

# Relazione PCD Assignment 1 (Pinball)

Relazione di Daniele Tentoni, Lorenzo Pagnini, Alberto Antonelli

|                                |          |
|--------------------------------|----------|
| <b>Introduzione</b>            | <b>2</b> |
| Componenti del gruppo          | 2        |
| Analisi del problema           | 2        |
| <b>Progettazione</b>           | <b>4</b> |
| Entità Body                    | 4        |
| Entità Simulator               | 4        |
| <b>Realizzazione</b>           | <b>5</b> |
| Dispensazione di permessi      | 5        |
| Condizione di mutua esclusione | 5        |
| <b>Tests</b>                   | <b>7</b> |
| Test sul modello               | 7        |
| Prestazioni                    | 7        |
| Jpf                            | 8        |
| Fase di Update                 | 8        |
| Fase di collisione             | 9        |
| Test grafici                   | 10       |
| Con 100 palline e 5 thread.    | 10       |
| Con 1000 palline e 5 thread.   | 11       |
| Con 5000 palline e 5 thread.   | 11       |
| Test grafico con Jpf           | 11       |

# Introduzione

## Componenti del gruppo

Il gruppo è composto da Daniele Tentoni, Lorenzo Pagnini e Alberto Antonelli.

## Analisi del problema

L'obiettivo dell'assignment consiste nel realizzare un sistema concorrente, a partire da quello sequenziale, dell'algoritmo proposto che massimizzi le performance.

Di seguito si propone l'algoritmo in pseudo-codice che aggiorna tutti i corpi della simulazione e progredisce verso l'iterazione successiva.

```
vt = 0; /* virtual time */  
dt = 0.01; /* time increment at each iteration */  
loop {  
    Update bodies positions, given virtual time elapsed dt;  
    Check and solve collisions;  
    vt = vt + dt;  
    Display current stage;  
}
```

In un primo momento abbiamo pensato di organizzare il progetto attraverso un'architettura 'orizzontale' in cui i thread sono divisi in tre categorie di lavoro: aggiornamento della posizione delle palline (update), calcolo delle collisioni tra le palline (collider) e calcolo della collisione con il bordo del box (checkBoundary).

In questa architettura i thread, di una specifica categoria di lavoro, ad ogni iterazione dopo aver fatto le dovute computazioni, si sospendono in attesa dell'esecuzione degli altri thread per poi essere risvegliati all'iterazione successiva.

L'immagine seguente mostra l'idea dell'architettura:



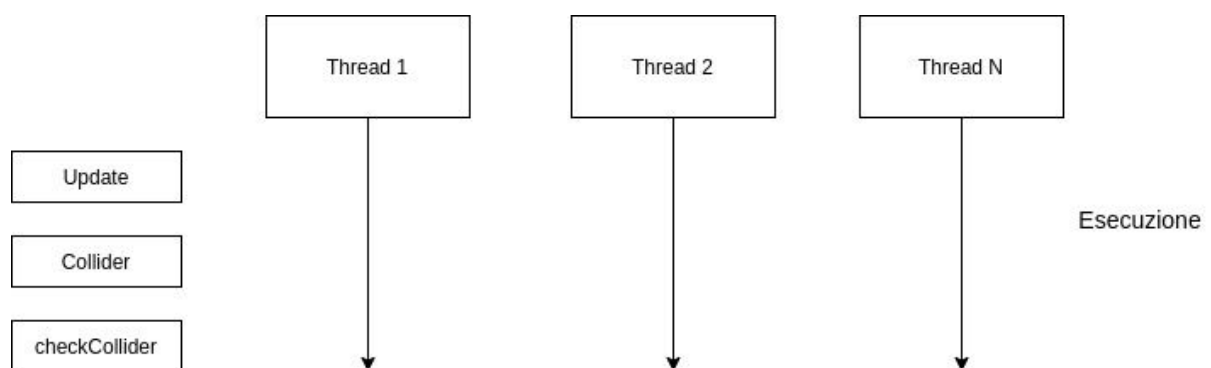
Il nome della struttura ('orizzontale') è dato dal fatto che l'esecuzione dei thread avviene in direzione orizzontale rispetto alla categoria di lavoro.

Dai test effettuati abbiamo analizzato che le performance invece di aumentare, rispetto al sistema sequenziale, sono peggiorate con un tempo di esecuzione molto maggiore.

