Relazione Assignment-01

Corso di Programmazione Concorrente e Distribuita

Università di Bologna

Acampora Andrea

Accursi Giacomo

Indice

Capitolo 1

STEP 1

1.1 Analisi del problema

Il progetto consiste nel realizzare una versione concorrente di un simulatore implementato in modo sequenziale. Nello specifico, il programma simula il movimento di N corpi su un piano bidimensionale, soggetti a due tipi di forze: una forza repulsiva ed una forza di attrito. L'algoritmo del simulatore, ad ogni step della simulazione e per ogni corpo presente:

1. calcola la forza a cui è soggetto che dipende dalla posizione degli altri corpi e dalla frizione
2. calcola l'accelerazione istantanea, data la forza totale e la massa del corpo
3. aggiorna il vettore velocità del corpo, data l'accelerazione ed il tempo trascorso
4. aggiorna la posizione del corpo, data la velocità ed il tempo e
5. controlla le collisioni con i *boundary* della simulazione.

L'algoritmo del simulatore impone alcuni limiti che è obbligatorio rispettare, tra i quali:

- Il calcolo delle forze al tempo t avviene considerando coerentemente le posizioni dei corpi al tempo t
- L'aggiornamento delle posizioni può avvenire solo dopo che tutte le forze sono state calcolate (e le velocità aggiornate).

1.2 Descrizione della strategia risolutiva

Gli step della simulazione vengono eseguiti obbligatoriamente in modo sequenziale in quanto è necessario che tutti i corpi abbiano aggiornato la propria posizione prima di calcolare le forze di repulsione e di attrito. A fronte di ciò, dovrà essere resa parallela solo la computazione interna di ogni step della simulazione. Analizzando l'algoritmo del simulatore si può notare che ad ogni step vengono iterati tutti i corpi presenti e per ciascuno è necessario eseguire operazioni in lettura sulle posizioni di tutti gli altri corpi, ad esempio per calcolare le forze che agiscono su esso. Tutte queste operazioni, però, possono essere svolte in modo parallelo senza meccanismi di sincronizzazione in quanto nella prima fase la posizione di ogni corpo rimane invariata. Successivamente, invece, sarà obbligatorio sincronizzare gli agenti nella fase inerente all'aggiornamento delle posizioni dei corpi. Nella sezione successiva verrà analizzata con maggior dettaglio l'implementazione di questa strategia.

1.3 Descrizione dell'architettura proposta

Per quanto riguarda l'architettura dell'applicazione è stato scelto di utilizzare il design pattern **Model-View-Controller**. Sebbene il primo step non richiedesse l'utilizzo di un'interfaccia grafica, il pattern ha permesso di migliorare notevolmente l'organizzazione del codice ed inoltre è stato fondamentale per poter poi aggiungere la GUI nello step 2 senza alcuna difficoltà. Il *Model* dell'applicazione comprende i *Body*, i *Boundary* ed una classe `SimulationState` contenente tutte le informazioni sullo stato della simulazione (numero di corpi, numero iterazioni eseguite, tempo trascorso). Il *Controller*, invece, è rappresentato dall'interfaccia `Controller`, la quale ha il compito di iniziare la simulazione. La *View* nello step uno è stata utilizzata solo per effettuare log su console.

Per quanto riguarda la trasformazione del programma sequenziale in un programma concorrente, la prima fase effettuata è stata la scelta dell'architettura concorrente da adottare. In particolare si è scelto di utilizzare l'architettura **Master-Workers**.
Il *Master Agent* ha il compito di:

- istanziare i *Worker Agents*, in particolare un numero di workers pari ai processori disponibili al sistema più uno.
- suddividere la computazione di ogni step della simulazione in sotto tasks.

- assegnare i task ai workers tramite una *Bag of Tasks* e utilizzare appropriati meccanismi di sincronizzazione.

