PCD: Elaborato uno

Bachetti Michele, Marcantognini Elia, Tinti Angelo

Marzo 2022

Indice

Introduzione

Il progetto si pone come obiettivo quello di realizzare una versione concorrente del programma presentato a lezione, ovvero una simulazione del movimento di corpi puntiformi in uno spazio bidimensionale limitato. Il programma visto possedeva già buona parte del materiale necessario per creare il programma ed anche una sua interfaccia grafica.

In "Analisi del problema" che segue, sarà possibile visionare l'analisi che è stata fatta per raggiungere la soluzione consegnata. Sono quindi presenti i diversi ragionamenti incrementali che hanno portato a una versione ottimale.

In "Architettura e implementazione" vengono mostrati i compiti delle varie classi utilizzate e come cooperano fra di loro. Sono presenti in particolare i metodi utilizzati per evitare corse critiche e attuare le sincronizzazioni.

Infine, in "Verifica e prestazioni" sono discussi i test effettuati sul sistema, le statistiche ottenute utilizzando diverse varianti del sistema, compresa quella fornita inizialmente. Vengono inoltre presentati i controlli effettuati tramite *Java PathFinder* che abbiamo visto a lezione per i relativi problemi che possono insorgere nei programmi concorrenti.

Per visualizzare il codice è presente il repository su GitHub alla pagina `https://github.com/eliamarcantognini/pcd2122_ass1`.

Capitolo 1

Analisi del problema

Il sistema da implementare deve gestire il comportamento di corpi in un mondo bidimensionale. Il movimento dei corpi nella simulazione è dato dalle forze che agiscono su di loro e, inoltre, dai limiti del mondo che portano i corpi a rimbalzare con la possibilità di cambiare velocità. Gli elementi base che costituiscono la simulazione sono quindi:

Corpi Elementi puntiformi in grado di muoversi all'interno del mondo. Possiedono diverse proprietà: **posizione, velocità e massa**. La velocità del corpo è data dalle forze che vengono esercitate su di esso.

Forze Forze in grado di modificare la velocità e le traiettorie compiute dai corpi. Poossono essere di due tipologie:

- *Forza repulsiva*: data dalla massa e la distanza da ogni corpo. Tale forza tra un corpo b_i e b_j è così calcolabile:

$$ForzaRepulsiva_{ij} = \frac{k_{rep} \cdot m_i}{d_{ij}^2} \tag{1.1}$$

  dove k_{rep} è una costante fornita dal sistema, d_{ij} la distanza fra due corpi e m_i è la massa del corpo. La direzione della forza è data dal $versore = (b_j - b_i)$ – ovvero respingente per il corpo b_j.

- *Forza d'attrito*: su ogni corpo in moto è esercitata una forza d'attrito che si oppone al movimento.

$$ForzaAttrito_i = -(k_i \cdot v_i) \tag{1.2}$$

Al sistema sono richiesti alcuni vincoli:

- Il calcolo delle forze al tempo t deve avvenire considerando le corrette posizioni dei corpi che essi hanno al medesimo tempo t.

- L'aggiornamento delle posizioni può avvenire solamente dopo che tutte le nuove forze sono state aggiornate.

Il sistema inoltre, terminata la parte che gestisce la fisica della simulazione, dovrà essere esteso implementando un'interfaccia grafica per mostrare l'andamento della simulazione. La scena mostrata dovrà essere coerente e sincrona con la "scena" logica raggiunta dal sistema e quest'ultima potrà procedere all'iterazione successiva solamente dopo essere stata mostrata correttamente nell'interfaccia grafica. L'interfaccia oltre a ciò dovrà possedere due bottoni: "`START`" e "`STOP`"; in grado di fermare la simulazione e farne iniziare una nuova successivamente.

Un altro aspetto importante da considerare nella realizzazione del sistema è quello di renderlo il più efficiente più possibile grazie alle metodologie di sviluppo concorrente viste a lezione.

Sotto questo aspetto la problematica principale è data dalla necessità di parallelizzare la computazione relativa alle forze in gioco su ogni corpo, dato che queste ricoprono lo sforzo algoritmo principale e sono per lo più indipendenti tra loro.

Capitolo 2

Strategia Risolutiva

2.1 Prima soluzione

Trattandosi di un programma concorrente la prima problematica da risolvere è stata quella di individuare quali sezioni fossero tra loro indipendenti e quindi eseguibili nello stesso momento. Come descritto nel capitolo precedente la scelta è stata quella di rendere le computazioni delle forze agenti sui diversi corpi parallele tra loro, tenendo però in considerazione la necessità di gestire la possibile corsa critica per il reperimento delle informazioni necessarie ai vari calcoli.