Questo risultato si spiega dal fatto che ogni gruppo di thread di uno specifico lavoro si sospende per un lasso di tempo elevato tra una iterazione e l'altra, e questo ha portato ad un decadimento generale delle performance.

La seconda soluzione pensata, ovvero quella definitiva che abbiamo consegnato, rispecchia un'architettura 'verticale'. Ogni thread esegue tutte e tre le fasi (update, collider, checkBoundary) su tutte i corpi presenti. Si veda il paragrafo Progettazione, Realizzazione, e Test per i dettagli dell'architettura e considerazioni.

L'immagine seguente mostra l'idea dell'architettura:



In questo caso il nome 'verticale' è dato dall'esecuzione dei thread che avviene in direzione verticale rispetto alle fasi dell'algoritmo.

# Progettazione

## Entità Body

Abbiamo iniziato a progettare il programma partendo dall'entità corpo (nel codice Body), pensando che fosse l'entità che più richiedesse un refactor del codice per considerarla come entità sincronizzabile da più thread che lavorassero in parallelo.

Abbiamo quindi provveduto a sincronizzare i metodi che lavorano su variabili condivise e creato dei metodi (si veda in Realizzazione) che 'prendono' il permesso su un determinato corpo così che altri thread potessero scartare quel corpo poiché già processato o in fase di aggiornamento.

## Entità Worker e SimulationWorker

L'entità Worker è colei che rappresenta il singolo thread e che dovrà accedere alle varie proprietà della pallina in diversi modi. L'entità SimulationWorker contiene il metodo run() e il comportamento dei thread secondo il design verticale.

## Entità Simulator

Questa entità si occupa di generare i corpi della simulazione e i thread che si occuperanno del loro aggiornamento. Successivamente, si occuperà di farli avviare e attendere il loro completamento.

# Realizzazione

In questo paragrafo discutiamo dei meccanismi di coordinazione utilizzati all'interno del progetto realizzato.

## Dispensazione di permessi

Abbiamo creato dei metodi dentro l'entità Body da usare come "dispensatori di permessi" ai Worker che vi accedono per effettuare le varie operazioni. Tali metodi sono:

1) `synchronized boolean takeUpdate()`:

Un worker può richiamare questo metodo per chiedere alla pallina se può "assumersi l'incarico di aggiornare la loro posizione all'iterazione corrente. Se nessun altro Worker ha chiamato prima questo metodo, semplicemente la pallina "si lascia" aggiornare, altrimenti lo impedisce e il Worker deve passare ad un'altra pallina.

2) `synchronized boolean takeCollide()`

Similmente al precedente metodo, il worker chiede a questo metodo se può controllarne le collisioni con le altre palline.

3) `synchronized boolean takeViewer()`

Similmente ai precedenti metodo, il worker chiede a questo metodo se può effettuare l'aggiornamento della posizione della pallina nella view.

## Condizione di mutua esclusione

Oltre a questi metodi di sincronizzazione, abbiamo anche pensato che fosse necessario impedire a diversi Worker di accedere alla Velocità di un corpo durante la risoluzione delle collisioni tra corpi. Questo perché i Worker devono sempre accedere all'ultima velocità aggiornata.

Nella fase di lettura della velocità, abbiamo implementato una Condition Variables (velocityLock) per sospendere i thread nel caso non è possibile leggere la velocità:

```
while(!this.velocityAvailable) {  
    worker.log("Velocità ferma " + this.index);  
    this.velocityLock.await();  
}
```

e nella fase di scrittura, dopo la modifica della velocità, il thread risveglia i processi in attesa:

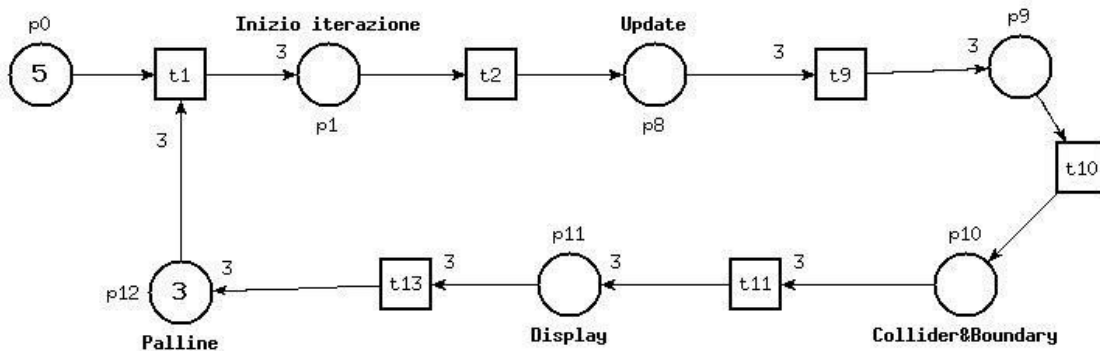
```

    vel.change(vx, vy);
    this.velocityAvailable = true;
    this.velocityLock.signalAll();

