PCD Assignment 01

Francesco Foschini, Davide Alpi

10 aprile 2022

Sommario

L'obiettivo di questo assignment è realizzare una versione concorrente del sistema "Simulator" Il programma simula il movimento di N corpi su un piano bidimensionale, soggetti a due tipi di forze:

- una forza repulsiva, per cui ogni corpo bi esercita su ogni altro corpo bj una forza in modulo pari a: Fij = krep * mi / dij2 , dove mi è la massa del corpo bi, krep è una costante data, dij è la distanza fra i due corpi. La direzione della forza è data dal versore (bj - bi) – ovvero respingente per il corpo bj.
- una forza di attrito, per cui su ogni corpo bi che si muove ad una velocità vi è esercitata una forza FRi = - kfri* vi che si oppone al moto, quindi in direzione opposta alla sua velocità, dove kfri è una costante data.

L'algoritmo che definisce il comportamento del simulatore - contenuta nella classe Simulator - in pseudocodice è il seguente:

```
1  vt = 0;      /* virtual time */
2  dt  = 0.01; /* time increment at each iteration */
3
4  loop {
5      For each body b[i]:
6        compute total force exerted by other bodies b[j]
             and
7        friction
8        compute the instant acceleration, given the
             total force
9        and mass
10       update body velocity, given the acceleration and
             the
11       virtual time elapsed dt
12     Update bodies positions, given the velocity and
           virtual
13     time elapsed dt
14     Check boundary collisions;
15     vt = vt + dt;
16     Display current stage;
17 }
```

Indice

1 Analisi del problema

Data la versione concorrente del sistema "Simulator", l'obiettivo è parallelizzare ove possibile l'esecuzione del programma ottenendo un buon valore di speedup.

1.1 Proprietà dei corpi

Le proprietà di ciascun corpo sono: - posizione (P) - velocità (V)

- Per calcolare la velocità aggiornata di un corpo occorre effettuare una READ sulle P di tutti i corpi e una WRITE sulla V del corpo.
- Per calcolare la posizione aggiornata di un corpo occorre effettuare una READ sulla V del corpo e una WRITE sulla P del corpo.
- Il controllo della collisione con il bordo richiede una READ sulla P del corpo ed eventualmente anche una WRITE su P e V.

1.2 Vincoli alla parallelizzazione

L'aggiornamento delle V di tutti i corpi può essere svolto in parallelo da n worker. Ciascun worker dovrà aggiornare la V di nbody/n body. L'operazione di READ delle P degli altri corpi (necessaria per il calcolo della V) può essere svolta concorrentemente in quanto operazione di READ. Non serve quindi regolare l'accesso in lettura alla P dei corpi in questa fase.

L'aggiornamento delle posizioni (READ su V e WRITE su P) creerebbe problemi di corse critiche alla parte di calcolo delle velocità (che deve fare READ su P di tutti i corpi). Si devono quindi mettere in campo dei meccanismi che permettano ai worker di procedere all'aggiornamento delle posizioni dei corpi solamente dopo che tutte le velocità sono state aggiornate.

La parte di controllo di collisione con i bordi di un corpo deve avvenire dopo che la sua velocità e posizione sono state aggiornate.

2 Strategia risolutiva e architettura proposta

La classe architetturale che abbiamo preso come riferimento è quella del result parallelism. Il nostro controller (master), divide la lista dei corpi in n parti uguali e li assegna ad n worker, che eseguono, per ciascun corpo:

- il calcolo della velocità aggiornata
- l'aggiornamento della posizione (e il controllo di possibili collisioni con i bordi)

Le posizioni devono essere aggiornate solo dopo che tutte le velocità sono state aggiornate. Per realizzare ciò i worker si coordinano tramite un monitor barriera:

Listing 1: Barrier

```
1  public class BarrierImpl implements Barrier{
2      private final int nWorkers;
3      private int nHits;
4
5      public BarrierImpl(int nWorkers) {
6          this.nWorkers = nWorkers;
7          this.nHits = 0;
8      }
9
10     @Override
11     public synchronized void hitAndWaitAll() throws
           InterruptedException {
12         nHits++;
13         if(nHits == nWorkers) {
14             notifyAll();
15         } else {
16             while (nHits < nWorkers) {
17                 wait();
18             }
19         }
20     }
21 }
```

A ciascun worker viene assegnato dal master un numero di corpi pari a nbody/nworker. Dividiamo a priori i body in parti uguali tra i worker perchè le operazioni sui diversi corpi sappiamo avere complessità computazionale equivalente.

