PCPC - Nbody

Daniele De Vinco 0522500801

Soluzione proposta

La soluzione proposta segue lo schema definito nell'algoritmo sequenziale di nbody[1].
Lo sforzo è stato incentrato principalmente nel ridurre l'overhead delle comunicazioni fra i processi, come indicato nel problema[2].
Il processo master inizializza con un algoritmo deterministico le particelle, e le inoltra con un messaggio broadcast a tutti gli altri processi.
Viene poi effettuato il calcolo per suddividere il lavoro in maniera equa tra tutti i processi, incluso il master.
Una volta completata la computazione, ogni processo deve inviare la propria porzione a tutti gli altri processi, in modo che siano in grado di effettuare il prossimo step della simulazione.
Vengono quindi stampati i tempi di esecuzione, e se richiesti, i valori delle particelle. E' possibile anche scrivere su un file i risultati per effettuare un confronto.
Infine le risorse utilizzate vengono liberate.

Il file viene eseguito con il seguente schema di comandi:

```
mpicc <nomefile.c> -o <nomeeseguibile> -lm
mpirun -np <num processi> <nomeeseguibile> < num particelle>
<num iterazioni> <scrittura su file> <scrittura su stdout>
```

L'opzione *-lm* viene utilizzata per linkare la libreria math.h.
Un esempio pratico di come viene lanciato il programma in locale:

```
mpicc nbodypcpc.c -o esame -lm
mpirun -np 8 esame 30000 5 risultato_8_proc.txt 1
```

Struttura del progetto

Dopo l'inizializzazione delle variabili, viene chiamata una funzione per effettuare il parsing degli argomenti della linea di comando.

```
Get_args(argc, argv, &nIters, &nBodies, &file_name, &print_result,
  world_rank);
```

L'utente può scegliere il numero di particelle e il numero di iterazioni, e settare un flag, *print_result*, per la visualizzazione dei risultati e il partizionamento della particelle. E' possibile passare anche il nome di un file da linea di comando, in modo da poter scrivere i risultati in binario su un file. Se non viene passato nessun parametro da linea di comando, il numero di particelle è fissato a 10 e il numero di iterazioni a 5.
Viene quindi creata la struttura di tipo MPI per facilitare lo scambio di messaggi tra i processi [3].

```
blockcounts[0] = 6;
oldtypes[0] = MPI_FLOAT;
offset[0] = 0;

MPI_Type_create_struct(1, blockcounts, offset, oldtypes, &body_struct)
MPI_Type_commit(&body_struct);
```

Il processo master, associato in questo progetto sempra al processo con rank 0, inizializza tutti i valori delle particelle con un seed fissato, in modo da poter rieffettuare la computazione con gli stessi valori, dopodichè li invia con un messaggio broadcast [4] a tutti gli altri processi, se ne esistono altri.

```
  void randomizeBodies(float *data, int n) {
    srand(1);
    for (int i = 0; i < n; i++) {
      data[i] = 2.0f * (rand() / (float)RAND_MAX) - 1.0f;
    }
  }
```

```
if (world_rank == 0)
  randomizeBodies(buf, 6*nBodies);

if (world_size > 1)
  MPI_Bcast(p, nBodies, body_struct, 0, MPI_COMM_WORLD);
```

Ogni processo calcola in maniera identica la partizione degli elementi in modo equo, distribuendo il carico di lavoro. Si utilizzano tre array, che memorizzano la grandezza di ciascuna partizione, il punto iniziale da dove iniziare a effettuare la computazione e il punto dove terminare. Questi array sono utilizzati anche per poter effettuare lo scambio di messaggi tra i vari processi.

```
local_count = (int*) malloc(sizeof(int)*world_size);
starting_point = (int*) malloc(sizeof(int)*world_size);
ending_point = (int*) malloc(sizeof(int)*world_size);
```

Viene quindi effettuata la computazione parallela, in cui vengono passati alla funzione i valori di inizio e fine della propria sezione. A differenza di quella sequenziale, abbiamo quindi un primo ciclo che effettua la computazione solamente sulla propria sezione, mentre il secondo ciclo rimane invariato.

```
bodyForceParallel(p, dt, nBodies, starting_point[world_rank], ending_p

void bodyForceParallel(Body *p, float dt, int nBodies, int starting_po
  for(int i = starting_point; i < ending_point; i++){
    /*
    calcoli
    /*
  }
}
```