```

Infine è stata utilizzata una barriera ciclica tra un'iterazione e quella successiva. Alla fine di un'iterazione (dopo il controllo delle collisioni con il box), i thread vengono sospesi in attesa che tutti finiscano le computazioni per poi iniziare l'iterazione successiva tutti insieme.

Il progetto così realizzato può essere modellato con la Rete di Petri nel seguente modo:



Lo stato P12 contiene il numero di corpi (in questo caso 3 palline) mentre lo stato P0 il numero di iterazioni da eseguire (in questo caso 5 iterazioni).

Ad ogni iterazione vengono svolte le fasi di Update, Collider, Boundary e stampa nella Gui delle palline per poi iniziare una nuova iterazione finché non vengono esauriti i token nello stato P0.

# Tests

Il processore della macchina utilizzata per fare i test è un Intel(R) Core(™) i5-4440 a 3.10 GHz, con 1 processore fisico, 4 cores, 4 processori logici, senza virtualizzazione o hyper threading. La cache è suddivisa come 256KB, 1 MB e 6 MB.

Successivamente viene fatto riferimento a due tipi d'iterazioni diverse:

- Iterazione Globale: si intende l'iterazione di un singolo thread. Necessaria per la sincronizzazione tra di essi, viene inizializzata a 0 e un thread continua la sua esecuzione ciclicamente fino ad arrivare all'iterazione massima.
- Iterazione Locale: si intende l'iterazione di un singolo corpo. Utilizzata esclusivamente a fini di test, indica l'iterazione a cui un corpo è stato sottoposto durante un test (dopo un update, l'iterazione locale di un corpo è successiva a quella locale di tutta l'esecuzione).

## Test sul modello

### Prestazioni

Di seguito si propone la tabella contenente i vari test effettuati con varie configurazioni per testare il funzionamento e lo speed up della versione parallela a 1 thread con quella a 5 thread a parità di iterazioni e palline (Lo speed up è calcolato come rapporto tra tempo versione sequenziale e tempo versione parallela).

| Mettiamo a confronto le prestazioni con 500 iterazioni  |        |        |         |            |        |        |        |              |  |
|---|--------|--------|---------|------------|--------|--------|--------|--------------|--|
| Iterazioni  | Thread | Bodies | Time    | Iterazioni | Thread | Bodies | Time   | SpeedUp      |  |
| 500   | 1      | 100    | 173     | 500        | 5      | 100    | 198    | 0,8737373737 |  |
| 500   | 1      | 1000   | 11271   | 500        | 5      | 1000   | 3184   | 3,539886935  |  |
| 500   | 1      | 5000   | 283040  | 500        | 5      | 5000   | 60342  | 4,690596931  |  |
| Mettiamo a confronto le prestazioni con 1000 iterazioni |        |        |         |            |        |        |        |              |  |
| Iterazioni  | Thread | Bodies | Time    | Iterazioni | Thread | Bodies | Time   | SpeedUp      |  |
| 1000  | 1      | 100    | 272     | 1000       | 5      | 100    | 134    | 2,029850746  |  |
| 1000  | 1      | 1000   | 23030   | 1000       | 5      | 1000   | 6598   | 3,490451652  |  |
| 1000  | 1      | 5000   | 564386  | 1000       | 5      | 5000   | 123554 | 4,567929812  |  |
| Mettiamo a confronto le prestazioni con 5000 iterazioni |        |        |         |            |        |        |        |              |  |
| Iterazioni  | Thread | Bodies | Time    | Iterazioni | Thread | Bodies | Time   | SpeedUp      |  |
| 5000  | 1      | 100    | 1310    | 5000       | 5      | 100    | 638    | 2,053291536  |  |
| 5000  | 1      | 1000   | 113185  | 5000       | 5      | 1000   | 30762  | 3,679377154  |  |
| 5000  | 1      | 5000   | 2824459 | 5000       | 5      | 5000   | 608084 | 4,644850054  |  |