2.1.1 Entità attive

Una prima soluzione sperimentata è stata quella di seguire il pensiero più logico che si può avere nell'analizzare il problema: un *mapping* uno a uno tra le entità attive in esecuzione nel sistema e i corpi da gestire. In questo modo è possibile ottenere la parallelizzazione massima del sistema, in quanto ogni singola computazione legata al calcolo delle forze è eseguita parallelamente alle altre.

Per la realizzazione di queste entità attive si è deciso di adottare il *modello ad Agenti*, quindi seguendo come linea di principio il fatto che dovevano essere completamente indipendenti tra loro e che come mezzo di comunicazione potessero utilizzare solo oggetti passivi esterni, nel nostro caso variabili condivise, senza richiamarsi direttamente.

In pratica ad ogni ciclo di esecuzione del sistema ognuno di questi Agenti ha il compito di:

1. Computare il valore delle forze agenti sul corpo da lui gestito, in base alle formule presentate in "Analisi del problema"
2. Determinare la velocità vettoriale risultante dall'applicazione delle forze sul corpo
3. "Spostare" il corpo sul piano 2D seguendo la velocità ricavata, modificando le sue coordinate

2.1.2 Concorrenza

Considerando che gli Agenti dovranno lavorare parallelamente, specialmente per l'esecuzione del punto 1, si potrebbe verificare una situazione di corsa critica per l'accesso o la modifica dei dati condivisi. Per risolvere questa problematica si è deciso di separare fisicamente gli oggetti tra lettura e scrittura. In questo modo non può accadere che un'entità attiva modifichi un dato relativo a un corpo, per esempio la sua posizione, mentre un'altra cerca di leggerlo in quanto staranno, nell'effettivo, operando su oggetti con stessi dati ma fisicamente distinti.

Si è anche ragionato sulla possibilità di utilizzare un "monitor" per gestire l'accesso a queste risorse, ma, a causa della loro realizzazione in Java, avrebbe reso sequenziale l'accesso anche in lettura di un certo dato, situazione che avrebbe sicuramente gravato sulle prestazioni generali.

Inoltre, si è presa in considerazione anche l'idea di realizzare un monitor per gestire l'accesso ad una specifica risorsa, quindi uno specifico corpo, per impedire la possibilità

di far avvenire una modifica e una lettura contemporaneamente, riprendendo l'idea della soluzione al problema dei lettori e scrittori presentata in letteratura.

Ragionando in questo modo, però, si sarebbero verificati dei problemi riguardo alla consistenza dei dati. In particolare, se si aggiornasse la posizione di un corpo prima che tutti gli altri abbiano letto il suo valore per calcolare le forze, i dati non sarebbero congrui all'iterazione della quale si stanno computando le forze poiché alcune posizioni sarebbero relative alla situazione precedente mentre altre sarebbero già state aggiornate.

2.1.3 Sincronizzazione

Per quanto riguarda la sincronizzazione del sistema, data la natura del tempo nel simulatore (ovvero il dover aggiornare il suo stato solo dopo che tutte le nuove posizioni fossero state modificate), c'era la necessità di un meccanismo che permettesse di agire su tale aggiornamento ma che al tempo stesso non facesse proseguire le entità fino alla sua conclusione. Per raggiungere questo obiettivo si è scelto di utilizzare una struttura dati di tipo barriera[1].

2.1.4 Visualizzazione a schermo

Alla natura del tempo del sistema si ricollega anche la decisione di effettuare la visualizzazione su GUI delle modifiche alla posizione dei corpi solo dopo l'avvenuta computazione di tutti i componenti e non seguendo una tecnica a *frame per second*. In questo modo si è sicuri di rispettare il requisito di coerenza tra ciò che viene mostrato a schermo e lo stato effettivo interno del sistema e corpi. D'altro canto, il dover attendere la terminazione dei componenti porta all'impossibilità di aggiornare la visualizzazione se questi vengono interrotti nella loro esecuzione.

2.2 Problematiche riscontrate nel modello iniziale

Effettuando dei test (riportati nel capitolo 4) l'idea di utilizzare un'entità per singolo corpo si è dimostrata poco efficiente, addirittura peggiore di quella sequenziale quando si andava a lavorare con un numero elevati di corpi.

Questo comportamento è dovuto al fatto che il numero di entità attive create che si andavano a creare non teneva conto del numero di thread fisici su cui venivano eseguiti, portando ad una situazione in cui il carico di lavoro dovuto alle operazioni necessarie per l'*interleaving* dei processi era talmente elevato da portare a prestazione inferiori della versione sequenziale.

