# Corso di Programmazione Concorrente e Distribuita

## Relazione Assignment #1

Simulazione del movimento di N corpi in un piano bidimensionale

Alessandro Magnani, Simone Montanari, Andrea Matteucci

# Analisi del problema

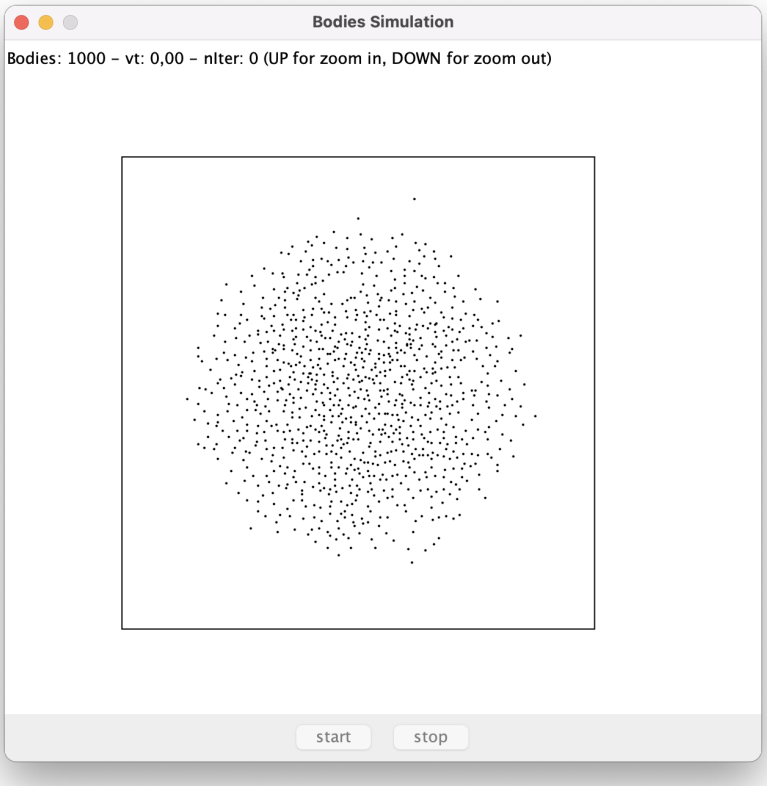
Il problema riguarda la gestione concorrente di un numero N di palline che si muovono in uno spazio bidimensionale, soggette a due tipi di forze:

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

Nella versione sequenziale fornita era già presente l'algoritmo per poter gestire il movimento delle palline (calcolo delle forze, aggiornamento delle posizioni e controllo di eventuali collisioni con i bordi) e una GUI per poter visualizzare il funzionamento del sistema.

Dunque, il principale problema da affrontare è quello di capire come realizzare una versione concorrente dell'algoritmo sequenziale fornito in precedenza, eventualmente aiutandosi sfruttando una GUI.

È di seguito riportato uno screenshot del sistema in esecuzione.



# Architettura proposta e strategia risolutiva

Inizialmente, durante la prima parte della fase di problem-solving, è stato visionato il codice cercando di capirne il funzionamento interno e, successivamente, è stata modificata la struttura cercando di trasformarla seguendo il pattern architetturale MVC (Model-View-Controller).

Durante la seconda parte della fase di problem-solving sono state discusse possibili soluzioni per la gestione della concorrenza e si è optato per un'architettura **Master-Workers** (in cui è stato utilizzato un **Latch** per gestire la comunicazione tra il master ed i Workers, 3 **CyclicBarrier** per gestire la concorrenza tra i Workers e, nella versione con GUI, un **Monitor** per riuscire a gestire concorrentemente un oggetto stopFlag).

## Gestione delle palline e della visualizzazione nella GUI

Inizialmente la scelta implementativa è stata quella di suddividere le palline in gruppi e passare ad ogni Worker una sottolista di quella principale (contenente solamente le palline relative al singolo Worker). Agendo in questo modo, però, si è notato come la gestione della visualizzazione nella GUI risultasse complicata: occorreva "riassemblare" le varie sottoliste in una lista unica contenente tutte le palline.

Per questo motivo si è deciso di passare a tutti i Workers la stessa lista contenente tutte le palline, con l'aggiunta del numero di palline relative al Worker e di un indice di inizio, affinché ciascuno potesse gestire un numero limitato di palline ma avesse il riferimento anche alle modifiche apportate dagli altri Workers, rendendo di fatto la lista contenente tutte le palline come una risorsa condivisa.
In questo modo si è potuto così evitare il problema citato precedentemente e, dunque, semplificare notevolmente il processo di visualizzazione nella GUI.