Speedup con 500 Iterazioni

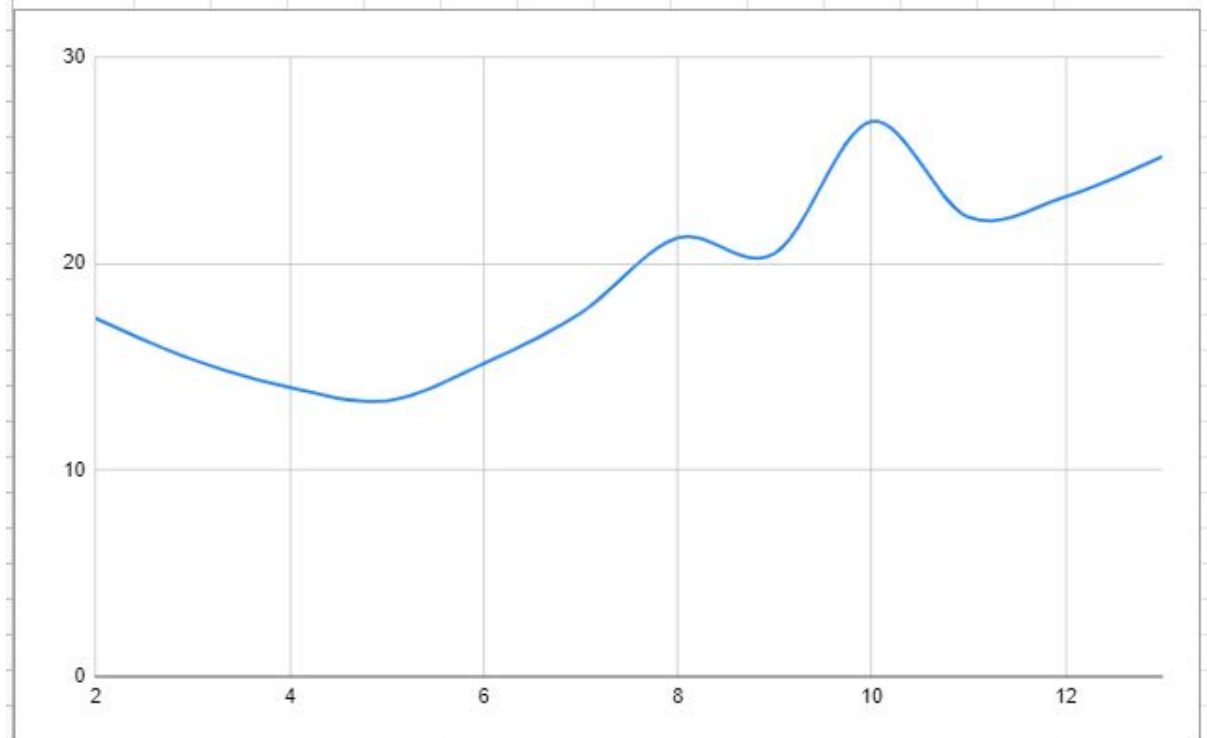
Speedup con 1000 Iterazioni

Speedup con 5000 Iterazioni

A parte per la computazione con 500 iterazioni e 100 body dove la versione con 1 solo thread risulta essere più performante, anche se di pochi ms, la versione a 5 thread risulta essere sempre migliore, fino ad arrivare ad un incremento delle performance di 4,5 volte con molte iterazioni e molti corpi.

Di seguito invece il grafico che mostra come, testando con 100 corpi e 100 iterazioni, prendendo una media tra 20 risultati, si arriva al tempo d'esecuzione minimo con 5 thread, cioè il numero teorico perfetto (dato il numero di core sulla macchina su cui si esegue il programma). Aumentando troppo il numero di thread, il tempo aumenta vertiginosamente. Tale incremento può essere causato dall'elevato tempo necessario alla gestione dei thread superflui.

