valori relativi alla velocità. Dunque quando il programma viene eseguito con più processi: per le particelle vengono allocati due array, uno per le coordinate e l'altro per le velocità(entrambi con dimensione pari al numero di particelle dato in input); vengono definiti i tipi di dati derivati MPI corrispondenti e viene fatto il commit di questi ultimi per poter utilizzare le strutture definite nelle funzioni di comunicazione MPI.

```
    // DEFINE BODY TYPES
    MPI_Datatype BODY_POS, BODY_VEL;
    MPI_Datatype strcut_types[1];
    int blockcounts[1];
    MPI_Aint offsets[1], lb, extent;
    offsets[0] = 0;
    strcut_types[0] = MPI_FLOAT;
    blockcounts[0] = 3;

    MPI_Type_create_struct(1, blockcounts, offsets, strcut_types, &BODY_POS);
    MPI_Type_create_struct(1, blockcounts, offsets, strcut_types, &BODY_VEL);
    MPI_Type_commit(&BODY_POS);
    MPI_Type_commit(&BODY_VEL);
    // WORKLOADS DISTRIBUTION CALCULATION
    BodyPosition body_pos[nBodies];
    BodyVelocity body_vel[nBodies];
```

Questa suddivisione è stata fatta, come già accennato, perché il processo per calcolare la forza e la velocità delle proprie particelle ha bisogno di conoscere solo le coordinate di tutte le altre. L'unico processo che invece ha bisogno di sapere oltre alle coordinate anche gli altri valori è il MASTER, poiché esso deve collezionare tutti i risultati per poi stamparli su un file. Grazie a queste osservazioni è stato possibile ridurre considerevolmente l'overhead di comunicazione allo stretto necessario.

Distribuzione del carico di lavoro

```
    // WORKLOADS DISTRIBUTION CALCULATION
    int portion = nBodies / n_workers;
    int rest = nBodies % n_workers;
```

```
  int portions_sizes[n_workers], portions_starts[n_workers];
  calculatePortions(portions_sizes, portions_starts, n_workers, portion,
rest);
```

Indichiamo con *n_workers* il numero di processi che contribuiscono al calcolo della forza delle particelle. Ogni processo contribuisce alla computazione e il processo MASTER, in più rispetto agli altri colleziona i risultati delle varie iterazioni e li scrive su un file. Il numero di particelle viene distribuito tra gli n processi andando a dividere la taglia dell'input per il numero di n_workers. Calcolata la porzione che dovrà essere assegnata a ciascun processo e l'eventuale resto, vengono allocati gli array *procs_portions_sizes[ ]* e *procs_portions_starts[ ]*. Questi array servono per tener traccia delle porzioni di ciascun processo per la fase di invio e ricezione delle altre particelle(questo aspetto verrà approfondito in più avanti, quando verrà trattata la fase di invio e ricezione tramite le chiamate collettive *MPI_Ibcast*). Dopodiché viene fatto il calcolo delle porzioni e degli indici di inizio corrispondenti da assegnare a ciascun processo, come mostrato di seguito:

```
void calculatePortions(int procs_portions_sizes[], int
procs_portions_starts[], int n_workers, int portion, int rest)
{
    for (int i = 0; i < n_workers; i++)
    {
        procs_portions_sizes[i] = portion;
        if (rest > 0)
        {
            procs_portions_sizes[i]++;
            rest--;
        }
    }

    procs_portions_starts[0] = 0;
    for (int i = 1; i < n_workers; i++)
        procs_portions_starts[i] = procs_portions_starts[i - 1] +
procs_portions_sizes[i - 1];
}
```

Per semplicità se il numero di particelle non è divisibile per il numero di processi, il resto delle particelle viene distribuito a partire dal 1° processo(processo con rank 0) in poi, andando ad assegnare quindi una particella in più, delle restanti, a tali processi.