Oltretutto, il sistema così congegnato non risultava essere scalabile all'aumentare o al diminuire delle risorse messe a disposizione dal calcolatore sul quale il programma è in esecuzione, rendendolo particolarmente sensibile a sistemi che dispongono di pochi thread, sia fisici che virtuali.

2.3 Evoluzione del modello iniziale

Le problematiche riportate nella sezione precedente hanno portato a un cambio di strategia. In particolare, si è deciso di mettere in campo un numero di entità attive pari a quello dei core che l'applicazione ha a disposizione e suddividere la gestione dei corpi tra questi.

In questo modo si ottiene una soluzione più efficiente e scalabile, perché si è sicuri che i flussi di controllo creati siano sempre in esecuzione invece che essere sospesi a causa della discrepanza tra il loro numero e quello dei core messi a disposizione dal sistema. Si è scelto precisamente questo numero di thread, e non maggiore, perché non essendoci operazioni bloccanti, per esempio di I/O, che potessero bloccare l'esecuzione, non ci sarebbe stato alcun guadagno.

[1] `https://en.wikipedia.org/wiki/Barrier_(computer_science)`

Per ciò che riguarda le soluzioni proposte per gestire la concorrenza e la sincronizzazione del sistema non sono state riscontrate ulteriori problematiche, dunque sono state adottate e implementate quelle proposte nella sezione “Prima soluzione”.

Capitolo 3

Architettura e implementazione

Per la realizzazione del progetto si è deciso di adottare il pattern MVC. All'interno del Model sono state inserite le classi che vanno a descrivere le caratteristiche dei corpi e dello spazio 2D in cui questi si muovono, oltre a ciò, sono presenti altre classi necessari per la corretta esecuzione della simulazione.

Nel Controller sono state inserite le entità attive del sistema, cioè coloro che eseguono effettivamente i calcoli delle forze, e un controller generico che ha lo scopo di preparare l'ambiente all'esecuzione della simulazione e creare i thread che svolgono la maggior parte del lavoro.

Infine, nella View sono stati racchiusi i componenti necessari alla realizzazione dell'interfaccia testuale e la GUI.

3.1 Model

Il Model è il componente dell'applicazione che racchiude le regole dei vari oggetti utilizzati nel sistema.

3.1.1 Fisica dell'ambiente

Si è partiti dal codice fornito a lezione che permetteva già di modellare la fisica della simulazione. Le classi che permettono ciò sono `P2d`, `V2d`, `Body` e `Boundary`.

P2d Modella ipotetiche coordinate x,y posizioni in uno 2D.

V2d Modella vettori in uno spazio 2D e offre alcuni metodi per le operazioni più comuni effettuabili con i vettori.

Body Modella oggetti puntiformi in uno spazio 2D. Fornisce metodi per gestire la velocità, la posizione e calcolo delle relative forze.

Boundary Modella i bordi del mondo della simulazione.

3.1.2 Context

Struttura utilizzata per ottenere informazioni sulla simulazione in corso, possiede dati riguardo la posizione dei confini del mondo, sa se la simulazione è terminata o meno e possiede la lista condivisa che viene aggiornata da `BodyAgent`.

3.1.3 BodiesSharedList

Classe che rappresenta una lista contente dei Body 3.1.1 che deve essere condivisa utilizzata da `Simulator` e `BodyAgent`. Fornisce metodi aggiungere corpi, per aggiornare un corpo in una specifica posizione e resettarla.

Essendo una struttura dati condivisa tra più entità attive è stato necessario gestire la possibilità di eventuali corse critiche.

In particolar modo, per evitare problemi legati alla contemporanea lettura e scrittura di un certo elemento della lista si è deciso di utilizzare due istanze distinte di questa classe: una per leggere i dati relativi ai corpi e un'altra su cui scrivere le modifiche ai dati.

Per quanto riguarda l'accesso contemporaneo in scrittura non ci sono particolari problematiche in quanto ad ogni agente viene affidato un particolare sottogruppo di corpi da gestire, identificato dagli indici della loro posizione nella lista. In questo modo si evita l'accesso contemporaneo alle stesse posizioni della lista.

3.1.4 Configuration

Configuration è una classe utilizzata per leggere i dati necessari alla simulazione da file all'interno del percorso `src/main/resources/`. Una volta acquisite le informazioni la classe li renderà disponibili tramite dei *getter* specifici.

Nel caso si utilizzi la simulazione come file `.jar`, il file dovrà essere presente nella medesima cartella, `Configuration` infatti cercherà il file su questi due possibili percorsi. Nel caso il file non venga trovato, allora verrà lanciata un'eccezione gestita da Simulator.