Il master viene notificato del completamento del'esecuzione dei worker tramite un monitor latch:

Listing 2: Latch

```
1  public class LatchImpl implements Latch {
2      private int nWorker;
3
```

```
4       public LatchImpl(int nWorker) {
5           this.nWorker = nWorker;
6       }
7
8       @Override
9       public synchronized void notifyCompletion() {
10          this.nWorker--;
11          if(this.nWorker == 0){
12              notifyAll();
13          }
14      }
15
16      @Override
17      public synchronized void waitCompletion() throws
            InterruptedException {
18          while (this.nWorker > 0){
19              wait();
20          }
21      }
22  }
```

Ad ogni iterazione il master crea nuovi worker, oltre che una nuova barriera e un nuovo latch.

2.1 Versione con GUI

Nella versione con GUI, alla fine di ogni iterazione, il master si occupa anche di invocare l'aggiornamento dell'interfaccia grafica.

Abbiamo implementato i pulsanti di start e stop con semantica pause-continue. L'Event Dispatcher Thread di Java, alla pressione del tasto di pausa/continue, setta un flag di stop. Al termine di ogni iterazione, prima di invocare l'aggiornamento dell'interfaccia grafica, il master controlla il flag di stop e se è settato a true si mette in wait. Abbiamo implementato il flag come semplice monitor:

Listing 3: Flag

```
1   public class Flag {
2       boolean flag = false;
3
4       public synchronized void set(boolean v){
5           this.flag = v;
6           notifyAll();
7       }
8
9       public synchronized void waitWhile(boolean v) {
10          while (this.flag == v){
11              try {
12                  wait();
13              } catch (InterruptedException ex){}
14          }
```

```
15          }
16  }
```