I *Worker Agents*, invece, hanno il compito di ottenere i task dal *Master Agent* ed eseguirli. Si è scelto di trattare i *Worker Agents* come dei *Generalists*, ossia essi non sanno a priori qual è il compito che andranno ad eseguire in quanto la logica è incapsulata all'interno del task. La seconda fase di design comprende l'identificazione dei task presenti e l'analisi delle dipendenze tra essi. In particolare, si è scelto di utilizzare **Task Decomposition Pattern**: la computazione del programma che si intende parallelizzare è stata decomposta in un insieme di task. A seguito di ciò sono emersi tre task principali:

- **ComputeForcesTask**: il calcolo delle forze che agiscono su un corpo e l'aggiornamento del vettore velocità a fronte delle forze calcolate è indipendente rispetto a tutti gli altri corpi presenti.
- **UpdatePositionTask**: l'aggiornamento della posizione di un corpo sulla base della velocità precedentemente calcolata è indipendente rispetto a tutti gli altri corpi presenti.
- **CheckCollisionTask**: il controllo delle collisioni di un corpo con i *Boundary* della simulazione è indipendente rispetto a tutti gli altri corpi presenti.

Come anticipato nella descrizione della strategia risolutiva, la dipendenza tra i vari task è esclusivamente di tipo temporale: i task per l'aggiornamento della posizione possono essere eseguiti solo al termine dei task sulla computazione delle forze ed i task sul controllo delle collisioni solo al termine dei task sull'aggiornamento delle posizioni.
Successivamente, analizzando il secondo ed il terzo task è stato riscontrato che questo vincolo temporale non è necessario in quanto il controllo delle collisioni per un corpo può essere effettuato immediatamente in seguito all'aggiornamento della posizione. A fronte di ciò si è scelto di utilizzare solo i primi due task: **ComputeForcesTask** e **UpdatePositionTask** e di incorporare il controllo delle collisioni nel secondo task. Questa dipendenza temporale tra i due task è stata risolta utilizzando un *Latch* condiviso da *Master Agent* e *Worker Agents*.
Una volta identificati i task è stato necessario comprendere come decomporre i corpi della simulazione a fronte dei task presenti. Sono emerse due possibili strategie:

1. Nella **prima strategia** ogni task si occupa di aggiornare un solo *Body*. In questo modo ad ogni step della simulazione il *Master Agent* dovrà creare un numero di task

pari al numero di Body presenti sia per aggiornare la velocità che successivamente per aggiornare la posizione. I *Worker Agents* continuamente dovranno estrarre un task dalla Bag, eseguirlo e procedere con un nuovo task fino a che non saranno terminati.

2. Nella **seconda strategia** ogni task si occupa dell'aggiornamento di una lista di *Body*. In questo caso ad ogni step della simulazione il *Master Agent* dovrà creare un numero di task pari al numero di *Worker Agents* ed assegnare ad ogni task una sotto porzione dei corpi presenti. I *Worker Agents* in questo caso estraggono un solo task dalla *Bag*, lo eseguono e si mettono in attesa del prossimo task.