3.1.5 config.properties

Non rappresenta direttamente una componente del Model ma costituisce comunque un dato fondamentale della simulazione. I boundary forniscono informazioni riguardo le posizioni dei confini del mondo; delta rappresenta la quantità di tempo virtuale aggiunta ad ogni iterazione della simulazione; bodies e incremental informano su quanti corpi e quante iterazioni l'applicativo farà prima di terminare.

Listing 3.1: Esempio di file config.properties

```
upper_boundary=-6
lefter_boundary=-6
righter_boundary=6
lower_boundary=6
delta_incremental_time=0.001
bodies_initial_quantity=100
iterations_initial_quantity=1000
```

3.2 Controller

Qui sono presenti le classi attive dell'applicativo che sfruttano le precedenti classi del Model per gestire il core del sistema.

3.2.1 Simulator

È il motore del sistema e vi è racchiusa la maggior parte della logica. Si occupa di inizializzare gli agenti all'avvio dell'applicazione e ad avviarli alla pressione del tasto "`START`". Al suo interno possiede le due istanze di `BodiesSharedList`, in quanto è suo compito passare i riferimenti agli agenti che crea e occuparsi della loro sincronizzazione. Nello specifico, si è utilizzata una *CyclicBarrier*[1] che permette di interrompere la loro esecuzione fino a che tutti non abbiano raggiunta l'istruzione di *await* sulla barriera. Questa struttura fornisce inoltre un ulteriore funzionalità, ovvero l'uso di un `Runnable` da avviare quando la barriera viene "distrutta" e che viene eseguito prima che i thread ripartano dalla precedente istruzione wait. È in questo `Runnable` che la lista adibita alla lettura dei dati viene aggiornata con quelli nuovi contenuti nella lista usata dagli agenti per memorizzare le nuove informazioni nell'iterazione appena terminata. Oltre

[1]Struttura dati di java.util.concurrent - CyclicBarrier: `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CyclicBarrier.html`

a ciò, viene avvisata anche la view in maniera tale che possa essere aggiornata con le nuove posizioni.

Quando la simulazione viene avviata, i suoi parametri vengono calcolati grazie a `Configuration`, in caso il file per i parametri non fosse presente, `Simulator` usa dei dati predefiniti.

3.2.2 BodyAgent

Rappresenta la parte del sistema adibita ai calcoli e viene istanziato da `Simulator`. È modellato come classe che estende `Thread`[2]. Il suo compito è quello di occuparsi di un sottoinsieme di corpi[3], ovvero calcolare i loro nuovi dati esaminando le distanze da tutti gli altri corpi presenti nella simulazione. Al suo interno utilizza due liste, una che viene modificata inserendoci nuovi corpi con i nuovi dati, e una che viene utilizzata per leggere i dati delle posizioni precedenti[4]. La nuova lista viene aggiornata con nuovi corpi in maniera tale che quando questi vanno ad essere modificati durante l'aggiornamento, non possano generare corse critiche. Una volta terminato il calcolo di tutti i loro corpi, i thread entrano in uno stato di wait fino a che la CyclicBarrier non li sbloccherà. In Figura 3.1 è possibile osservare il loro comportamento e l'azione della barriera.

