Corso di Programmazione Concorrente e Distribuita

Relazione Assignment #3

Simulazione del movimento di N corpi in un piano bidimensionale sfruttando un approccio basato su attori.

Alessandro Magnani, Simone Montanari, Andrea Matteucci

Analisi del problema

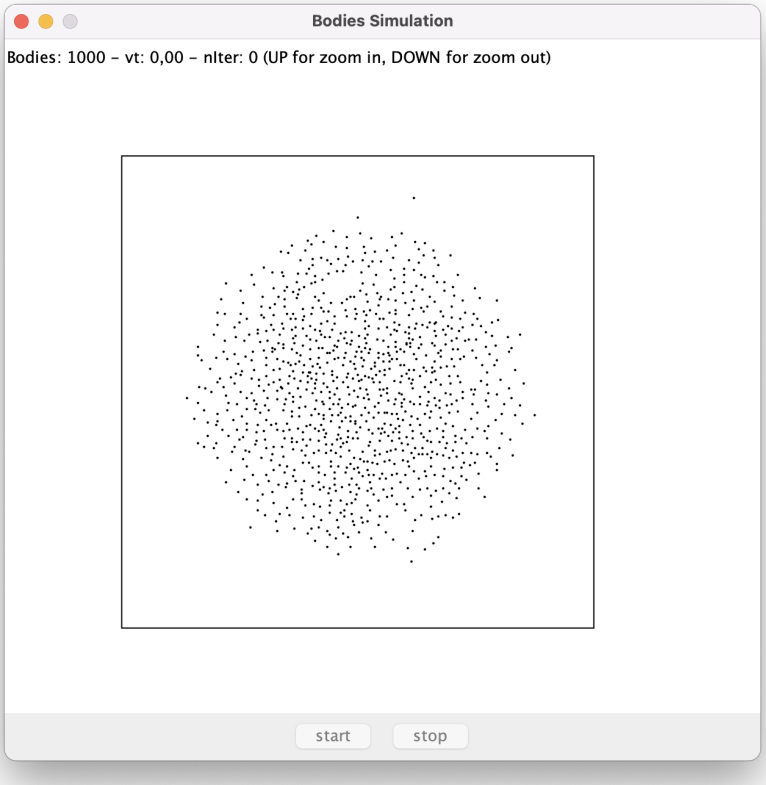
Il problema riguarda la gestione concorrente di un numero N di palline che si muovono in uno spazio bidimensionale, soggette a due tipi di forze:

- *Forza repulsiva*, esercitata da un corpo sugli altri;
- *Forza di attrito*, contraria al moto del singolo corpo, ed opposta alla sua velocità.

Nelle versioni relative ai precedenti Assignment, l'algoritmo per poter gestire il movimento delle palline in modo concorrente faceva uso di Thread (Assignment #1) e di Task (Assignment #2).

In questo caso, il principale problema da affrontare è quello di implementare una soluzione a task adottando un approccio basato su attori.

È di seguito riportato uno screenshot del sistema in esecuzione.



Architettura proposta e strategia risolutiva

All'interno del Main relativo alla parte 1 è possibile avviare due versioni diverse del programma proposto:

- Una composta dalla GUI ed i relativi pulsanti di avvio e di terminazione
- Una senza GUI, per la sola misurazione delle performance, ovvero del tempo di esecuzione

Versione senza GUI

In questa versione, similmente a quanto fatto nei due precedenti assignment, si è scelto di implementare un'architettura Master-Worker, in cui un attore Master (*MasterActor*) delega degli attori Workers (*WorkerActor*) e assegna loro quali palline dovranno gestire mediante lo scambio di messaggi definiti nel protocollo (*WorkerProtocol*).

Nello specifico, è stato utilizzato l'algoritmo realizzato nell'Assignment #1 (*createWorkerActorAndAssignBalls()*) per poter distribuire equamente il numero di palline tra i vari attori Workers, che ovviamente gestiranno un sottoinsieme limitato di palline (alla creazione di ogni attore Worker viene specificato l'indice iniziale e il numero di palline da gestire).

Ogni attore Worker computa le 3 operazioni sulle palline (*ComputeTotalForceAndUpdateVelocity()*, *updatePosition()* e *checkBoundaryCollision()*) come comportamento alla ricezione di specifici messaggi.