|           |       |       |     |       |       |      |       |      |      |       |       |      |  |
|-----------|-------|-------|-----|-------|-------|------|-------|------|------|-------|-------|------|--|
| N Thread: | 2     | 3     | 4   | 5     | 6     | 7    | 8     | 9    | 10   | 11    | 12    | 13   |  |
| N Iter:   | 100   | 100   | 100 | 100   | 100   | 100  | 100   | 100  | 100  | 100   | 100   | 100  |  |
| N Body:   | 100   | 100   | 100 | 100   | 100   | 100  | 100   | 100  | 100  | 100   | 100   | 100  |  |
| Media:    | 17,35 | 15,35 | 14  | 13,35 | 15,15 | 17,6 | 21,25 | 20,5 | 26,9 | 22,25 | 23,25 | 25,2 |  |



## Jpf

Abbiamo fatto diversi controlli con Jpf perché risultava essere troppo pesante la computazione di tutto l'algoritmo, per cui l'abbiamo diviso in diversi sotto algoritmi da verificare singolarmente.

### Fase di Update

L'algoritmo da controllare è stato semplificato come segue:

```

Finché l'iterazione globale è inferiore all'iterazione massima
  Per ogni corpo
    se non è aggiornato
      lo aggiorni
      incremento l'iterazione locale
      incremento l'iterazione globale

```

La proprietà da verificare in questo modo si riduce a:



- a) L'iterazione locale del corpo è uguale all'iterazione globale prima che il corpo venga aggiornato;
- b) L'iterazione locale del corpo è pari a quella successiva dell'iterazione globale dopo che il corpo è stato aggiornato.

La valenza di questa condizione può essere provata nel codice utilizzando la classe UpdateMainJPF, restituendo questo output:

```

D:\WINDOWS\system32\cmd.exe
ork\Univ\pcd_assignments>java -jar JPF/jpf-core/build/RunJPF.jar +classpath=out/production/pcd_assignments
inballs.jpf.update.UpdateMainJpf
Pathfinder core system v8.0 (rev ae2cb8270b29f4adc74c017d353149684b521377) - (C) 2005-2014 United States Go
ent. All rights reserved.

===== system under test
pinballs.jpf.update.UpdateMainJpf.main()

===== search started: 11/04/20 15.22

===== results
errors detected

===== statistics
used time:      00:00:05
new=25482,visited=36661,backtracked=62143,end=3
ch:            maxDepth=532,constraints=0
ce generators:  thread=25482 (signal=779,lock=1579,sharedRef=19242,threadApi=166,reschedule=1832), data=0
:              new=14527,released=13860,maxLive=500,gcCycles=58081
ructions:      1985427
memory:        615MB
ed code:       classes=96,methods=2141

===== search finished: 11/04/20 15.22

```

## Fase di collisione

L'algoritmo da controllare è stato semplificato come segue:

Finché l'iterazione globale è inferiore all'iterazione massima

```

    Per ogni corpo b1
        se non è stato ancora fatto collidere
            Per ogni altro corpo b2
                se il corpo b1 collide con il corpo b2
                    risolvo la collisione tra i due
            controllo se il corpo b1 collide con i bordi
        incremento l'iterazione globale

```

Le proprietà da verificare sono:

- 1) Che l'iterazione locale di un corpo, dopo che è stato preso da un worker per controllare le collisioni con gli altri, sia maggiore o uguale a quelle degli altri. Infatti, non può essere che un corpo successivo a quello in corso sia ad una iterazione maggiore di quella corrente.
- 2) Similmente all'update, controlliamo che:
  - a) L'iterazione locale del corpo è uguale all'iterazione globale prima che vengano controllate le collisioni su di esso;

- b) L'iterazione locale del corpo è pari a quella successiva della globale dopo che vengano controllate le collisioni su di esso.