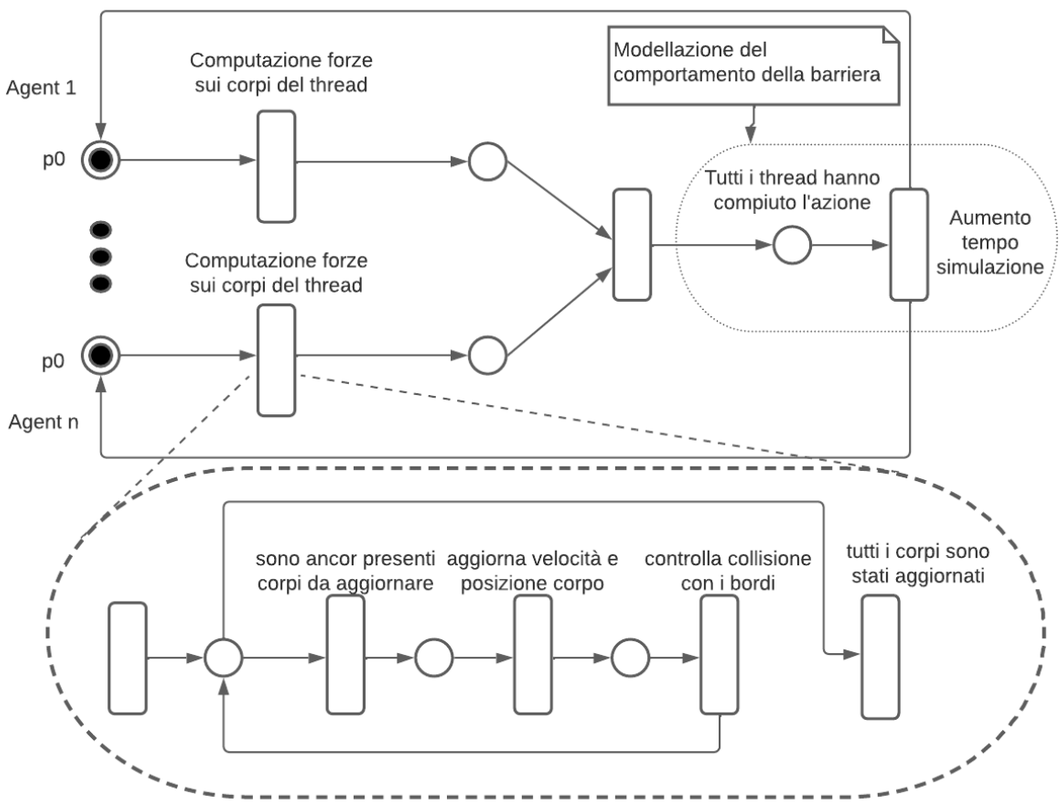


Figura 3.1: Rete di Petri del comportamento dei thread del sistema implementati in `BodyAgent`. Nella parte inferiore dell'immagine, nell'ovale con tratteggi, è possibile osservare il comportamento dei thread durante la computazione delle nuove forze dei corpi.

[2]Java Thread:`https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html`

[3]Un sottoinsieme di corpi perché nella versione scalable, quella definitiva, abbiamo tanti thread quanti core messi a disposizione dalla JVM e non più un thread per ogni corpo.

[4]Per la lettura e la computazione delle nuove forze abbiamo sfruttato la `ComputeTotalForceOnBody` che avevamo visto a lezione.

3.2.3 ViewListener

Classe utilizzata per gestire gli input provenienti dai comandi della GUI, così facendo si è evitato che le componenti della View chiamassero direttamente i metodi del controller migliorando l'indipendenza delle componenti e l'estendibilità del sistema.

3.3 View

Le componenti della View sono molto simili a quelle fornite in precedenza, sono state separate alcune parti e introdotto il codice per fornire le funzionalità richieste. È stata inoltre aggiunta l'interfaccia `View` che fornisce metodi per mostrare dati, attivare e disattivare bottoni.

3.3.1 VisualiserPanel

Questa classe è rimasta invariata rispetto al codice precedente, differisce però dal non essere più una classe innestata ma bensì una classe a sé stante che comunque viene utilizzata nel medesimo modo. Sono stati poi spostati i *keylisteners* in `GUIView`.

3.3.2 GUIView

Rispetto alla classe fornita presenta anche i bottoni "`START`" e "`STOP`" e metodi per modificarli. In essa sono stati inseriti inoltre i *listeners* per effettuare `zoom in` e `zoom out` utilizzabili con le frecce giù e su durante la simulazione che precedentemente si trovavano in `VisuliserPanel`.

Capitolo 4

Verifica e prestazioni

Nel presente capitolo si andranno ad analizzare il processo di verifica del modello implementato e si analizzeranno le prestazioni confrontando le performance di tre modelli:

- *Sequenziale*, il modello in cui non è implementato il multithreading;
- *1:1*, il modello in cui è implementato un mapping uno a uno tra numero di corpi e numero di thread, implementata inizialmente ma poi sostituita;
- *Fixed Thread*, il modello finale in cui il numero di thread è pari al numero di core utilizzabili dall'applicazione.

4.1 Verifica

Al seguito della definizione e implementazione del modello ritenuto migliore, si sono realizzate le verifiche di funzionamento e correttezza attraverso gli applicativi *VisualVM* e *Java PathFinder*.

4.1.1 VisualVM

L'utilizzo di VisualVM si è dimostrato fondamentale per l'analisi del funzionamento dei thread, per la profilazione delle prestazioni e per l'utilizzo della memoria.
Un esempio di utilizzo è riportato in Figura 4.1.