L'attore Master, invece, si occupa di :

- delegare le computazioni sulle palline ai vari Workers;
- tenere traccia del numero di iterazioni correnti e del tempo virtuale
- tenere traccia del numero di Workers che hanno eseguito le varie operazioni

I messaggi contenuti nel protocollo (*WorkerProtocol*) e, di conseguenza, quelli usati dagli attori sono i seguenti:

- ***BootMsg* (*Main/Controller → MasterActor*)**
  Segnala l'inizio della simulazione, viene inviato all'attore Master che alla ricezione:
  - genera le palline;
  - crea gli attori e assegna loro un numero distribuito di palline;
  - imposta le variabili di gestione (*actualSteps*, *vt* e *t0*);

    - invia il messaggio relativo alla prima operazione (*UpdateVelocities*) a tutti gli attori Workers;

- ***StopMsg* (*Controller → MasterActor | MasterActor → WorkerActor*)**
  Segnala l'interruzione della simulazione all'attore Master che, una volta ricevuto, invia ad ogni Worker un altro messaggio di stop, stampa a video il tempo di esecuzione per poi interrompere il suo comportamento restituendo *Behaviors.stopped()*.
  (ogni Workers alla ricezione di un messaggio di stop interrompe il suo comportamento tornando il medesimo comportamento);

- ***UpdateVelocities (MasterActor → WorkerActor)***
  Messaggio inviato dall'attore Master a tutti gli attori Workers per far eseguire la prima operazione sulle palline, specificando:
    - la lista di palline (*bodies*)
    - il time step (*dt*)
    - il riferimento a se stesso (Master) in modo tale che i Workers possano mandare un messaggio per segnalare la fine della computazione richiesta

  Ogni attore Worker alla ricezione di questo messaggio esegue la prima operazione sulle palline assegnategli alla creazione per poi segnalarlo al Master inviando un messaggio *EndUpdateVelocities.*

- ***EndUpdateVelocities (WorkerActor → MasterActor)***
  Messaggio inviato dagli attori Workers per segnalare la fine dell'esecuzione della prima operazione.
  L'attore Master alla ricezione del messaggio aumenta il contatore relativo ai Workers 'arrivati' (*workersArrived*) e, se tutti i Workers hanno finito la prima computazione, il Master segnala ad ogni Worker di eseguire le altre due operazioni sulle palline mediante il messaggio *UpdatePositionAndCheckBoundaryCollision*.

- ***UpdatePositionAndCheckBoundaryCollision (MasterActor → WorkerActor)***
  Messaggio inviato dall'attore Master a tutti gli attori Workers per far eseguire le altre due operazioni sulle palline, specificando:
    - la lista di palline (*bodies*)
    - il time step (*dt*)
    - il bordo (*bounds*)
    - il riferimento a se stesso (Master) in modo tale che i worker possano mandare un messaggio per segnalare la fine della computazione richiesta

Ogni attore Worker alla ricezione di questo messaggio esegue la seconda e la terza operazione sulle palline assegnatogli alla creazione per poi segnalarlo al Master inviando un messaggio *EndUpdatePositionAndCheckBoundaryCollision*

- ***EndUpdatePositionAndCheckBoundaryCollision (WorkerActor → MasterActor)***
  Messaggio inviato dagli attori Workers per segnalare la fine dell'esecuzione della seconda e terza operazione.
  L'attore Master alla ricezione del messaggio aumenta il contatore relativo ai Workers 'arrivati' (*workersArrived*) e, se tutti i Workers hanno finito la relativa computazione, il Master esegue le seguenti operazioni:
  - aumenta il numero di iterazioni correnti (*actualSteps*);
  - aumenta il tempo virtuale (*vt*) di un time step (*dt*);
  - se si è nella versione con GUI esegue la *display();*
  - se la simulazione non è finita (*actualSteps* < *nSteps*) il Master procede all'invio dei messaggi relativi alla prima computazione.
    Altrimenti stampa a video il tempo di esecuzione interrompendo il suo comportamento (*Behaviors.stopped()*).

<u>Versione con GUI</u>

Nella costruzione del codice si è seguito il pattern **MVC**, in modo da avere una struttura che ci permettesse un collegamento efficiente tra le diverse parti: Model, View e Controller.

La simulazione è gestita dalla classe *Controller* che, una volta notificato l'inizio della simulazione, provvederà alla creazione dell'attore Master (*MasterActor*) specificando che si tratta della versione con GUI (variabile booleana *isGuiVersion*).