La valenza di questa proprietà può essere dimostrata utilizzando Jpf sulla classe CollisionMainJpf, che restituisce come output:

```
C:\WINDOWS\system32\cmd.exe

work\Univ\pcd_assignments>java -jar JPF/jpf-core/build/RunJPF.jar +classpath=out/production/pcd_assignments p
pinballs.jpf.collision.CollisionMainJpf
aPathfinder core system v8.0 (rev ae2cb8270b29f4adc74c017d353149684b521377) - (C) 2005-2014 United States Gov
ment. All rights reserved.

===== system under test
.pinballs.jpf.collision.CollisionMainJpf.main()

===== search started: 11/04/20 15.43

===== results
errors detected

===== statistics
elapsed time:      00:00:17
new=108451,visited=155712,backtracked=264163,end=3
rch:              maxDepth=559,constraints=0
ice generators:    thread=108451 (signal=3320,lock=7702,sharedRef=83745,threadApi=768,reschedule=7573), data=0
p:                new=57720,released=56032,maxLive=509,gcCycles=247443
tructions:        7961159
memory:           615MB
ded code:         classes=96,methods=2141

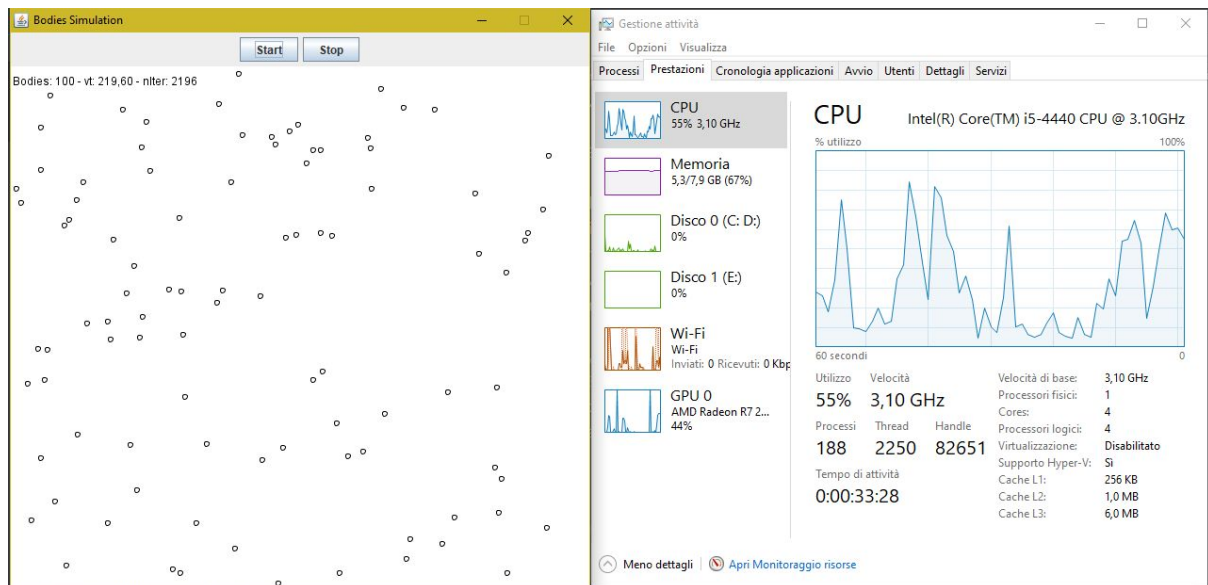
===== search finished: 11/04/20 15.44
```

## Test grafici

Di seguito si allegano le immagini dei test grafici con varie configurazioni. Il numero di iterazioni è sempre stato fissato a 5000, per avere abbastanza tempo per catturare le schermate.

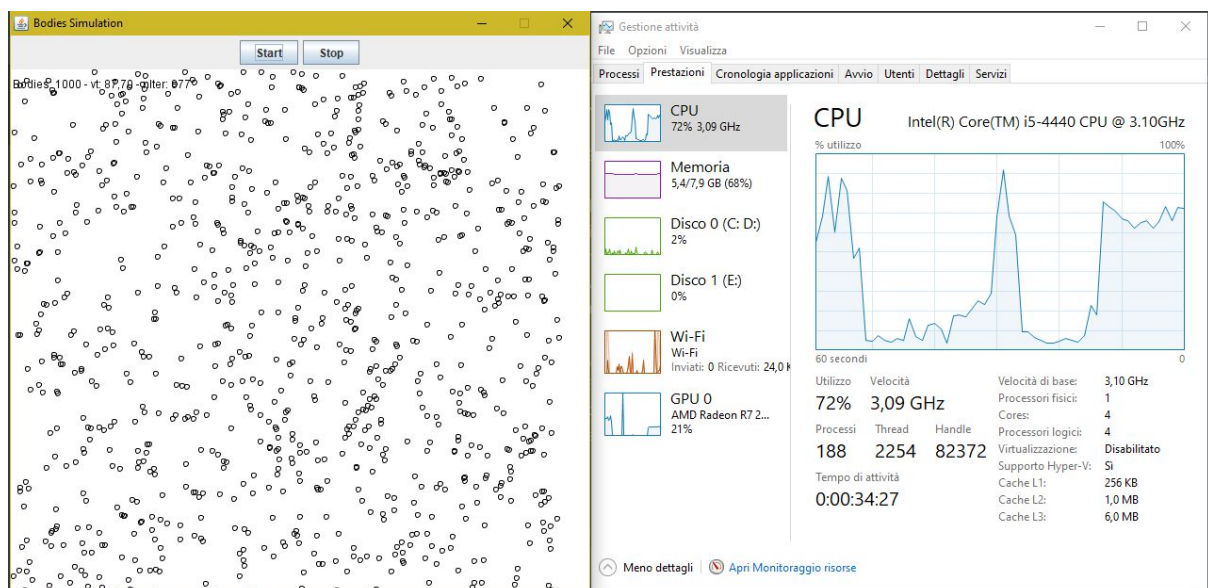
Il tempo di reazione del programma alla pressione dei bottoni è pari ad una iterazione, dato che il controllo viene eseguito solamente all'inizio di ogni iterazione, prima di compiere qualunque altra operazione.