Figura 4.1: Thread in esecuzione con 1000 Body - modello *Fixed Thread*

Listing 4.1: Corsa critica rilevata da JPF

```
gov.nasa.jpf.listener.PreciseRaceDetector
race for field concurrent.model.Context.keepWorking
  BodyAgent0 at concurrent.model.Context.setKeepWorking
" WRITE: putfield concurrent.model.Context.keepWorking
  BodyAgent0 at concurrent.model.Context.isKeepWorking
" READ: getfield concurrent.model.Context.keepWorking
```

Listing 4.2: Istruzioni con Verify API di JPF

```
if (iter >= nSteps || stopFromView) {
    Verify.beginAtomic();
    context.setKeepWorking(false);
    Verify.endAtomic();
}
```

4.1.2 Java Path Finder

Java PathFinder è stato utilizzato per la verifica della correttezza dell'elaborato, in particolare è stato adoperato per la ricerca di *Race Condition*.

Con l'esecuzione di JPF si sono scovate principalmente tre problematiche. Due, analoghe, relative ad una corsa critica sugli oggetti `P2d` e `V2d` che si presentava nel momento in cui un Body aggiornava la propria posizione. Infatti, non avendo creato inizialmente delle copie difensive degli oggetti `P2d` e `V2d`, quando un Body aggiornava la propria posizione nella lista dedicata alla scrittura, essa veniva aggiornata come *side effect* (avendo `P2d` e `V2d` lo stesso riferimento tra le liste) anche nella lista dedicata alla lettura creando un grosso problema di consistenza, non facilmente visibile in fase di debugging.

Il terzo problema scaturito dall'analisi con JPF necessita di una trattazione più approfondita. Riassumendo, si è utilizzata una `CyclicBarrier` come strumento per la sincronizzazione dei `BodyAgent` e il suo secondo parametro passabile al costruttore, un `Runnable`, è usato per modificare delle variabili globali. La modifica di una di queste variabili, `keepWorking`, da parte della barriera porta al rilevamento di una corsa critica come mostrato nel listato 4.1. Come scritto nella documentazione della `CyclicBarrier`, il suo `Runnable` è eseguito atomicamente prima di risvegliare tutti i thread presenti nella barriera e inoltre, per questioni di efficienza, viene eseguito sul thread che "rompe" la barriera[1].

La scrittura della variabile `keepWorking` è possibile solamente mediante il metodo `setKeepWorking` utilizzato nel `Runnable` della barriera, mentre la lettura di `keepWorking` è eseguita da ogni `BodyAgent` all'inizio della sua iterazione per fa sì che ogni agente possa capire se terminare o meno la sua esecuzione.
L'errore di JPF segnala la presenza di due thread, `BodyAgent0`, che accedono concorrentemente alla variabile. Teoricamente ciò non dovrebbe essere possibile poiché la modifica e la lettura della variabile **avvengono** in due momenti strettamente diversi, essendo `setKeepWorking` chiamato solamente dalla barriera e `isKeepWorking` chiamato solamente dai thread, che nel momento in cui avviene la modifica sono in bloccati dalla barriera stessa.
Sono state utilizzate anche le *Verify API* di JPF per verificare il comportamento dell'accesso alla variabile globale oggetto dell'errore, attraverso i costrutti `Verify.beginAtomic()` e `Verify.endAtomic()`, come mostrato nel listato 4.2.

Nello specifico, si è forzata la generazione di uno scenario in cui la chiamata `setKeepWorking(false)` è inevitabilmente atomica. In questo scenario si è constatato che JPF non segnala più la *Race Condition*, dimostrando come il problema non sussista se il thread che esegue il codice della barriera è unico e il solo in esecuzione in

[1] `http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/concurrent/CyclicBarrier.java`

quel preciso momento; condizione teoricamente confermata dalla documentazione della `CyclicBarrier`.
Infatti, riassumendo:

- utilizzando un modello a thread fissi, come visibile in figura 4.1, si ha il pieno controllo sul numero dei thread e sul loro nome, potendo dimostrare in fase di debugging la non presenza di due thread con lo stesso nome e attivi nello stesso momento;
- la `CyclicBarrier` esegue l'istruzione `setKeepWorking(false)` strettamente prima di rilasciare il controllo ai `BodyAgent`;
- i `BodyAgent` eseguono l'istruzione `isKeepWorking()` strettamente dopo che la barriera li abbia rilasciati.

In conclusione, si ritiene che non possa sussistere l'errore della corsa critica ma che esso sia rilevato per una mal interpretazione degli eventi da parte di JPF.

4.2 Prestazioni

L'analisi delle prestazioni è stata eseguita confrontando le performance dei tre modelli implementati ed ognuno di loro è stata testato su un numero variabile di corpi e iterazioni.