La prima implementazione provata era molto semplice, ovvero una volta terminata l'esecuzione della *BodyForceParallel*, si lanciava un messaggio broadcast verso tutti gli altri processi, trasmettendo la propria sezione.

```
if (world_size > 1)
  MPI_Bcast(p+starting_point[world_rank], local_count[world_rank], body
MPI_Barrier(MPI_COMM_WORLD);
```

Seppur funzionante, questo metodo si scontrava con la grandezza variabile delle sezioni, fornendo alcuni errori nell'accesso ai vari buff, soprattutto quando la divisione fra il numero di particelle e il numero di processori non aveva resto uguale a 0. Inoltre la necessità di dover sincronizzare l'operazione inficiava sui tempi di esecuzione.
La soluzione più efficiente utilizza una funzione definita nella libreria di *OPENMPI* [6] e seguendo l'esempio in [7]:

```
if (world_size > 1)
  MPI_Allgatherv(p+starting_point[world_rank], local_count[world_rank]
```

Allgatherv consente di avere alla fine dell'esecuzione gi stessi dati nel recvbuf per tutti i processi, consentendo anche di inviare buffer di lunghezze diverse.
Infine vengono stampati i tempi di esecuzione e liberate tutte le risorse.

Analisi delle performance

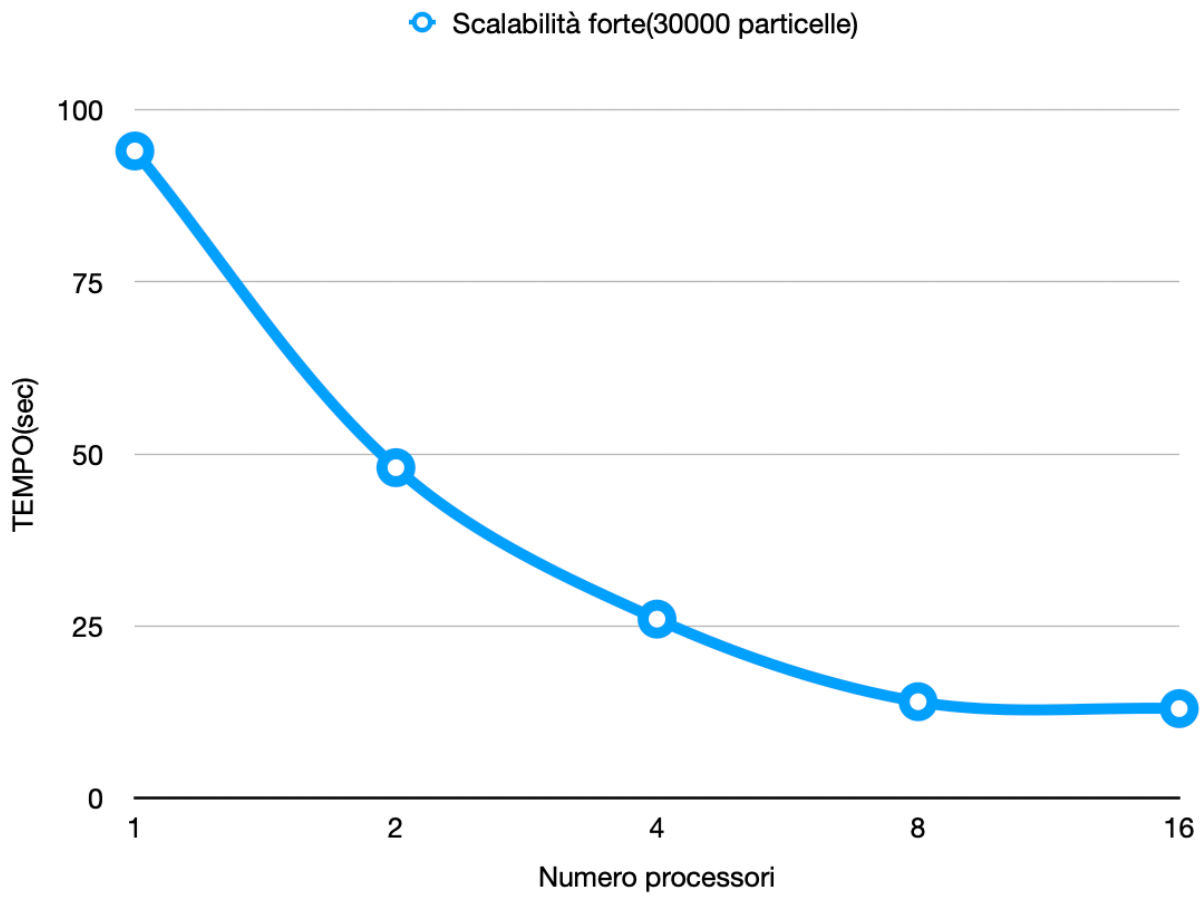
Definendo la scalabilità forte:

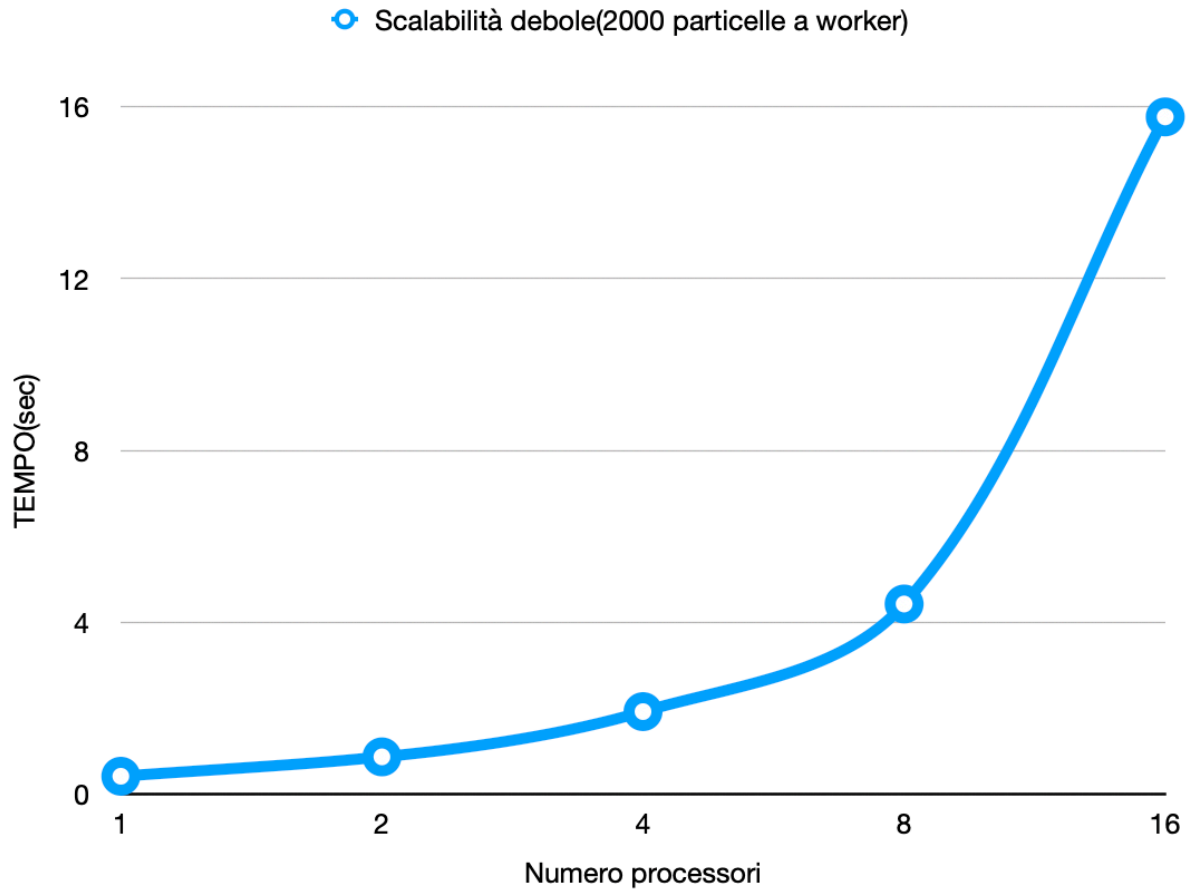
```
Tempo impiegato per risolvere un problema di grandezza fissata dato un
 numero variabile di processi
```