Fase di inizializzazione

```
  typedef struct
  {
      int rank, own_portion, start_own_portion, end_own_portion;
      int *procs_portions_sizes, *procs_portions_starts;
      float *Fx, *Fy, *Fz;

  } ProcVariables;
```

```
...

ProcVariables proc;
proc.rank = worker_rank;
proc.own_portion = portions_sizes[worker_rank];
proc.start_own_portion = portions_starts[worker_rank];
proc.end_own_portion = proc.start_own_portion + proc.own_portion;
float Fx[proc.own_portion], Fy[proc.own_portion], Fz[proc.own_portion];
proc.Fx = Fx;  proc.Fy = Fy;  proc.Fz = Fz;
proc.procs_portions_sizes = portions_sizes;
proc.procs_portions_starts = portions_starts;
// INIT OWN BODY PORTION
deterministicInitBodiesSplit(body_pos, body_vel, proc);

...
```

Vengono definite:

- la variabile proc, la quale è una struttura usata per raggruppare variabili che servono al processo in esecuzione come ad esempio la taglia e l'indice di inizio della sua porzione ecc.
- *Fx*, *Fy*, *Fz* servono per tener traccia dei valori intermedi relativi al calcolo della forza delle proprie particelle, durante i vari calcoli(tra le diverse invocazioni della funzione *bodyForceSplit*) con le altre particelle che vengono ricevute dagli altri processi. Ogni processo, una volta definito il proprio intervallo, inizializza le proprie particelle tramite un algoritmo deterministico, ovvero attraverso la funzione *determisticInitBodiesSplit*. A differenza del programma di partenza, che fa uso di un algoritmo che assegna valori alle particelle in maniera casuale, è stato utilizzato un algoritmo deterministico per poter valutare la correttezza del programma. Nel dettaglio la funzione utilizzata a tale scopo è la seguente:

```
void deterministicInitBodiesSplit(BodyPosition *body_pos, BodyVelocity
*body_vel, ProcVariables proc)
{
    for (int i = proc.start_own_portion; i < proc.end_own_portion; i++)
    {
        body_pos[i].x = i + 1;
        body_pos[i].y = i + 1;
        body_pos[i].z = i + 1;

        body_vel[i].vx = i + 1;
        body_vel[i].vy = i + 1;
        body_vel[i].vz = i + 1;
    }
}
```

Fase di simulazione

```
    // DEFINE REQUESTS FOR PORTIONS EXCHANGES
    MPI_Request bcast_reqs[n_workers];
    MPI_Request bcast_reqs2[n_workers];
    int reqs_ranks[n_workers];
    int req_done_indexes[n_workers];
    // START i-th PROCES WORK
    for (int iter = 1; iter <= nIters; iter++)
    {
        // PORTION EXCHANGE
        for (int i = 0; i < n_workers; i++)
            MPI_Ibcast(&body_pos[portions_starts[i]], portions_sizes[i],
BODY_POS, i, MPI_COMM_WORLD, &bcast_reqs[i]);
        // RESET ACCUMULATORS
        for (int i = 0; i < proc.own_portion; i++) {  Fx[i] = 0.0f;  Fy[i]
= 0.0f;  Fz[i] = 0.0f;  }
        // WORK ON OWN PORTION AND INCOMING PORTIONS
        bodyForceSplit(body_pos, dt, proc, proc.start_own_portion,
proc.end_own_portion);
        workOnIncomingPortions(n_workers, bcast_reqs, req_done_indexes,
proc, body_pos, dt);
        // SLAVES SEND THEIR OWN VELOCITIES AND MASTER RECEIVES THEM
        if (iter > 1)
            MPI_Gatherv(&body_vel[proc.start_own_portion],
proc.own_portion, BODY_VEL, &body_vel[proc.start_own_portion],
portions_sizes, portions_starts, BODY_VEL, MASTER, MPI_COMM_WORLD);
        // MASTER PRINTS RESOULTS
        if (worker_rank == MASTER && iter > 1 && print_res == 1)
            printResults(body_pos, body_vel, nBodies, output_file);
            // printResults(body_pos, body_vel, nBodies, fh);
        // UPDATE BODIES VALUES
        integrateVelocitySplit(body_vel, dt, proc);
        integratePositionSplit(body_pos, body_vel, dt, proc);
        // GO TO NEXT ITERATION
    }
    // FREE DATA TYPE
    MPI_Type_free(&BODY_POS);
    MPI_Type_free(&BODY_VEL);
}
```