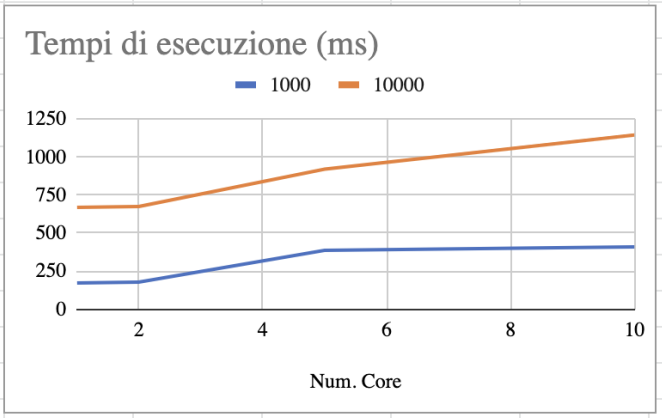
Il funzionamento è analogo a quello della versione senza GUI, l'unica differenza è legata alla presenza di quest'ultima. Difatti il *MasterActor* ad ogni iterazione, si occupa di stampare a video la situazione aggiornata delle palline.
Nello specifico, come comportamento alla ricezione di messaggi del tipo *EndUpdatePositionAndCheckBoundaryCollision.*

Performance

100 corpi

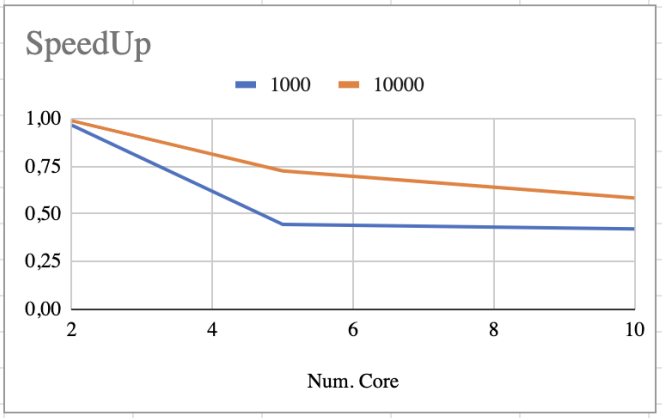
100 corpi
Tempi di esecuzione (ms)

	Iterazioni	
Num. Core	1000	10000
1	172	668
2	178	675
5	387	920
10	409	1144



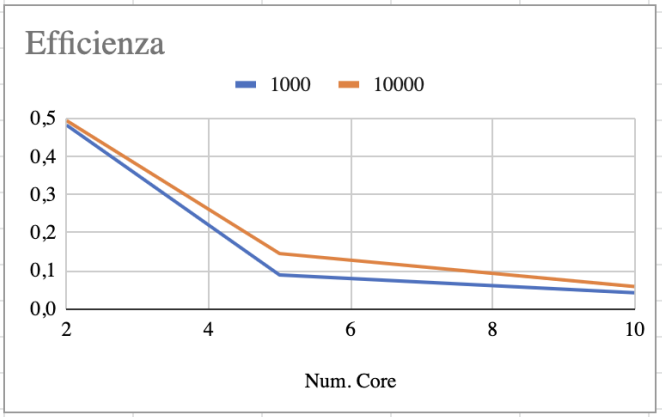
100 corpi
SpeedUp

	Iterazioni	
Num. Core	1000	10000
2	0,9662921	0,9896296
5	0,4444444	0,7260869
10	0,4205378	0,5839160



100 corpi
Efficienza

	Iterazioni	
Num. Core	1000	10000
2	0,4831460	0,4948148
5	0,0888888	0,1452173
10	0,0420537	0,0583916

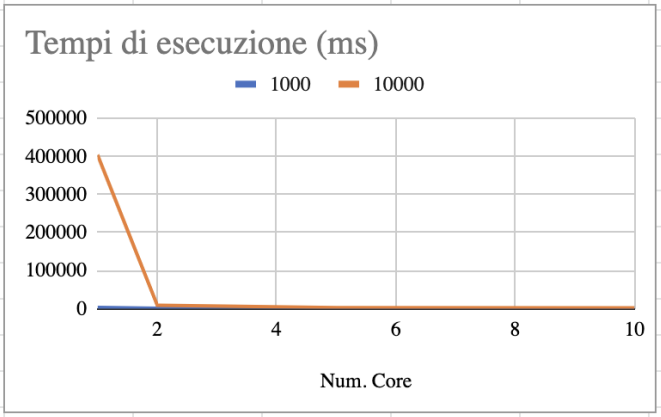


Abbiamo eseguito i test riguardanti le performance con gli stessi valori dell'Assignment #1 e #2, in modo da avere un migliore confronto tra le due realizzazioni.
Partendo con la situazione iniziale di 1000 corpi, notiamo che l'impatto del parallelismo non è visibile. Infatti abbiamo i tempi migliori con un minor numero di core. Di conseguenza, SpeedUp ed efficienza diminuiscono nel tempo.