Inoltre, grazie all'indice passato ad ogni Worker è possibile (tramite un semplice if in cui si controlla che l'indice di inizio corrisponda all'ultimo sottogruppo di palline di tutta la lista) riuscire a far eseguire la chiamata alla view solamente all'ultimo Worker.

# Gestione del latch

Si è optato di utilizzare un latch per garantire la comunicazione tra il Master e i Workers. Infatti il Master, dopo aver creato i Workers, si mette in attesa sul latch, per attendere il completamento del lavoro da parte dei Workers.

# Gestione delle barriere

La scelta di utilizzare le barriere nasce dalla volontà di sincronizzare le varie operazioni che vengono compiute dai Workers sulle palline. L'idea è proprio quella che una volta che un thread ha svolto una determinata operazione, si bloccherà aspettando la conclusione della medesima operazione da parte di tutti gli altri.

In particolare, il compito di ogni Worker è quello di svolgere le seguenti operazioni sulle palline:

- Calcolo delle forze
- Aggiornamento delle posizioni
- Controllo delle collisioni col bordo
- Visualizzazione nella GUI (eseguita solamente dal Worker che gestisce l'ultimo gruppo di palline)

Come spiegato, in ognuna di esse è presente il vincolo di poter eseguire l'operazione successiva solamente dopo aver eseguito quella corrente su tutte le palline: due delle tre barriere sono utilizzate per gestire questo meccanismo.

Nello specifico:

- I Workers eseguono la prima operazione sul proprio gruppo di palline e si mettono in attesa che tutti abbiano finito (prima barriera)
- Quando l'ultimo Worker ha svolto la prima operazione "sblocca" gli altri
- I Workers eseguono la seconda e la terza operazione sul proprio gruppo di palline e si mettono in attesa che tutti abbiano finito (seconda barriera)

A questo punto occorre aggiornare la GUI e solamente un Worker dovrà farlo (terza barriera). Si è deciso che l'ultimo Worker si occupa di richiamare la *display()*.

È molto importante soffermarsi sul fatto che si è scelto di utilizzare barriere cicliche (attraverso la classe *CyclicBarrier*). Questo perché la simulazione doveva continuare per un numero preimpostato di iterazioni e l'utilizzo di barriere "classiche" sarebbe risultato scomodo e poco efficiente in quanto obbligava a re-istanziarle ad ogni iterazione.

## Gestione del monitor

Per regolare il comportamento dei pulsanti “start” e “stop”, è stato implementato un monitor che regola una variabile *stopFlag* al fine di consentire sia l’avvio, che l’interruzione del programma attraverso la GUI.

# Comportamento del sistema

È di seguito riportato il comportamento del sistema:

- Inizialmente viene istanziato un oggetto *SimulationView*;

- Nel costruttore di *SimulationView* viene istanziato il controller della GUI (in attesa di chiamate da parte degli ActionListener dei bottoni) e un oggetto *VisualizerFrame*;

- In *VisualizerFrame* il codice è stato leggermente modificato aggiungendo, oltre al pannello già presente contenente le palline, uno per i due pulsanti ed uno che contenesse entrambi;

- A questo punto per iniziare l'esecuzione della simulazione occorre premere il tasto start che notifica al controller (metodo *notifyStarted()*) di iniziare l'esecuzione;

- Il controller genera un numero preimpostato di palline, istanzia il Master Agent (classe *MyMasterAgent*) ed esegue il Thread Master (metodo *start()*);

- Nel costruttore del Master Agent vengono istanziate le 3 *CyclicBarrier* ed il Latch (classe *TaskCompletionLatch*);

- Nel corpo del Thread Master (metodo *run()*) viene inizialmente richiamato il metodo *assignBallsToWorkers()* che istanzia tanti Worker Agents quanti sono i Core disponibili in quel momento, assegna loro un numero bilanciato di palline e dà inizio alla loro esecuzione (metodo *start()*).
  Successivamente il Master Agent è messo in attesa del completamento del lavoro da parte dei Worker Agents (metodo *waitCompletion()*) e, una volta "sbloccato", termina l'esecuzione del programma stampando a video il tempo totale;