Qui ogni processo tramite invocazioni della funzione collettiva non bloccante MPI_Ibcast invia la propria porzione di particelle a tutti gli altri processi, e riceve da questi ultimi le altre porzioni. In questa fase si tiene traccia delle corrispondenti richieste e durante la comunicazione il processo inizializza gli array Fx, Fy, Fz settando tutti i valori a 0. Lo scopo di tali array è memorizzare i "valori intermedi" della forza delle proprie particelle durante i vari calcoli (tra le varie invocazioni della funzione bodyForceSplit) con le altre che vengono ricevute. Dunque Fx, Fy, Fz si potrebbero pensare come a una sorta di "accumulatori temporanei" corrispondenti a ciascuna particella della porzione del processo, per calcolare la loro forza ad ogni iterazione(es. Fx[0],Fy[0],Fz[0] contengono i valori della forza della sua prima particella). In seguito il

processo inizia il calcolo della forza relativo alla sua porzione di particelle facendo interagire ciascuna di esse con tutte le altre al suo interno invocando la funzione bodyForceSplit:

```
void bodyForceSplit(BodyPosition *body_pos, float dt, ProcVariables proc,
int start_new_portion, int end_new_portion)
{
    for (int proc_portion = proc.start_own_portion, own_accumulator = 0;
proc_portion < proc.end_own_portion; proc_portion++, own_accumulator++)
    {
        for (int new_portion = start_new_portion; new_portion <
end_new_portion; new_portion++)
        {
            float dx = body_pos[new_portion].x - body_pos[proc_portion].x;
            float dy = body_pos[new_portion].y - body_pos[proc_portion].y;
            float dz = body_pos[new_portion].z - body_pos[proc_portion].z;

            float distSqr = dx * dx + dy * dy + dz * dz + SOFTENING;
            float invDist = 1.0f / sqrt(distSqr);
            float invDist3 = invDist * invDist * invDist;

            proc.Fx[own_accumulator] += dx * invDist3;
            proc.Fy[own_accumulator] += dy * invDist3;
            proc.Fz[own_accumulator] += dz * invDist3;
        }
    }
}
```

La funzione *bodyForeceSplit()* prende in input essenzialmente l'array di particelle tramite il parametro *body_pos*, un valore costante *dt*, le informazioni relative al processo in esecuzione tramite il parametro *proc*, gli indici dell'intervallo della nuova porzione, da utilizzare per il calcolo, tramite in parametri *start* ed *end*. Il processo calcola la forza di ciascuna sua particella facendo "interagire" ognuna di esse con quelle ricevute(con quelle della nuova porzione ricevuta). Quindi inizialmente il processo invoca la bodyForceSplit passandole come parametri *start_new_portion* ed *end_new_portion* l'inizio e la fine della propria porzione di particelle per iniziare il loro calcolo della forza facendo interagire ciascuna sua particella con tutte le altre al suo interno. Fatto ciò, viene invocata la funzione workOnIncomingPortions():

```
void workOnIncomingPortions(int n_req, MPI_Request bcast_reqs[], int
req_done_indexes[], ProcVariables proc, BodyPosition body_pos[], const int
dt)
{
    int num_req_compltd, start_new_portion, end_new_portion;
    while (1)
    {
        MPI_Waitsome(n_req, bcast_reqs, &num_req_compltd, req_done_indexes,
MPI_STATUSES_IGNORE);
        if (num_req_compltd == MPI_UNDEFINED)
            break;
```

```
        for (int i = 0; i < num_req_compltd; i++)
        {
            start_new_portion =
proc.procs_portions_starts[req_done_indexes[i]];
            end_new_portion = start_new_portion +
proc.procs_portions_sizes[req_done_indexes[i]];
            bodyForceSplit(body_pos, dt, proc, start_new_portion,
end_new_portion);
        }
    }
}
```