1000 corpi

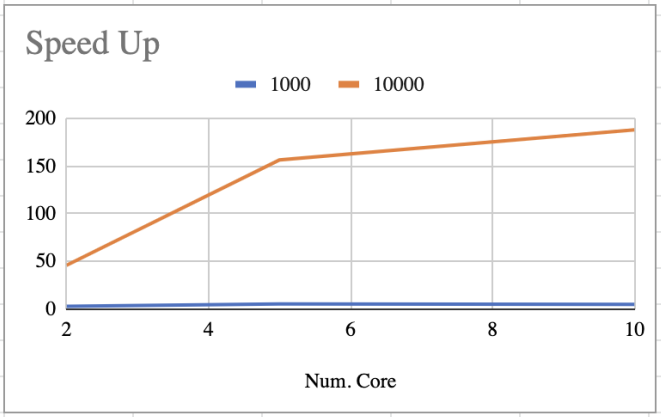
1000 corpi
Tempi di esecuzione (ms)

	Iterazioni	
Num. Core	1000	10000
1	3145	404172
2	1170	8833
5	629	2584
10	665	2148



1000 corpi
SpeedUp

	Iterazioni	
Num. Core	1000	10000
2	2,6880341	45,757047
5	5	156,41331
10	4,7293233	188,16201



1000 corpi
Efficienza

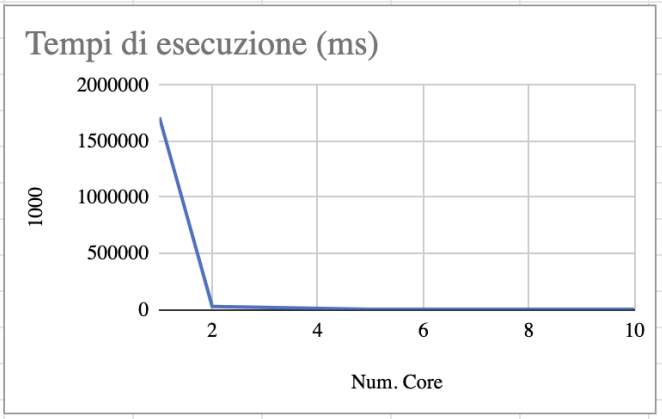
	Iterazioni	
Num. Core	1000	10000
2	1,3440170	22,878523
5	1	31,282662
10	0,4729323	18,816201

Con 1000 corpi, invece, notiamo tempi di esecuzione drasticamente minori dal momento in cui andiamo ad aumentare i core. Non siamo però riusciti a spiegarci il valore così alto raggiunto dal tempo di esecuzione con un unico core.

5000 corpi

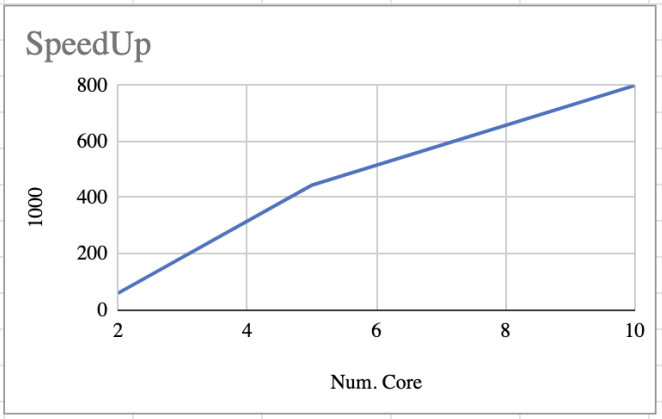
5000 corpi
Tempi di esecuzione (ms)

	Iterazioni	
Num. Core	1000	10000
1	1710457	
2	28605	
5	3857	695776
10	2142	18517



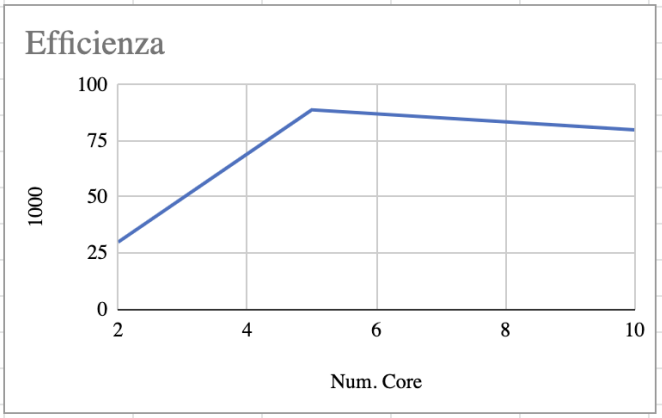
5000 corpi
SpeedUp

	Speedup	
Num. Core	1000	10000
2	59,795735	
5	443,46823	
10	798,53267	



5000 corpi
Efficienza

	Efficienza	
Num. Core	1000	10000
2	29,897867	
5	88,693647	
10	79,853267	



In questa ultima misurazione non siamo riuscirci a misurare i tempi di esecuzione con 1 e 2 core, in quanto risultavano essere molto lunghi.