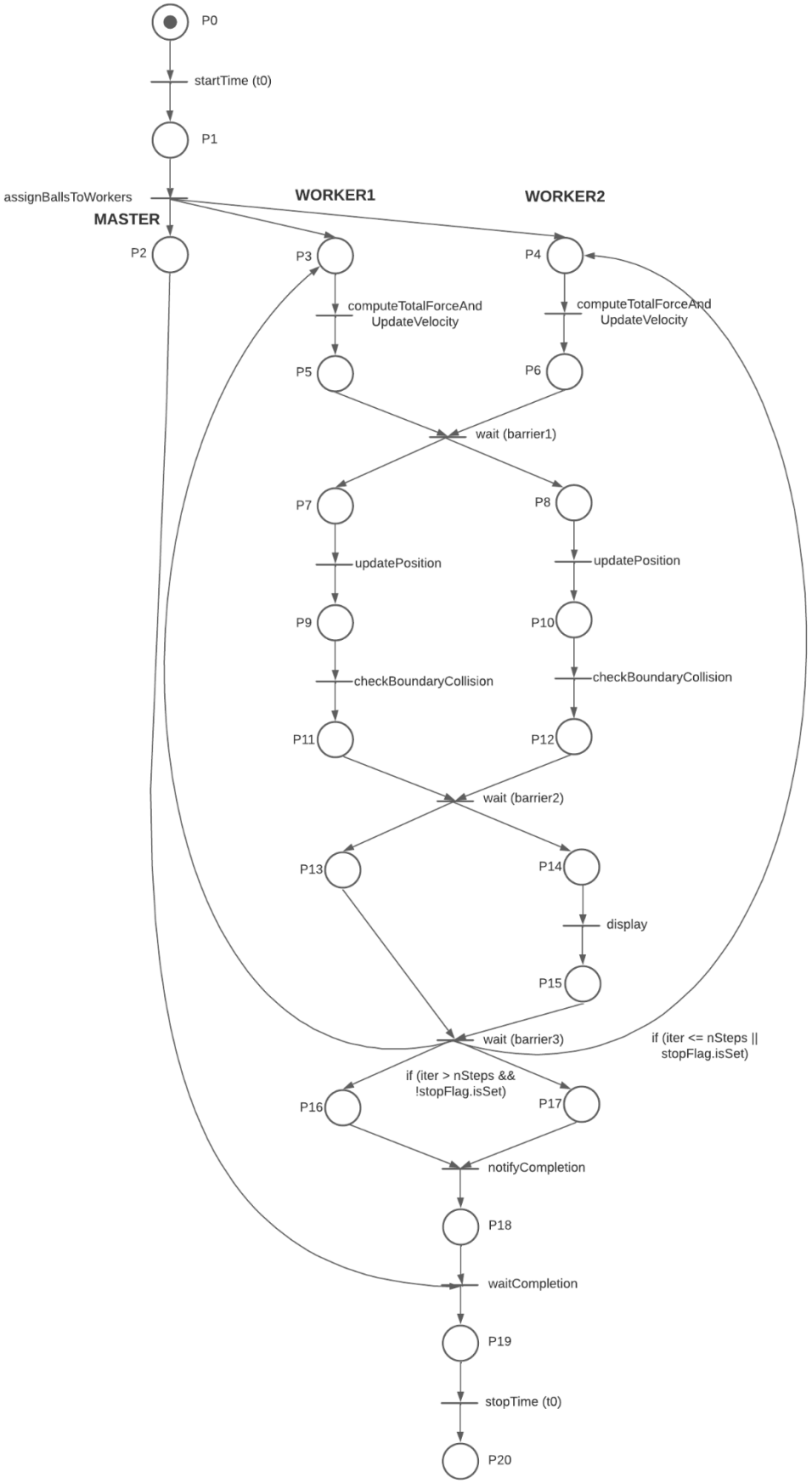
- Il corpo dei Thread Workers (metodo *run()*) è formato da un ciclo while (con condizioni riguardanti il numero di iterazioni e l'oggetto *Flag*) con il seguente funzionamento:
  - Computazione delle forze e aggiornamento delle velocità delle palline relative al Worker (metodo *computeTotalForceAndUpdateVelocity()*);
  - Attesa degli altri Workers tramite la prima barriera;

    - Aggiornamento delle posizioni e controllo su eventuali collisioni con il bordo delle palline relative al Worker (metodi *updatePosition()* e *checkBoundaryCollision()*);
    - Attesa degli altri Workers tramite la seconda barriera;
    - Esecuzione della display, da parte dell'ultimo Worker (metodo *display()*);
    - Attesa degli altri Workers tramite la terza barriera;

- A questo punto, una volta terminato il proprio compito, i Workers lo notificano al Master sbloccando il Latch (metodo *notifyCompletion()*);

- È possibile interrompere l'esecuzione del programma premendo il tasto "stop" (visibile solamente dopo aver premuto il pulsante di avvio) che notifica al controller di interrompere la simulazione (metodo *notifyStopped()*) ed esso modificherà lo stato dell'oggetto *Flag* (in questo modo la condizione del while nel corpo dei Workers non sarà più soddisfatta e, interrompendo il sistema).