## Con 100 palline e 5 thread



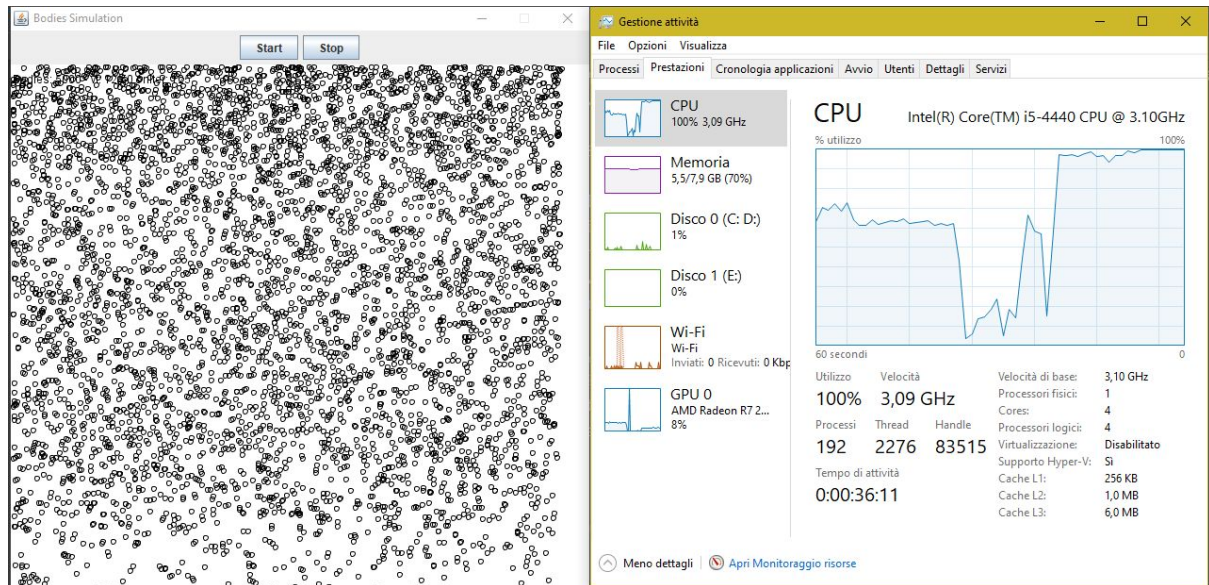
In questo test le palline si muovevano molto veloci, concludendo l'intera esecuzione delle 5000 iterazioni in ben meno di un minuto.

## Con 1000 palline e 5 thread



In questi test le palline si muovevano comunque con una discreta velocità, richiedendo appena più di un minuto per terminare le 5000 iterazioni.

Con 5000 palline e 5 thread



In questo test le palline si muovevano così lentamente che era possibile visualizzare le singole iterazioni a occhio nudo. Solamente in questa prova si è notato un utilizzo della CPU pressoché totale da parte del programma.

## Test grafico con Jpf

Abbiamo creato un nuovo Worker che simulasse il funzionamento di EDT e una nuova View che simulasse la delegazione a EDT di renderizzare le nuove posizioni prodotte dagli altri Workers. In comune possiedono un Bounded Buffer che simula la coda degli eventi di EDT. La View si occupa di riempire tale buffer e il nostro EDT si occupa di leggere tali valori.