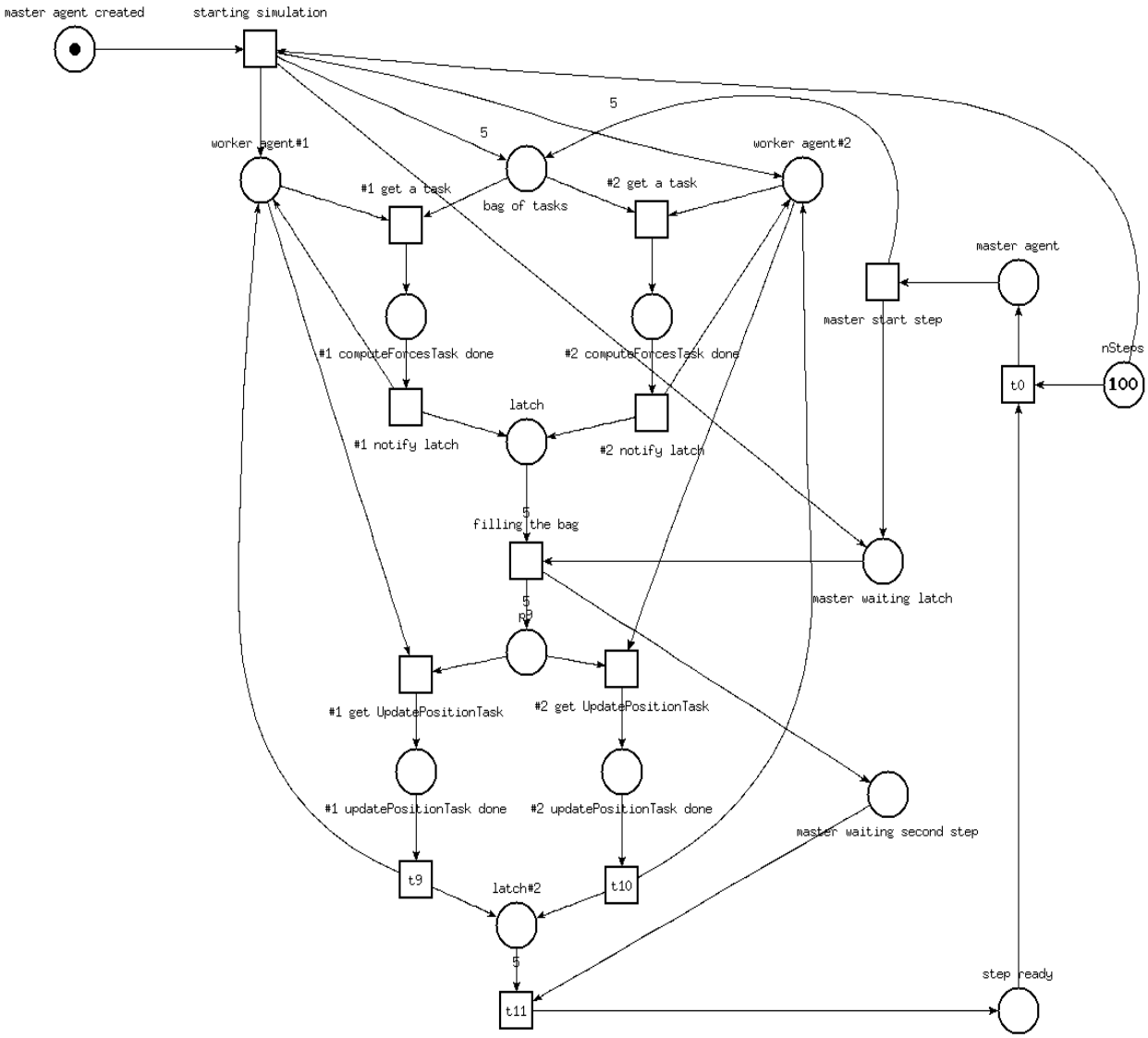
Tra i vantaggi della prima strategia (un solo corpo per task) c'è sicuramente il fatto di non avere situazioni in cui alcuni *Worker Agents* hanno terminato l'esecuzione dei task sulla propria lista di corpi prima di altri e si mettono quindi in attesa mentre c'è ancora della potenziale computazione da svolgere. D'altro canto però, a differenza della seconda strategia, si avrà un numero di accessi alla Bag of Task (implementata come monitor) molto più elevato. Nella sezione relativa alle performance verranno analizzate entrambe le strategie. In tutte le altra sezioni si assume utilizzata la prima strategia.