Il processo resta in attesa di ricevere le diverse porzioni da parte degli altri processi. Quindi attende il completamento di una o più richieste di comunicazione, tramite la funzione MPI_Waitsome. Quest'ultima va a scrivere in un array di interi, *req_done_indexes*, gli indici delle richieste che sono state completate. Per ogni richiesta completata viene utilizzata la corrispondente porzione di particelle ricevuta per il "calcolo intermedio" della forza di ciascuna particella del processo, invocando la funzione bodyForceSplit. Questa fase viene ripetuta per tutte le porzioni ricevute e fin quando tutte le richieste non sono state completate, ovvero quando la MPI_Waitsome restituisce MPI_UNDEFINED nella variabile num_req_compltd, la quale indica il numero di richieste completate dopo l'invocazione della MPI_Waitsome. Terminata la fase di comunicazione il processo è pronto per aggiornare i valori delle proprie particelle, ma prima di aggiornali, i processi SLAVE inviano al MASTER i risultati delle velocità calcolate attraverso la funzione MPI_Gatherv. Il MASTER dopo aver ricevuto i risultati, sempre tramite l'invocazione della funzione MPI_Gatherv, li scrive su un file creato all'avvio del programma. Una volta fatto ciò il processo aggiorna i valori delle proprie particelle e passa all'iterazione successiva della simulazione. Le funzioni utilizzate per gli aggiornamenti dei valori sono quelle riportate qui sotto:

```
void integratePositionSplit(BodyPosition *body_pos, BodyVelocity *body_vel,
float dt, ProcVariables proc)
{
    for (int i = proc.start_own_portion; i < proc.end_own_portion; i++)
    {
        body_pos[i].x += body_vel[i].vx * dt;
        body_pos[i].y += body_vel[i].vy * dt;
        body_pos[i].z += body_vel[i].vz * dt;
    }
}

void integrateVelocitySplit(BodyVelocity *body_vel, float dt, ProcVariables
proc)
{
    for (int i = proc.start_own_portion, own_accumulator = 0; i <
proc.end_own_portion; i++, own_accumulator++)
    {
        body_vel[i].vx += dt * proc.Fx[own_accumulator];
        body_vel[i].vy += dt * proc.Fy[own_accumulator];
        body_vel[i].vz += dt * proc.Fz[own_accumulator];
```

```
    }
}
```

Istruzioni per l'esecuzione

compilazione :

```
mpicc nbody.c -lm -o {nome eseguibile}
es: mpicc nbody.c -lm -o nbody_split.out
```

esecuzione **senza output su file**:

```
mpirun -np {numero processi} {nome eseguibile} {numero particelle}
es: mpirun -np 4 nbody_split.out 40000
```

esecuzione **con output scritto su file** per testare la correttezza del programma:

```
mpirun -np {numero processi} {nome eseguibile} {numero particelle} -t
es: mpirun -np 4 nbody_split.out 40000 -t
```

Correttezza del programma

La correttezza del programma viene verificata attraverso n esecuzioni dello stesso. Ogni esecuzione viene fatta utilizzando il comando

```
mpirun -np {numero processi} {nome eseguibile} {numero particelle} -t
```