Definendo la scalabilità debole:

```
Un problema aumenta la propria grandezza in modo proporzionale al nume
ro di processi
```

Possiamo quindi analizzare i risultati ottenuti sulla base di queste definizioni. Innanzitutto le performance sono simili, sia in locale che sul cluster AWS. Per quanto riguarda la computazione in locale, questi sono i risultati in termini di scalabilità forte e debole:



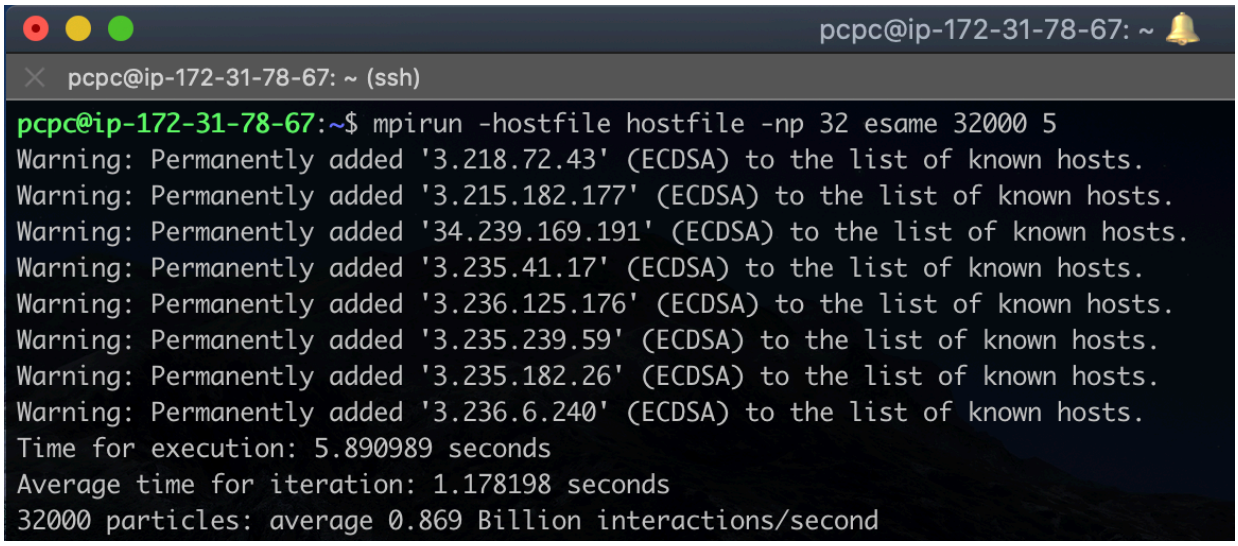


Per il mio progetto, mi sono state assegnate le macchine di AWS m4.xlarge. Ognuna di queste macchine ha 4 vCPU(2 core fisici) e 16 GB di RAM[8][9]. Essendo il limite per l'account student 32 vCPU, è stato creato un cluster con 8 istanze di queste macchine. Quindi sull'istanza che fa da master il programma è stato eseguito in questo modo:

```
    mpirun --hostfile hostfile -np 32 esame 64000 5
```

con l'host file così creato[10]:

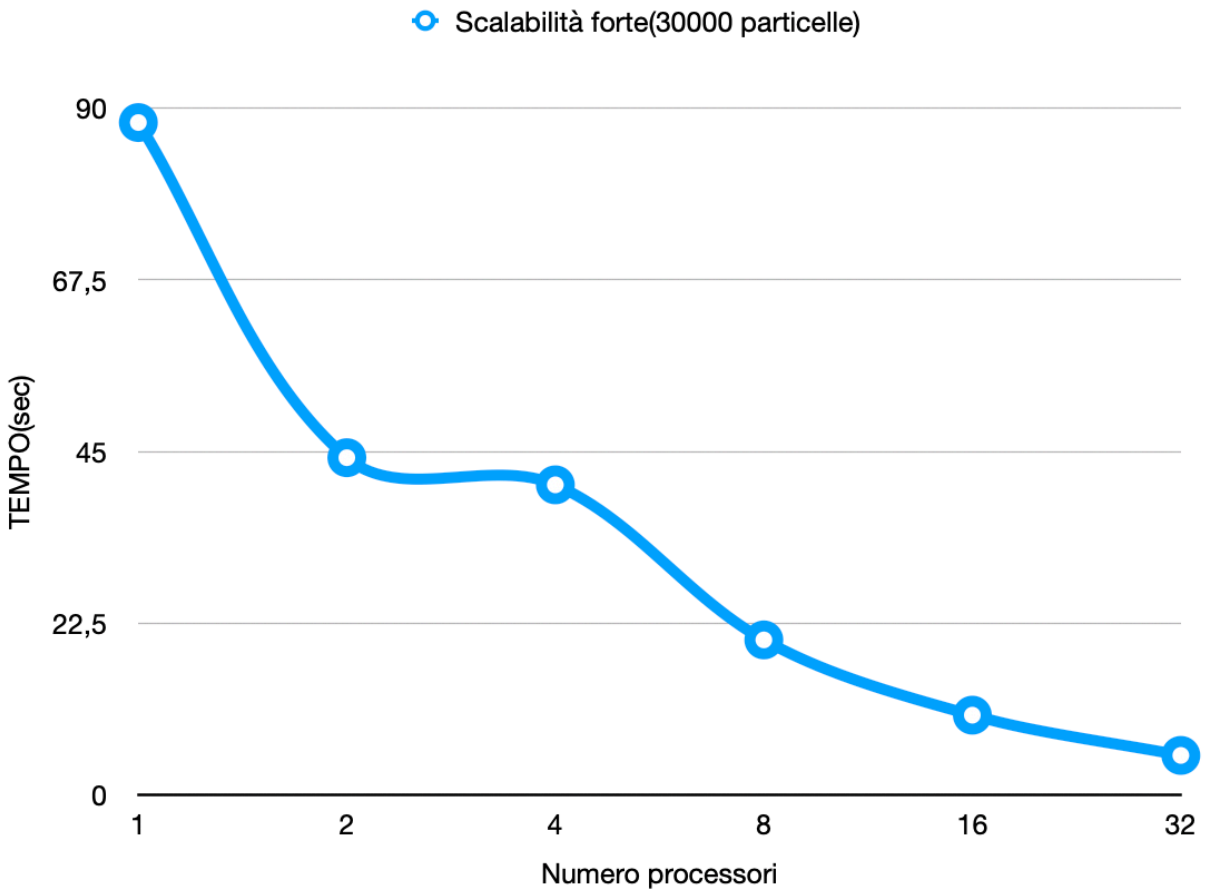
```
    3.236.125.176 slots=4 max-slots=4
    3.215.182.177 slots=4 max-slots=4
    3.218.72.43 slots=4 max-slots=4
    3.235.239.59 slots=4 max-slots=4
    3.235.41.17 slots=4 max-slots=4
    34.239.169.191 slots=4 max-slots=4
    3.236.6.240 slots=4 max-slots=4
    3.235.182.26 slots=4 max-slots=4
```



Questi i risultati in termini della scalabilità forte e debole:

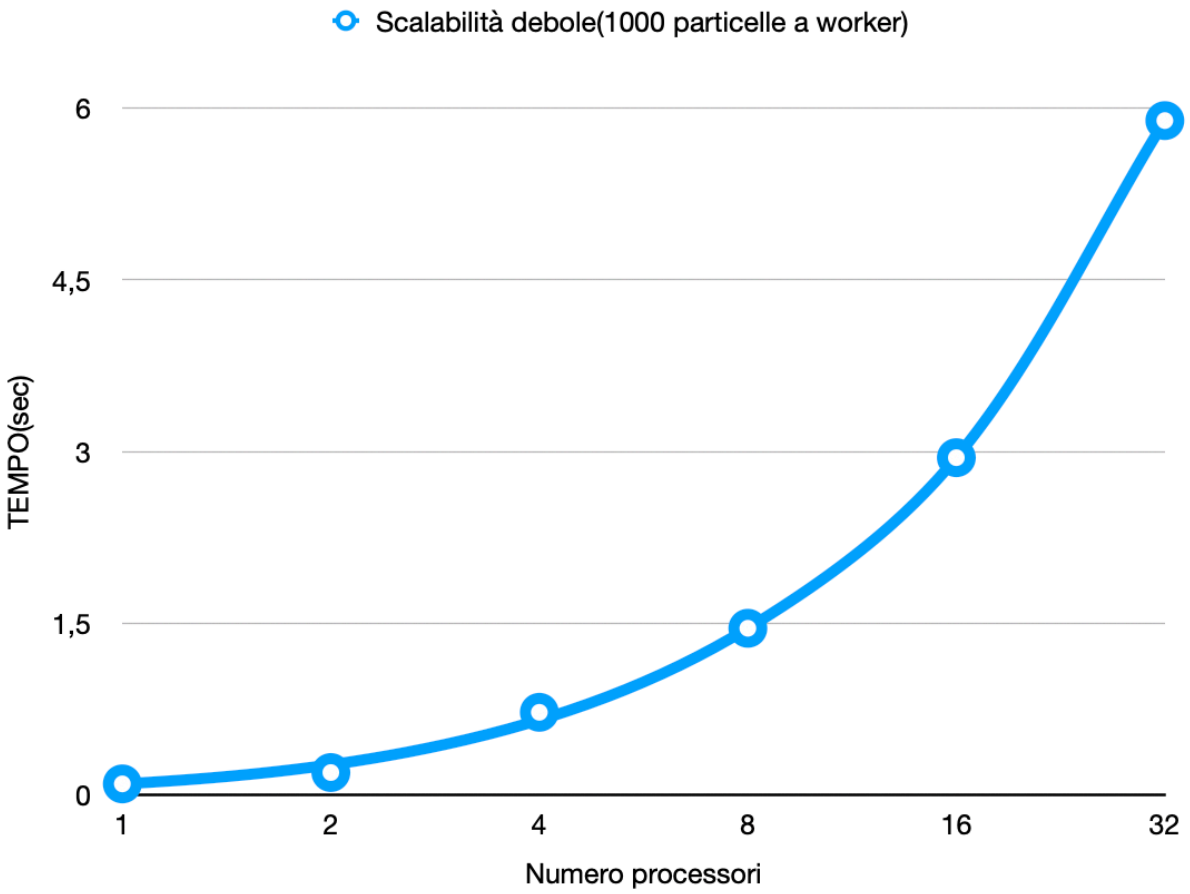
Strong scalability

	Tempo
1	88,094
2	44,188
4	40,631
8	20,315
16	10,457
32	5,183



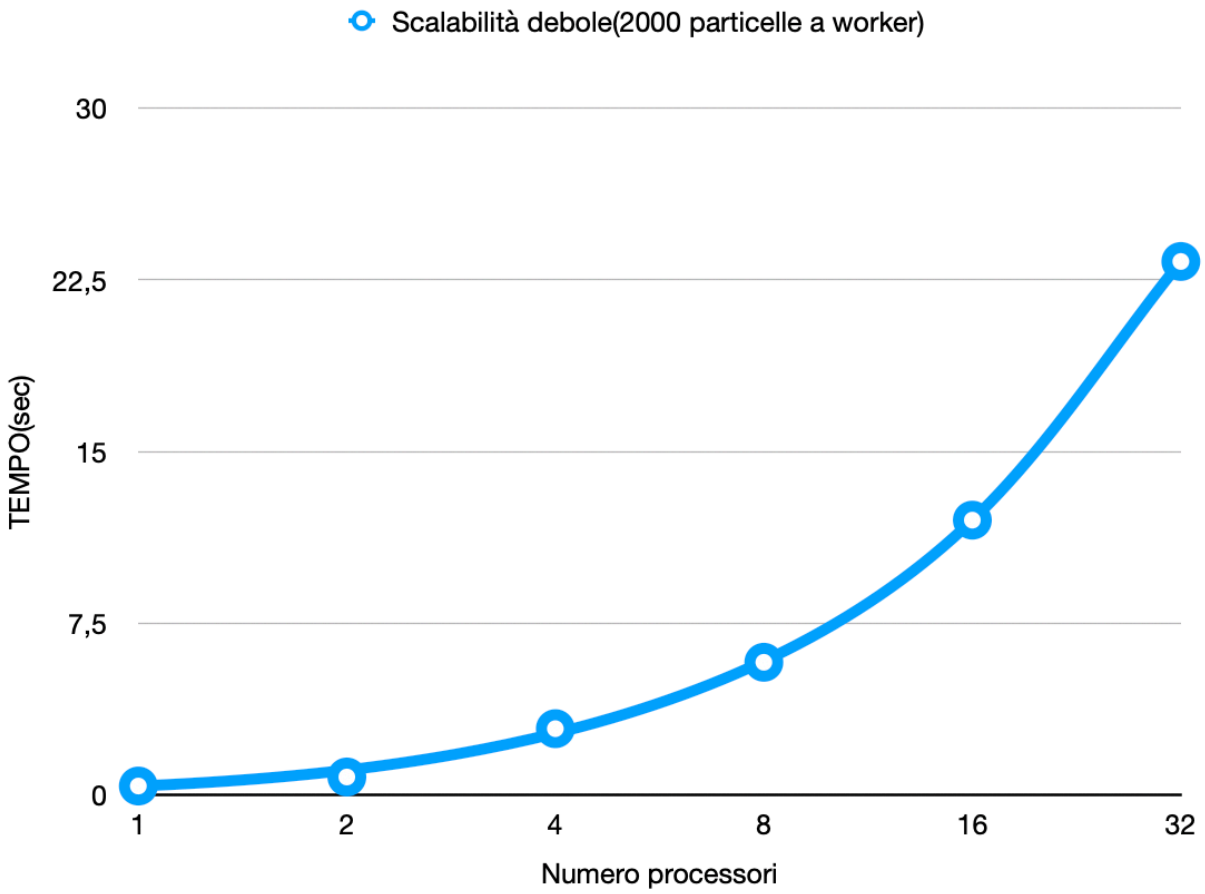
Weak scalability - n=1000

	Tempo
1	0,097
2	0,196
4	0,722
8	1,456
16	2,946
32	5,890



Weak scalability - n=2000

	Tempo
1	0,391
2	0,785
4	2,895
8	5,795
16	12,002
32	23,297



Conclusioni

Ho ottenuto delle buone performance sia in locale che sul cluster. La comunicazione tra diverse macchine funziona in maniera ottimale grazie ad *MPI* sul cluster.
Esiste un leggero degrado delle prestazioni sul cluster quando è stata provata la scalabilità forte con 4 processi. Le prestazioni sono state misurate con diverse configurazioni, avendo in media il medesimo risultato. Secondo la documentazione di *OPENMPI*, il comando mpirun si occupa di distribuire il carico di lavoro tra i vari host in modo da non sovraccaricare una macchina. Inoltre specificando nell'hostfile il numero di *slot* o *maxslot* è possibile forzare questo bilanciamento, impedendo alle macchine di utilizzare l'hyperthreading. In sostanza non vi è uno speedup deciso fra 2 e 4 processi, ma in generale i risultati mostrano che le prestazioni migliorano con il crescere del numero di processi.

Riferimenti

Gli esperimenti in locale sono stati effettuati su una macchina con queste caratteristiche:

```
Nome modello:               MacBook Pro
Identificatore modello:     MacBookPro16,1
Nome processore:            8-Core Intel Core i9
Velocità processore:        2,4 GHz
Numero di processori:       1
Numero totale di Core:      8
Cache L2 (per Core):        256 KB
Cache L3:                   16 MB
Tecnologia Hyper-Threading: Abilitato
Memoria:                    16 GB
```

Versione MPI utilizzata:

```
(Open MPI) 4.0.2
```

Link alle risorse utilizzate

1. nbody
2. PCPC 2020
3. Derived datatypes
4. Collective communications
5. Abort
6. AllgatherV - 1
7. AllgatherV - 2
8. AWS Instances
9. Optimize Instances
10. Hostfile