*Una versione PDF della rete di Petri è fornita in allegato*

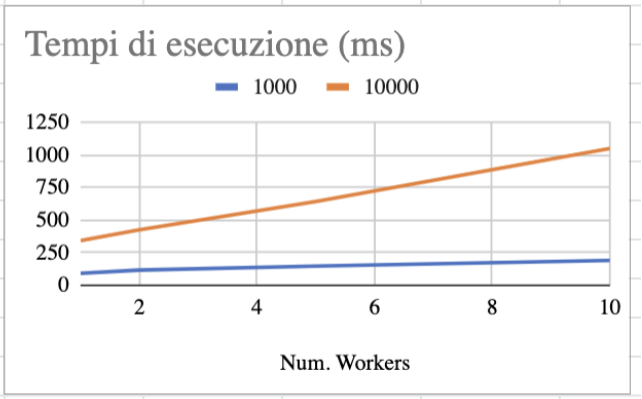


# Prove di performance

100 corpi

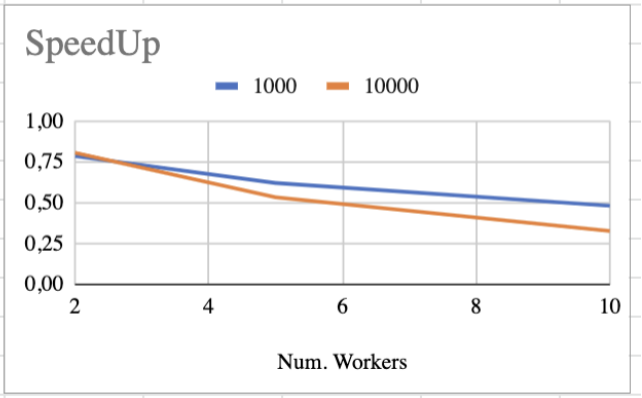
**100 corpi**
**Tempi di esecuzione (ms)**

| | Iterazioni | |
|---|---|---|
| Num. Worker | 1000 | 10000 |
| 1 | 92 | 343 |
| 2 | 117 | 425 |
| 5 | 148 | 642 |
| 10 | 191 | 1047 |



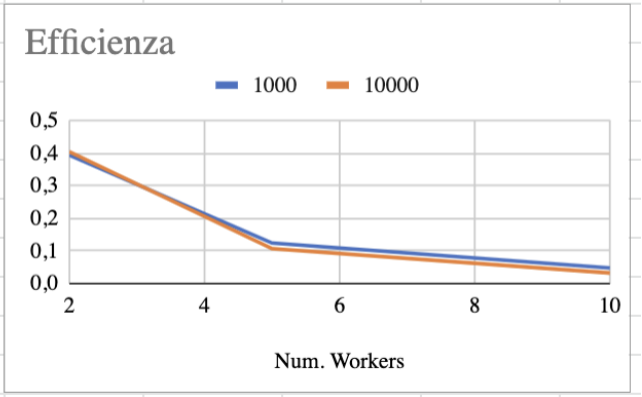

**100 corpi**
**SpeedUp**

| | Iterazioni | |
|---|---|---|
| Num. Worker | 1000 | 10000 |
| 2 | 0,7863247 | 0,8070588 |
| 5 | 0,6216216 | 0,5342679 |
| 10 | 0,4816753 | 0,3276026 |




**100 corpi**
**Efficienza**

| | Iterazioni | |
|---|---|---|
| Num. Worker | 1000 | 10000 |
| 2 | 0,3931623 | 0,4035294 |
| 5 | 0,1243243 | 0,1068535 |
| 10 | 0,0481675 | 0,0327602 |