Dove ad ogni esecuzione viene dato in input al programma la stessa quantità di particelle e il corrispondente numero di p processi da utilizzare assegnato in maniera progressiva. Quando viene specificato nel comando il parametro -t, all'inizio dell'esecuzione il programma genera un file nel quale poi vengono scritti i risultati dalla simulazione. Il nome con il quale viene creato il file dipende dal numero di processi con cui si esegue il programma. Per semplicità i file vengono nominati parallel_{numero processi} es:

```
es: mpirun -np 1 nbody_split.out 40000 -t
genera il file parallel_1
...
es: mpirun -np 4 nbody_split.out 40000 -t
genera il file parallel_4
...
```

Per verificare che l'esecuzione fatta con uno o più processi generi lo stesso output, al termine di tutte le esecuzioni, il file generato dal programma sequenziale (parallel_1.) viene confrontato con tutti gli altri. Per automatizzare la verifica della correttezza è stato realizzato uno scritp bash che verrà riportato qui sotto. Comandi per eseguire lo script:

```
bash correctness_verifier.sh {path file eseguibile} {numero particelle}
{numero processi} {parametri opzionali}
es: bash correctness_verifier.sh ./nbody_split_correctness.out 4000 32
```

Per quanto riguarda i parametri opzionali, se viene specificato il parametro -l seguito da un numero intero questo sta ad indicare il numero di ripetizioni del test. Es: `bash correctness_verifier.sh ./nbody_split_correctness.out 4000 32 -l 4` esegue il programma sul numero di particelle(4000) e numero massimo di processi(da 1 a 32) per 4 volte.

```
  #!/bin/bash

  rm parallel_* # elimina tutti gli eventuali file creati da un test
precedente

  progname=$1  #primo parametro path del programma da eseguire
  nBodies=$2   #secondo parametro numero di bodies
  n_proc=$3    #terzo parametro numero massimo di n processi
  loop=$4      #quarto parametro indica se il programma deve essere
eseguito in loop
  n_iters=1    #quinto parametro, se il quarto viene specificato, indica il
n° esecuzioni del programma
  proc=1       #variabile per incrementare il n° di processi
  i=1          #variabile per iterazione i-esima

  if [ "$#" -eq 5 ] # viene specificato il n° di esecuzioni del programma
  then
    n_iters=$5
  else
    n_iters=1
  fi
  while [[ ((i -le n_iters)) ]]
  do
  proc=1 # reset del numero di processi
  while [[ ((proc -le n_proc)) ]]
    do
      mpirun --allow-run-as-root -np $proc --oversubscribe $progname
$nBodies -t
      wait # attesa che il programma termini la sua esecuzione
      # confronto dei risultati con quelli del programma sequenziale
tramite il comando diff
      # diff -q solo se ci stanno differenze le restituisce altrimenti
restituisce stringa vuota
      DIFF=$(diff -q parallel_$proc parallel_1 )
```

```
      if [ "$DIFF" != "" ]
      then
        printf "parallel program np %d \!\!\! ERRORE \!\!\! \n\n" $proc
        exit # termina lo script
      else
        printf "parallel program np %d --> OK\n\n%s" $proc $DIFF
      fi
      wait # aspetta diff
      ((proc++)) # aumenta n° processi
    done
    # se viene passato solo -t e senza specificare il n° di iterazioni va
in loop infinito
    # dunque non incrementiamo la variabile i (arrestare manualmente)
    if [ "$#" -eq 3 ] || [ "$#" -eq 5 ]
    then
      ((i++)) # iterazione successiva
    fi
  done
```