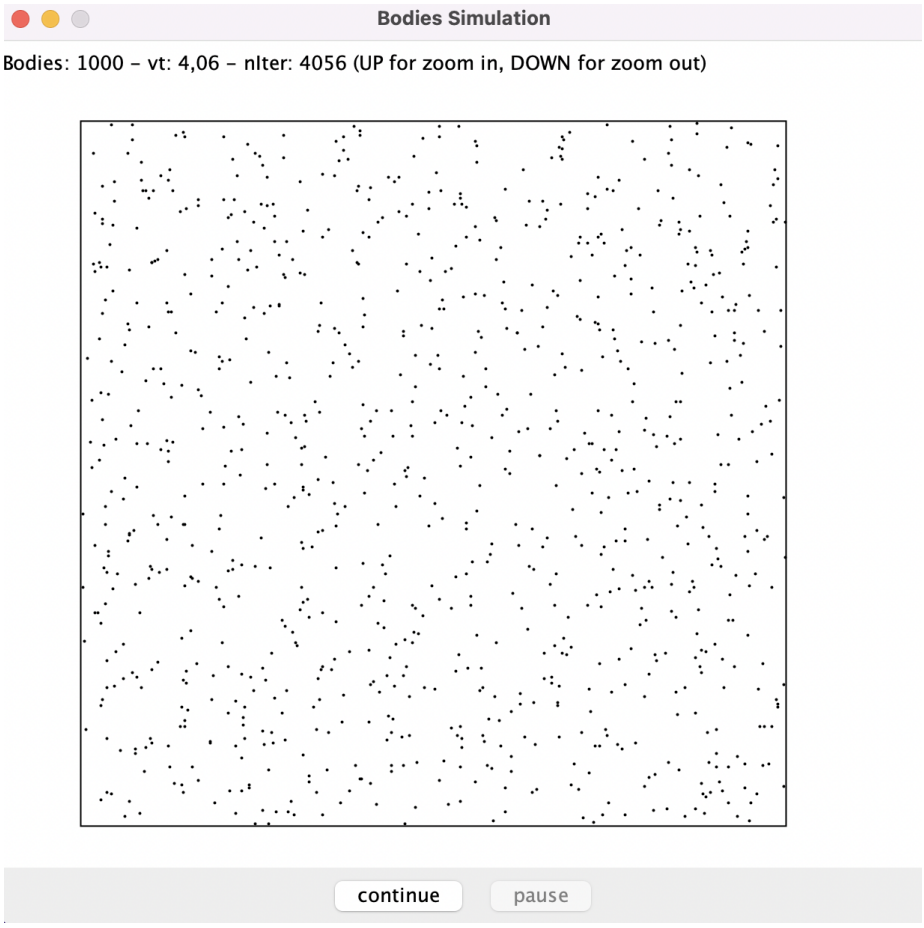

Figura 1: Screenshoot simulazione con GUI con mille corpi.

3 Comportamento del sistema

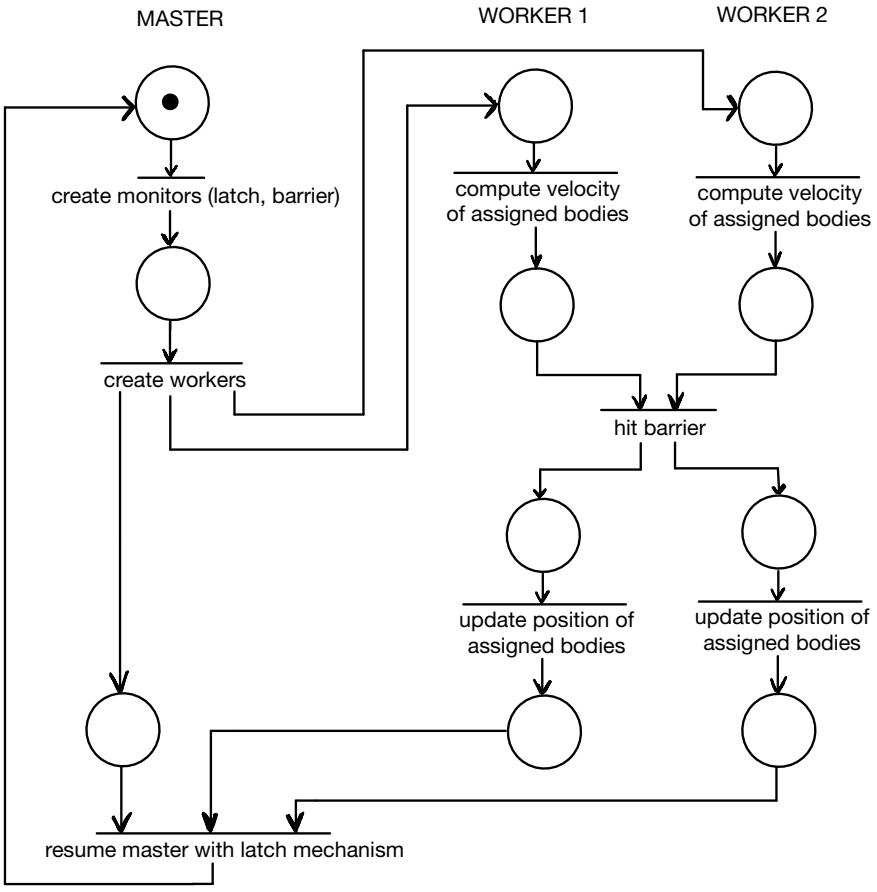

Figura 2: Rete di Petri che modella il comportamento del sistema senza GUI e con n worker = 2

4 Performance

I test sono stati eseguiti senza la parte di GUI, con un macbook air 2020 (M1).
I risultati evidenziano un buon speedup ottenuto dalla versione concorrente (più di 1 worker).