L'algoritmo semplificato del worker che crea la risorsa che il nostro EDT deve leggere è:

```
Finché l'iterazione globale è inferiore all'iterazione massima
```

```
    incremento l'iterazione corrente
```

```
    Se il corpo 0 non è ancora stato mostrato
```

```
        Dico all'EDT di mostrare a video i corpi
```

L'algoritmo semplificato dell'EDT che consuma la risorsa è:

```
Finché l'iterazione globale è inferiore all'iterazione massima
```

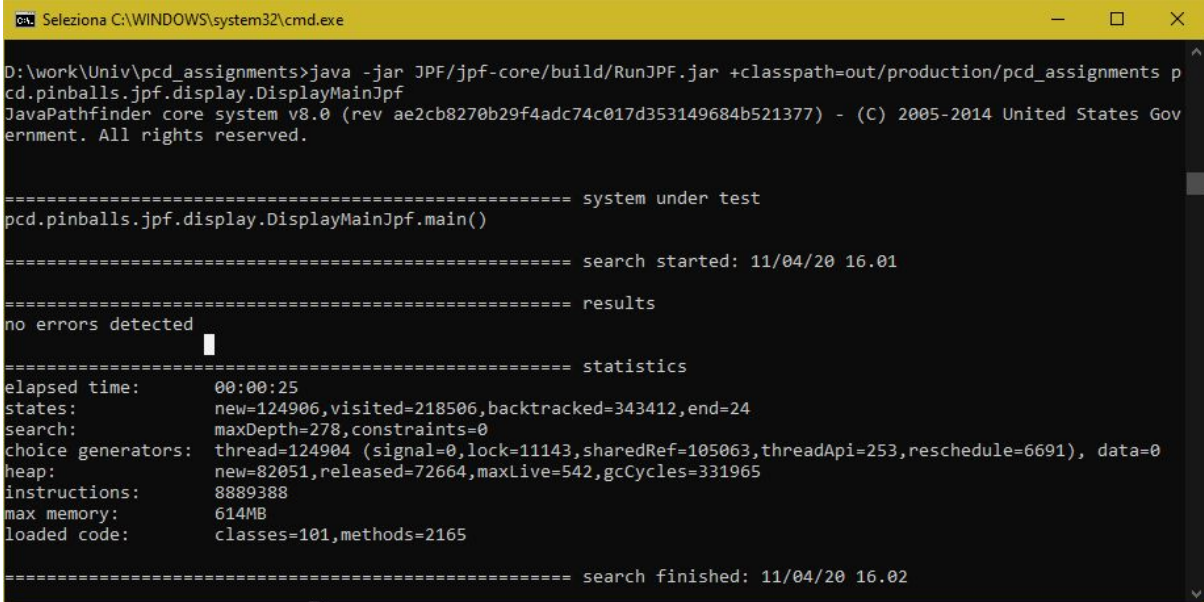
```
    mostro a video i corpi
```

```
    incremento l'iterazione globale
```

La proprietà da verificare è:

- 1) Che l'iterazione globale dell'edt non sia mai maggiore di quella massima tra tutti i singoli workers.

La valenza di questa proprietà può essere dimostrata utilizzando Jpf sulla classe DisplayMainJpf, che restituisce come output:



```
Seleziona C:\WINDOWS\system32\cmd.exe

D:\work\Univ\pcd_assignments>java -jar JPF/jpf-core/build/RunJPF.jar +classpath=out/production/pcd_assignments p
cd.pinballs.jpf.display.DisplayMainJpf
JavaPathfinder core system v8.0 (rev ae2cb8270b29f4adc74c017d353149684b521377) - (C) 2005-2014 United States Gov
ernment. All rights reserved.

===== system under test
pcd.pinballs.jpf.display.DisplayMainJpf.main()

===== search started: 11/04/20 16.01

===== results
no errors detected

===== statistics
elapsed time:      00:00:25
states:           new=124906,visited=218506,backtracked=343412,end=24
search:           maxDepth=278,constraints=0
choice generators: thread=124904 (signal=0,lock=11143,sharedRef=105063,threadApi=253,reschedule=6691), data=0
heap:             new=82051,released=72664,maxLive=542,gcCycles=331965
instructions:     8889388
max memory:       614MB
loaded code:      classes=101,methods=2165

===== search finished: 11/04/20 16.02
```