1.4 Comportamento del sistema

All'avvio dell'applicazione:

- Viene eseguito il *Master Agent*, a cui vengono passati la *View* ed il *Model*.
- Il *Master Agent* crea ed esegue i *Worker Agents* ed inizia il ciclo degli step della simulazione.
- Ad ogni step aggiunge i *ComputeForcesTasks* alla Bag e si mette in attesa su un *CompletionLatch* che i workers abbiano terminato.
- I *Worker Agents* richiedono continuamente tasks dalla Bag e li eseguono. Terminati i task viene restituito il controllo al *Master Agent* che procederà con la seconda fase.
- Una volta terminata la prima fase il *Master Agent* resetta il Latch, aggiunge alla Bag gli *UpdatePositionTasks* e si mette in attesa nuovamente sul latch.
- Una volta terminata anche la seconda fase incrementa gli step effettuati, aggiorna il tempo virtuale e procede con un nuovo step.

Il comportamento del sistema può essere rappresentato con la seguente rete di Petri. Per semplificare la rete viene mostrata una simulazione contenente 100 steps.


master agent created
starting simulation
worker agent#1
5
5
worker agent#2
bag of tasks
#1 get a task
#2 get a task
master agent
master start step
#1 computeForcesTask done
#2 computeForcesTask done
nSteps
100
t0
latch
#1 notify latch
#2 notify latch
5
filling the bag
master waiting latch
5
#1 get UpdatePositionTask
#2 get UpdatePositionTask
#1 updatePositionTask done
#2 updatePositionTask done
master waiting second step
t9
latch#2
t10
5
t11
step ready

1.5 Analisi delle performance

Il sistema è stato testato su un pc portatile con le seguenti caratteristiche:

- Processore Intel Core i5-8250U
- RAM da 8 GB DDR4

Di seguito sono riportati i risultati dei test effettuati nella versione sequenziale e nelle due versioni concorrenti di cui si è parlato nei paragrafi precedenti. Tutte le simulazioni sono state effettuate con batteria al 100% e alimentatore collegato per evitare che il sistema operativo mettesse in atto politiche di risparmio energetico. I tempi sono espressi in secondi.

	1000 steps	10000 steps	50000 steps
100 bodies	0.10	0.62	2.81
1000 bodies	5.61	55.33	276.63
5000 bodies	148.76	1684.25	8421.63

Tabella 1.1: Sequential

	1000 steps	10000 steps	50000 steps
100 bodies	0.21	1.32	6.02
1000 bodies	2.58	25.60	137.50
5000 bodies	46.52	620.49	3116.92

Tabella 1.2: Concurrent One Body Per Task

	1000 steps	10000 steps	50000 steps
100 bodies	0.134	0.654	2.776
1000 bodies	2.05	21.27	113.34
5000 bodies	48.03	626.02	3187.81

Tabella 1.3: Concurrent SubList Per Task

Le due versioni concorrenti sono state testate anche utilizzando il doppio dei core disponibili, come ci si aspettava però questo non ha portato nessun successo.

	1000 steps, 9 worker	1000 steps 4 worker
100 body	0.45	0.57
1000 body	2.17	2.30
5000 body	3.20	2.72

Tabella 1.4: SpeedUp comparing sequential and Concurrent One Body Per Task

	1000 steps, 9 worker	1000 steps 4 worker
100 body	0.06	0.14
1000 body	0.27	0.58
5000 body	0.40	0.68