Numero test	Corpi	Iterazioni	Thread	Tempo di esecuzione (ms)	Speedup
1	100	1000	1	84	1
2	100	1000	5	158	0.53
3	100	1000	9	295	0.28
4	100	1000	20	617	0.13
5	100	10000	1	676	1
6	100	10000	5	1308	0.51
7	100	10000	9	2360	0.28
8	100	10000	20	5444	0.12
9	100	50000	1	3069	1
10	100	50000	5	6199	0.49
11	100	50000	9	11462	0.267
12	100	50000	20	26732	0.11
13	1000	1000	1	2729	1
14	1000	1000	5	1065	2.56
15	1000	1000	9	1053	2.59
16	1000	1000	20	1244	2.19
17	1000	10000	1	25596	1
18	1000	10000	5	8778	2.91
19	1000	10000	9	9375	2.73
20	1000	10000	20	11413	2.24
21	1000	50000	1	130838	1
22	1000	50000	5	43497	3.00
23	1000	50000	9	44428	2.94
24	1000	50000	20	56032	2.335
25	5000	1000	1	70477	1
26	5000	1000	5	21715	3.245
27	5000	1000	9	17896	3.938
28	5000	1000	20	18165	3.879
29	5000	10000	1	1002825	1
30	5000	10000	5	415426	2.41
31	5000	10000	9	350071	2.864
32	5000	10000	20	355361	2.82
33	5000	50000	1	5430121	1
34	5000	50000	5	2254882	2.408
35	5000	50000	9	1680942	3.23
36	5000	50000	20	1649430	3.29

Il test delle performance ha evidenziato come in alcuni casi sia più performante la versione sequenziale rispetto a quella concorrente. Quando all'interno della simulazione i corpi sono pochi la versione sequenziale del sistema risulta più efficace perchè priva di meccanismi di sincronizzazione tra i thread.

5 Identificazione di proprietà di correttezza e verifica

5.1 Java Path Finder

Le proprietà di Safety e Liveness sono state verificate con Java Path Finder. I due listener utilizzati per la ricerca di violazioni a queste due proprietà sono:

- gov.nasa.jpf.listener.PreciseRaceDetector
- gov.nasa.jpf.listener.DeadlockAnalyzer

Comando per lanciare la ricerca:

```
1    java -jar ./JPF/jpf-core/build/RUNjpf.jar src/main/java/
         ass01/jpf/safety.jpf
```

5.2 TLA+

Il sistema è stato modellato in versione semplificata tramite TLA+, astraendo tutte le computazioni che ogni worker deve fare con il semplice settaggio di 2 flag (positions e velocities).

- 0: valore da aggiornare
- 1: valore aggiornato

I worker devono quindi settarli a 1 simulando le due fasi di aggiornamento delle velocità e aggiornamento delle posizioni. Il master si occupa al termine di ciascuna iterazione di risettarli a 0.

Inoltre, i processi worker non vengono ricreati ad ogni iterazione e controlliamo la corretta sincronizzazione con la seguente invariante:

```
1    WorkerIterationInSyncWithMaster == (\A n \in 1..
         NUMBER_OF_WORKERS: workerIteration[n] >= iteration)
```

Tramite la definizione di proprietà, verifichiamo inoltre che:

- il calcolo della posizione avvenga sempre dopo che tutte le velocità sono state calcolate
- ad ogni iterazione vengano effettivamente calcolate tutte le velocità e le posizioni
- ad ogni iterazione venga "distrutta" la barriera e "sbloccato" il latch
- che venga raggiunto e non superato il numero di iterazioni previsto

```
1   PositionUpdatedAfterVelocity == []( (\A n \in 1..
        NUMBER_OF_WORKERS: positions[n] = 1) => (\A n \in 1..
        NUMBER_OF_WORKERS: velocities[n] = 1) )
2   VelocityComputedEachStep ==  (iteration < STEPS) => <>(\A n
         \in 1..NUMBER_OF_WORKERS: velocities[n] = 1)
3   PositionComputedEachStep ==  (iteration < STEPS) => <>(\A n
         \in 1..NUMBER_OF_WORKERS: positions[n] = 1)
4   LatchTerminationEachStep == (iteration < STEPS) => <>(latch
         = 0)
5   BarrierTerminationEachStep == (iteration < STEPS) => <>(
        barrier = 0)
6   SimTermination == <>(iteration = STEPS)
7   NumSteps == [](iteration <= STEPS)
```