Analizzando in dettaglio i risultati riportati nella Tabella 4.1, è possibile calcolare l'incremento della velocità di esecuzione dei vari modelli a confronto, trascurando i risultati ottenuti con il parametro *nBodies* pari a 100 poiché poco significativi nell'analisi della scalabilità dell'applicazione.

Confrontando il modello sequenziale con quello a thread fissi, si può notare come la seconda soluzione sia circa quattro volte più performante in tutti gli scenari.

Per ciò che concerne invece il confronto tra modello sequenziale e modello 1:1, l'analisi è più complessa: con un numero di corpi pari a 1000 (ma analogamente si ha lo stesso comportamento con 100) il modello 1:1 risulta peggiore del modello sequenziale, il quale risulta circa 1.5 volte più veloce. La situazione si inverte con un numero di corpi pari a 5000, ove il modello concorrente risulta tre volte più veloce del sequenziale. In quest'ultimo caso è chiaro come lo scheduler del sistema operativo vada a ripartire i thread *virtuali* della JVM in modo del tutto simile all'approccio a thread fissi[2].

Parametri		Risultati dei modelli in secondi		
N° corpi	N° iterazioni	Sequenziale	1:1	Thread fissi
100	1000	0.083	0.956	0.152
	10000	0.686	8.874	0.805
	50000	2.465	44.897	3.646
1000	1000	7.145	10.572	1.716
	10000	72.094	109.132	16.680
	50000	359.054	527.608	83.070
5000	1000	175.565	58.198	38.800
	10000	1477.274	578.232	458.425
	50000	9271.029	2897.990	1973.932

Tabella 4.1: Tabella riassuntiva delle misurazioni.

Come mostrato nei grafici riportati nelle figure in 4.3 e 4.4 il modello *Fixed Thread* si è dimostrato più scalabile rispetto agli altri due, ottenendo prestazioni nettamente migliori all'aumentare della complessità del problema.

Studiando l'andamento dei tre modelli, si può evincere che:

[2]Understanding Threads `https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/thread_basics.html`

- Il modello sequenziale ha un andamento polinomiale, come visibile in Figura 4.2. Infatti, raddoppiando il numero dei corpi raddoppia anche il numero di istruzioni che vengono eseguite sullo stesso thread, dando come risultato un tempo leggermente superiore al doppio con un andamento compatibile con una funzione polinomiale. Particolarmente interessante è il "gomito" che la curva subisce quando si superano i 1000, indicando che fino a quel valore il modello sequenziale risulta comunque competitivo.

- Il modello 1:1 avrebbe un andamento lineare se ipoteticamente un calcolatore potesse eseguire N thread in parallelo, con N pari al numero dei corpi. Tuttavia, il numero di quest'ultimi solitamente è limitato e di conseguenza al crescere dei thread si ha una diminuzione delle prestazioni.

- Il modello *Fixed Thread* ha infine un andamento lineare in qualsiasi scenario, poiché allocando un numero di thread pari al numero di core disponibili si è sicuri che il comportamento, al crescere dei corpi, sia linearmente crescente, rendendolo di fatto la soluzione migliore al problema analizzato.

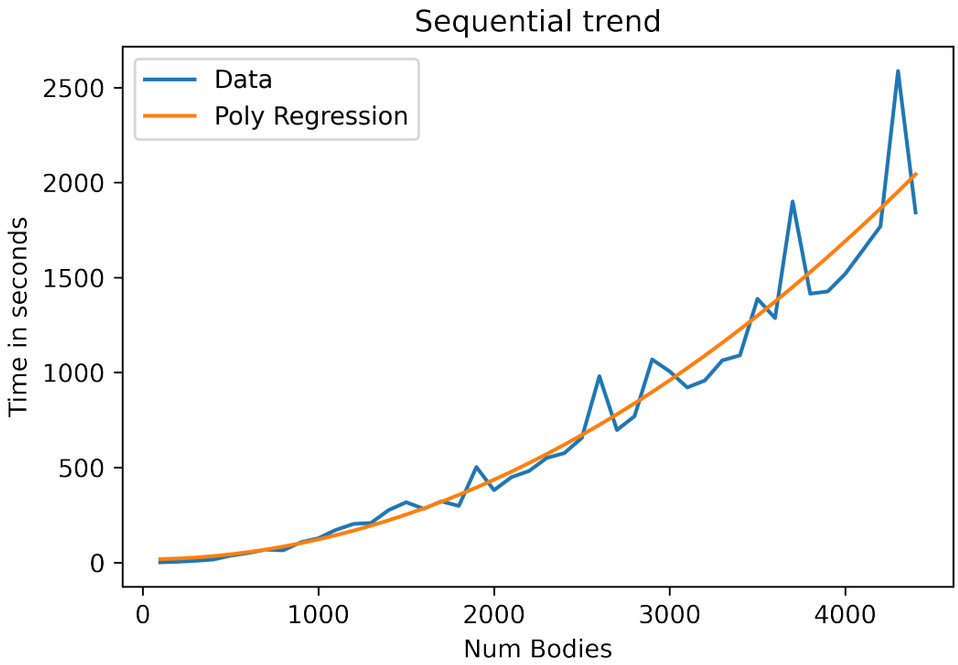
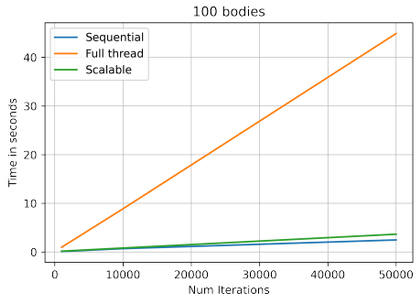
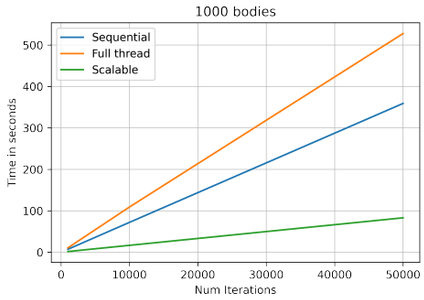