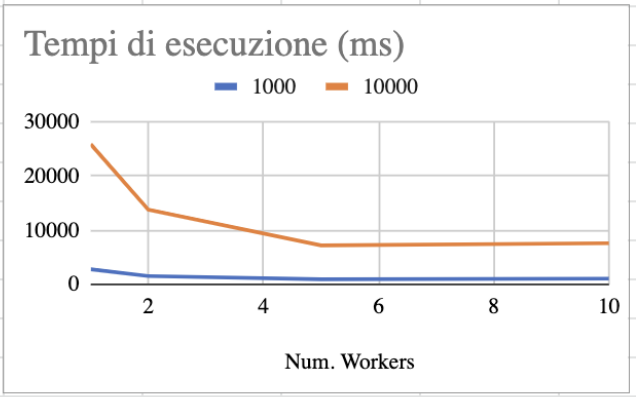



Il primo test eseguito sulle performance è quello che comprende 100 corpi. Possiamo notare come con un numero di corpi che ancora non è molto alto, non si hanno grandi vantaggi: infatti il tempo di esecuzione sequenziale è migliore rispetto agli altri. Di conseguenza anche SpeedUp ed Efficienza vanno peggiorando all'aumentare del numero di Workers e di iterazioni.

# 1000 corpi

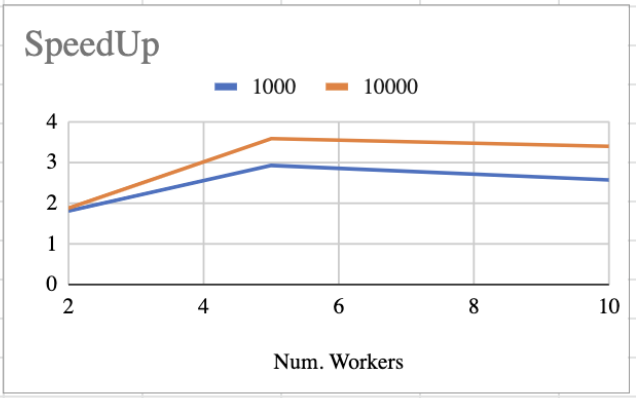
**1000 corpi**
**Tempi di esecuzione (ms)**

| | Iterazioni | |
|---|---|---|
| Num. Workers | 1000 | 10000 |
| 1 | 2804 | 25845 |
| 2 | 1552 | 13775 |
| 5 | 958 | 7210 |
| 10 | 1091 | 7611 |




**1000 corpi**
**SpeedUp**

| | Iterazioni | |
|---|---|---|
| Num. Worker | 1000 | 10000 |
| 2 | 1,8067010 | 1,8762250 |
| 5 | 2,9269311 | 3,5846047 |
| 10 | 2,5701191 | 3,3957430 |




**1000 corpi**
**Efficienza**

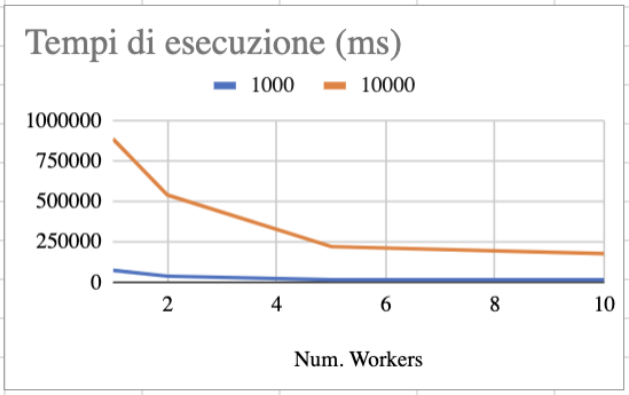
| | Iterazioni | |
|---|---|---|
| Num. Worker | 1000 | 10000 |
| 2 | 0,9033505 | 0,9381125 |
| 5 | 0,5853862 | 0,7169209 |
| 10 | 0,2570119 | 0,3395743 |


Con 1000 corpi si verificano diversi cambiamenti nel comportamento del sistema. Infatti notiamo subito che il tempo di esecuzione sequenziale, quindi quello appartenente ad un solo Worker, è maggiore rispetto a quelli concorrenti. Questo è dovuto al fatto che l'uso di più Workers, già con 1000 corpi, velocizza molto l'esecuzione dell'intero sistema.
Di conseguenza anche lo SpeedUp migliora aumentando via via il numero di Workers. Infine notiamo un fattore importante riguardo all'efficienza: l'uso di più core di esecuzione non garantisce un suo miglioramento. Vedremo nell'ultimo caso, quello con 5000 corpo, il perché di questo fenomeno.