Discussione dei risultati

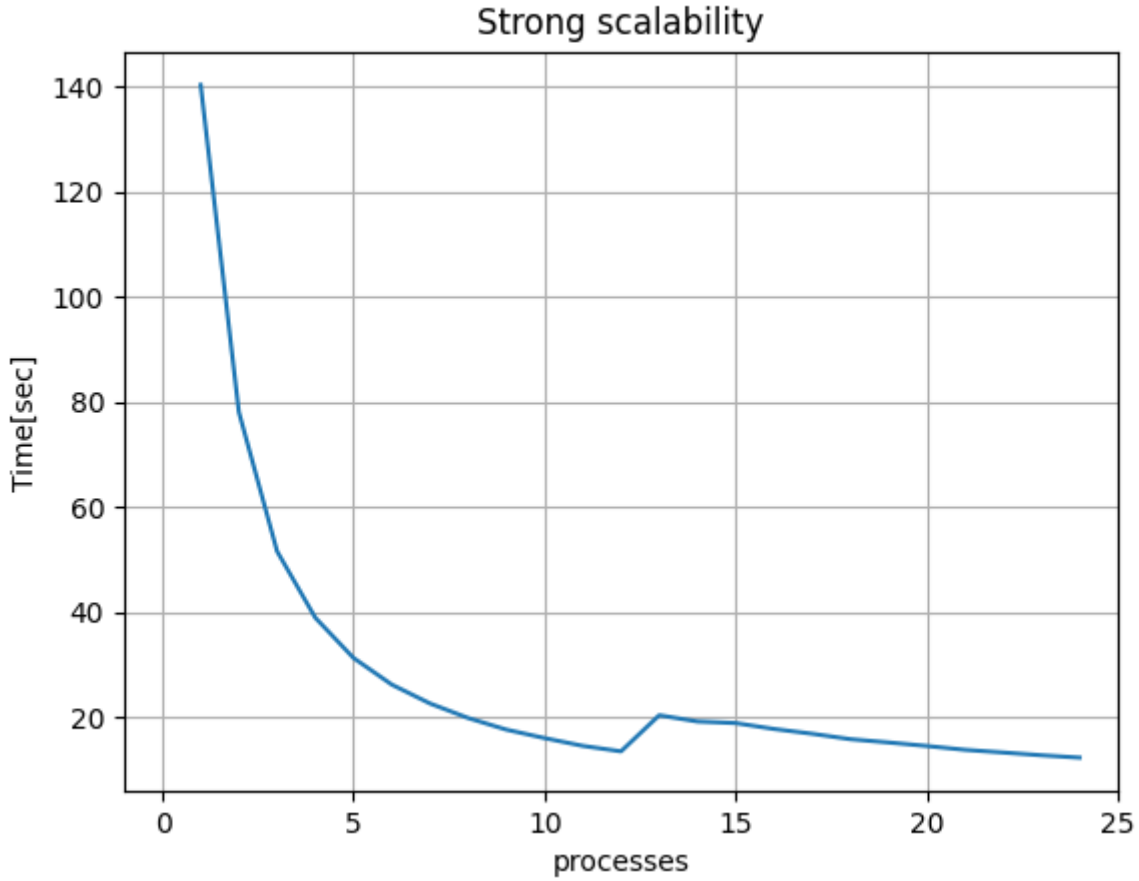
Le prestazioni del programma sono state valute tenendo in considerazione sia la scalabilità forte che quella debole. I benchmark sono stati svolti su un cluster Google Cloud con 6 istanze di macchine virtuali (e2-standard-4), aventi ciascuna 2 core(4vCPU). Sia per la scalabilità forte che per quella debole sono stati eseguiti 24 esperimenti.

Strong scalability

La strong scalability viene valutata, mantenendo la taglia dell'input del problema fissa e facendo aumentare progressivamente il numero di processori. Gli esperimenti sono stati condotti su un input fissato di 24000 particelle. Di seguito vengono mostrati i risultati ottenuti:

vCPU	Input	Tempo	speed-up
1	24000	140.244	1.000
2	24000	78.054	1.796
3	24000	51.582	2.710
4	24000	38.953	3.600
5	24000	31.304	4.480
6	24000	26.247	5.343
7	24000	22.652	6.191
8	24000	19.881	7.054
9	24000	17.672	7.935

vCPU	Input	Tempo	speed-up
10	24000	16.063	8.730
11	24000	14.584	9.616
12	24000	13.544	10.354
13	24000	20.409	6.871
14	24000	19.220	7.296
15	24000	18.920	7.412
16	24000	17.802	7.877
17	24000	16.892	8.302
18	24000	15.859	8.843
19	24000	15.234	9.205
20	24000	14.566	9.628
21	24000	13.835	10.136
22	24000	13.325	10.524
23	24000	12.799	10.957
24	24000	12.348	11.357