Figura 4.2: Andamento del modello sequenziale

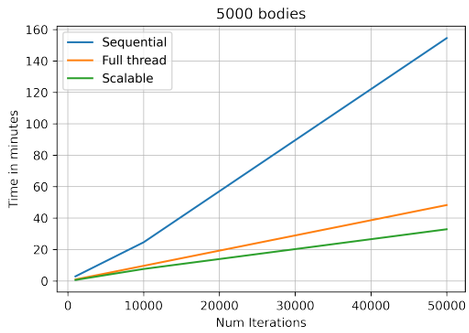
In sintesi, il modello *Fixed Thread* si è rivelato essere il più appropriato per un approccio scalabile e prestazionalmente stabile al crescere della complessità computazionale.



(a)

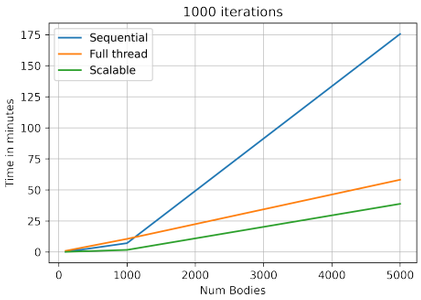


(b)

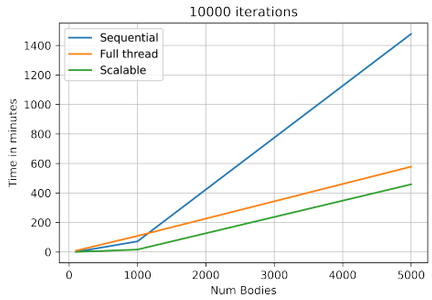


(c)

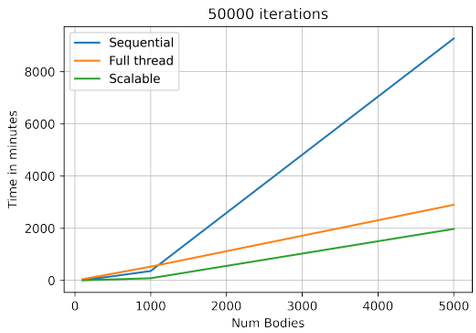
Figura 4.3: Prestazioni al variare delle iterazioni



(a)



(b)



(c)

Figura 4.4: Prestazioni al variare del numero di corpi

Conclusioni

In conclusione, si è soddisfatti della soluzione ideata e implementata in quanto si è riusciti ad ottenere uno *speed up* del sistema più che apprezzabile.

Una possibile miglioria da implementare in futuro potrebbe essere la realizzazione di listener più generici in modo da gestire più tipologie di comandi dall'interfaccia grafica, come ad esempio una pausa del sistema.

Inoltre, la view realizzata per poter stampare a video le varie iterazioni della simulazione potrebbe essere espansa in modo da poter simulare il comportamento dei bottoni di Start e Stop della GUI tramite linea di comando, attualmente non implementati poiché non facenti parte dei requisiti.

Infine, lo sviluppo dell'elaborato si è rivelato molto utile per approfondire la concorrenza, la verifica della correttezza attraverso l'applicativo *Java Path Finder* e l'uso di strumenti specifici per l'analisi di programmi concorrenti come *VisualVM*.