Se però guardiamo le esecuzioni con 1000 corpi, notiamo un miglioramento progressivo dello SpeedUp, ed un'efficienza che tende regolarizzarsi superati i 5 core.

Comportamento del sistema

È di seguito riportato il comportamento del sistema con GUI:

1. Inizialmente nel main del programma vengono istanziati il componente grafico (*SimulationView*) ed il controller (*Controller*), specificando numero di iterazioni, numero di palline e numero di Workers

2. Una volta premuto il pulsante START, viene notificato **il controller** (metodo *notifyStarted()*) che conseguentemente **creerà l'attore Master** (*MasterActor*) specificando:
   - il controller
   - se si tratta della versione con GUI (*isGuiVersion*)
   - il numero di iterazioni (*nSteps*)
   - il numero di palline (*nBalls*)
   - il numero di Workers (*nWorkers*)

   Successivamente il controller manda un messaggio di boot (*BootMsg*) al Master per segnalare l'inizio della simulazione

3. L'attore **Master** alla ricezione del messaggio di boot:
   - genera le palline
   - crea i Workers e assegna loro un numero distribuito di palline
   - imposta a 0 il numero di iterazioni corrente e il tempo virtuale
   - segnala a tutti i Workers di procedere con la prima operazione sul loro sottoinsieme di palline (messaggio *UpdateVelocities*)

4. Ogni **Worker** alla ricezione del messaggio *UpdateVelocities*:
   - esegue la prima operazione sul proprio sottoinsieme di palline
   - segnala la fine della computazione inviando al master un messaggio *EndUpdateVelocities*

5. L'attore **Master** alla ricezione del messaggio *EndUpdateVelocities*:
   - incrementa il contatore (*workersArrived*, inizialmente posto a 0) relativo ai Workers 'arrivati'
   - se il numero di Workers arrivati equivale al numero totale di Workers il Master azzera il contatore e segnala a tutti i Workers di procedere con le altre due operazioni (messaggio *UpdatePositionAndCheckBoundaryCollision*)

6. Ogni **Worker** alla ricezione del messaggio *UpdatePositionAndCheckBoundaryCollision*:
    - esegue le due operazioni sul proprio sottoinsieme di palline
    - segnala la fine della computazione inviando al master un messaggio *EndUpdatePositionAndCheckBoundaryCollision*

7. L'attore **Master** alla ricezione del messaggio *EndUpdatePositionAndCheckBoundaryCollision*:
    - incrementa il contatore (*workersArrived*, inizialmente posto a 0) relativo ai Workers 'arrivati'
    - se il numero di Workers arrivati equivale al numero totale di Workers il Master azzera il contatore, incrementa il numero di iterazioni correnti, aumenta il tempo virtuale e stampa la situazione aggiornata delle palline (*display()*).
    - se il numero di iterazioni corrente è minore rispetto al numero di quelle totali (indicando quindi che le iterazioni non sono ancora terminate), il Master inizia un'altra iterazione mandando ad ogni Worker un messaggio *UpdateVelocities*.
      In caso contrario stampa a video il tempo di esecuzione totale e interrompe il suo comportamento restituendo *Behaviors.stopped()*

8. Se durante l'esecuzione della simulazione viene premuto il pulsante STOP, viene notificato il controller (metodo *notifyStopped()*) che conseguentemente provvederà a segnalare l'interruzione all'attore Master con un messaggio *StopMsg*

   L'attore Master alla ricezione di un messaggio di stop:
    - segnala l'interruzione a tutti i workers mandando loro uno *StopMsg*
    - stampa a video il tempo di esecuzione
    - interrompe il suo comportamento tornando *Behaviors.stopped()*

   Ogni Worker alla ricezione di un messaggio di stop, interrompe il suo comportamento restituendo *Behaviors.stopped()*.

Comportamento del sistema senza GUI

Il funzionamento della versione senza GUI è analogo, l'unica differenza è che la variabile *isGuiVersion* è impostata a false e dunque l'attore Master alla ricezione del messaggio *EndUpdatePositionAndCheckBoundaryCollision* non eseguirà la *display()*.

Rete di Petri