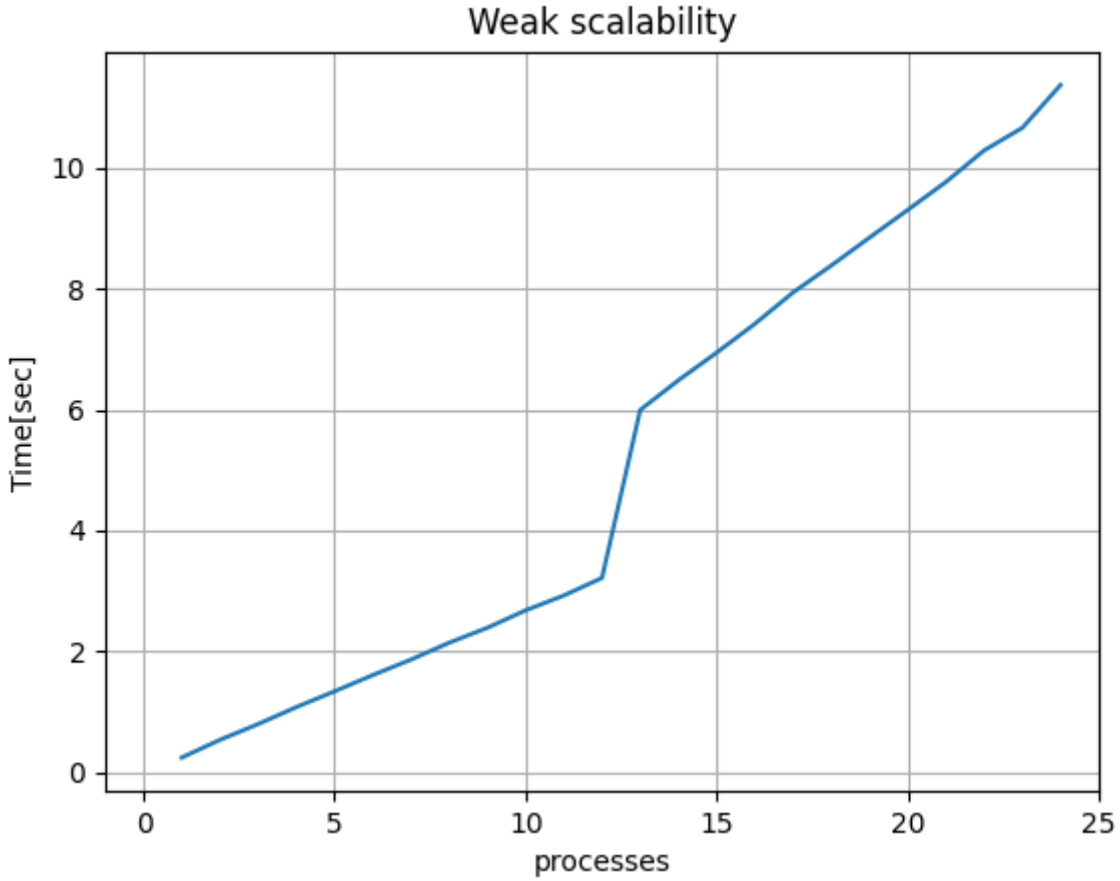
Con la scalabilità forte si vede quanto lo speed-up ottenuto dal programma è approssimativamente proporzionale al numero di processori utilizzati, cioè di quanto è stato ridotto il tempo di esecuzione del programma sequenziale utilizzando n processori e quanto esso sia approssimativamente vicino al fattore n. Dagli esiti si può riscontrare che all'aumentare del numero di processi il tempo di esecuzione è abbastanza vicino al fattore n utilizzato e il tempo di esecuzione diminuisce fino ad un certo punto. Da lì in poi la velocità di esecuzione del programma aumenta sempre meno. Questo perché con l'aumentare del numero di processi, il lavoro svolto da ciascuno di essi diminuisce, ma di contro aumenta l'overhead di comunicazione. Dunque con l'aumentare del numero di processi ognuno di essi spende sempre meno tempo per la computazione e sempre più tempo per la comunicazione che è la parte più onerosa dell'esecuzione e che quindi richiede più tempo.