# 5000 corpi

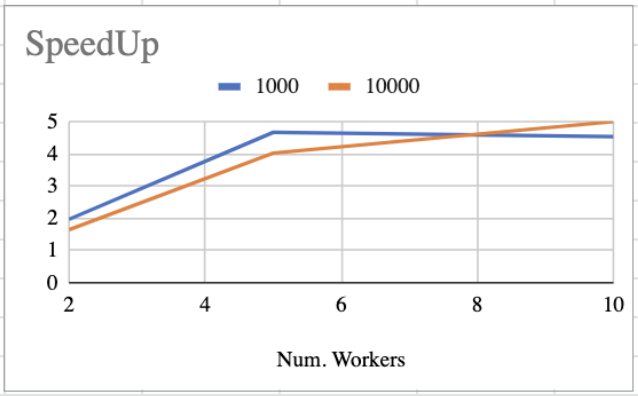
**5000 corpi**
**Tempi di esecuzione (ms)**

| | Iterazioni | |
|---|---|---|
| Num. Worker | 1000 | 10000 |
| 1 | 74663 | 886545 |
| 2 | 37965 | 539008 |
| 5 | 16025 | 220561 |
| 10 | 16482 | 177521 |




**5000 corpi**
**SpeedUp**

| | Speedup | |
|---|---|---|
| Num. Worker | 1000 | 10000 |
| 2 | 1,9666271 | 1,6447715 |
| 5 | 4,6591575 | 4,0195002 |
| 10 | 4,5299720 | 4,9940288 |




**5000 corpi**
**Efficienza**

| | Efficienza | |
|---|---|---|
| Num. Worker | 1000 | 10000 |
| 2 | 0,9833135 | 0,8223857 |
| 5 | 0,9318315 | 0,8039000 |
| 10 | 0,4529972 | 0,4994028 |


Utilizzando 5000 corpi le operazioni da svolgere aumentano ancora, e l'uso di un numero di Workers sempre maggiore risulta essere positivo per il tempo di esecuzione e per lo SpeedUp.
Come accennato nell'esempio precedente, questo non si può dire per l'efficienza. Infatti questa risulta peggiorare quando oltre ai 5 core fisici vengono utilizzati gli altri 5 core logici del computer su cui sono stati effettuati i test.
Possiamo quindi concludere che, per avere un trade-off tra questi tre parametri, conviene utilizzare 5 core (5 Workers) per avere la soluzione più performante del sistema.

# Identificazione proprietà di correttezza e verifica

Nel nostro sistema si è utilizzata l'architettura Master-Workers, e per verificare tutti gli scenari di esecuzione, si è utilizzato Java Pathfinder.
Si è scelto di identificare le proprietà di correttezza basandosi su due casi significativi:

- 2 palline, 2 Woker, 2 step
- 4 palline, 2 Worker, 2 step

In ognuno di questi, durante l'esecuzione, non sono stati riscontrati errori.

*2 palline, 2 worker, 2 step*

```
====================================================== results
no errors detected

====================================================== statistics
elapsed time:       00:00:29
states:             new=206666,visited=466220,backtracked=672886,end=45
search:             maxDepth=394,constraints=0
choice generators:  thread=206666 (signal=5760,lock=8289,sharedRef=171286,threadApi=11,reschedule=21320), data=0
heap:               new=38307,released=568824,maxLive=455,gcCycles=640820
instructions:       7137898
max memory:         501MB
loaded code:        classes=81,methods=1865

====================================================== search finished: 08/04/22 20.18
```

*4 palline, 2 worker, 2 step*

```
====================================================== results
no errors detected

====================================================== statistics
elapsed time:       00:04:14
states:             new=1799172,visited=4120180,backtracked=5919352,end=45
search:             maxDepth=986,constraints=0
choice generators:  thread=1799172 (signal=14472,lock=21750,sharedRef=1603967,threadApi=11,reschedule=158972), data=0
heap:               new=303401,released=4783084,maxLive=465,gcCycles=5714587
instructions:       63613399
max memory:         500MB
loaded code:        classes=81,methods=1865

====================================================== search finished: 08/04/22 20.17
```

# Conclusioni

Inizialmente abbiamo riscontrato diverse difficoltà nell'approccio al problema, per questo abbiamo deciso di dedicare molto tempo alla fase di problem solving per cercare di individuare la struttura più consona nel rispondere ai problemi di concorrenza citati precedentemente.
Successivamente, pianificata la struttura, abbiamo costruito il sistema con maggiore facilità. In questa fase, abbiamo ritenuto molto utile il materiale fornito durante i laboratori.

In conclusione, siamo dell'idea che la realizzazione di questo primo Assignment ci abbia aiutato a comprendere ancora meglio i concetti spiegati durante le lezioni teoriche.