Weak scalability

La weak scalability invece, viene valutata facendo variare la taglia dell'input al variare del numero di processori facendo in modo che ogni processo abbia sempre lo stesso workload(ovvero la stessa quantità di lavoro). Gli esperimenti sono stati fatti in modo che ogni processo lavorasse su 1000 particelle. Dunque partendo con un processo con 1000 particelle in input, fino ad arrivare a 24 processi con 24000 particelle come input. Di seguito vengono mostrati i risultati ottenuti:

vCPU	Input	Tempo	Efficienza
1	1000	0.238	1.0000
2	2000	0.532	0.2236
3	3000	0.793	0.1000
4	4000	1.074	0.0554
5	5000	1.335	0.0356
6	6000	1.604	0.0247
7	7000	1.860	0.0182
8	8000	2.140	0.0139
9	9000	2.387	0.0110
10	10000	2.676	0.0088
11	11000	2.921	0.0074
12	12000	3.213	0.0061
13	13000	5.994	0.0030
14	14000	6.485	0.0026
15	15000	6.942	0.0022
16	16000	7.419	0.0020
17	17000	7.938	0.0017

vCPU	Input	Tempo	Efficienza
18	18000	8.384	0.0015
19	19000	8.846	0.0014
20	20000	9.304	0.0012
21	21000	9.767	0.0011
22	22000	10.290	0.0010
23	23000	10.663	0.0009
24	24000	11.373	0.0008



La scalabilità debole valuta la capacità di mantenere il tempo di risoluzione fisso risolvendo problemi più grandi su risorse di calcolo più grandi. Quindi il suo obiettivo è quello di misurare la capacità di mantenere un tempo di risoluzione "costante" al variare del numero di processori e taglia dell'input. Possiamo notare dai risultati della weak scalability che le prestazioni degradando costantemente di quasi la metà ogni volta che taglia e numero di processi crescono progressivamente. Il problema in questo caso è che man mano si scala l'overed di comunicazione per ciascun processo aumenta. Questo si evince anche dai valori di efficienza, che nello scalare sono sempre più distanti dal valore 1 il quale sta ad indicare "il valore ottimo" che si mira a raggiungere, ovvero più il valore di efficienza è vicino ad 1 e più i processori non sono in stato di hidle e i costi di comunicazione e sincronizzazione sono più bassi. Più nel dettaglio, il lavoro di computazione per ogni processo rimane lo stesso, ma a questo va aggiunto il tempo di comunicazione, il quale aumenta man mano che si scala, poiché ogni processo deve comunicare con più processi.

Conclusioni

Come si è potuto riscontrare dalla strong scalability i risultati ottenuti sono soddisfacenti. In termini di tempo di esecuzione sono stati riscontrati miglioramenti. Mentre gli esiti dalla weak scalability non sono stati così tanto positivi visto che vi è un costante incremento, sempre in termini di tempo di esecuzione, di quasi il doppio man mano che si scala. In conclusione i risultati ottenuti si possono ritenere accettabili vista la complessità dell'algoritmo e in particolare il suo considerevole overhead di comunicazione, il quale è causato dal fatto che ogni processo ha bisogno delle porzioni di tutti quanti gli altri per svolgere il proprio lavoro.