Tabella 1.5: SpeedUp comparing sequential and Concurrent One Body Per Task

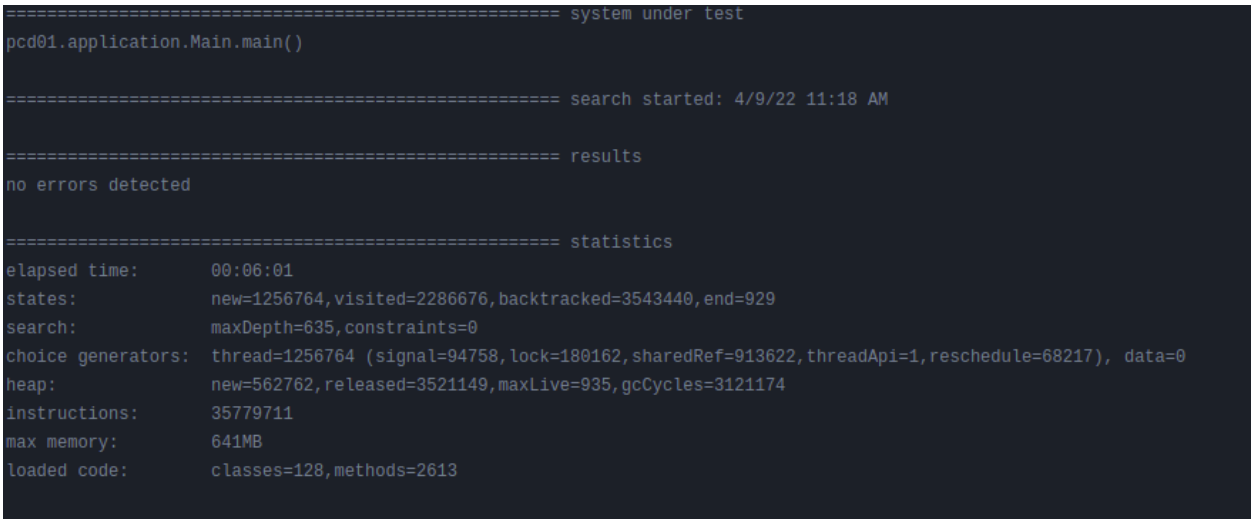
Come si può notare diminuendo il numero di core, all'aumentare del numero di corpi della simulazione, lo speed up tende a diminuire. Si guadagna invece in termini di efficienza. E' stato riportato lo speed up solo per le simulazioni con mille corpi poichè, come ci aspettava, il tempo impiegato aumenta in maniera lineare all'aumentare degli step.

1.6 Verifica e controllo delle proprietà di correttezza

1.6.1 Java Path Finder

La correttezza del programma è stata testata con il tool *Java Path Finder*. Nello specifico, il tool è stato lanciato con una versione semplificata dell'applicazione:

- si considera solo la parte inerente allo step 1 ossia senza GUI.
- vengono resi atomici alcuni blocchi di codice per ridurre lo spazio dei possibili stati e per diminuire il tempo di esecuzione del programma.
- sono state utilizzate le API di jpf per testare il programma con un numero variabile di corpi ed un numero variabile di *Worker Agents*.



Per lanciare il programma è sufficiente spostarsi nel branch `jpf` e lanciare il comando:

```
./gradlew jpfVerify -Pfile="src/main/application/jpf/Main.jpf"
```

1.6.2 TLA+

È stata effettuata una verifica del programma anche con il tool *TLA+*. Anche in questo caso è stato realizzato un algoritmo rappresentate una versione semplificata dell'applicazione. Nello specifico è stata catturata l'interazione tra il *Master Agent* ed i *Worker Agents*. Il file si può trovare sotto la directory *TLA* presente all'interno della cartella del progetto.

Capitolo 2

STEP 2

2.1 Analisi del problema

Il secondo step richiede di estendere la simulazione includendo una GUI con pulsanti start/-stop per lanciare/fermare la simulazione e visualizzare l'andamento, includendo informazioni circa il tempo virtuale. La GUI si presuppone visualizzi stati consistenti della simulazione. A differenza dello step 1, al momento della creazione, il *Master Agent* non potrà partire immeditamente ma dovrà aspettare che l'utente prema il bottone START. Durante la simulazione la computazione dovrà essere interrotta in maniera reattiva a fronte della pressione del bottone STOP.

2.2 Descrizione della strategia risolutiva

Per visualizzare i corpi della simulazione sarà necessario aggiornare la GUI in modo ciclico. Alla fine della computazione di ciascuno step, sarà compito del *Master Agent* comunicare alla View di aggiornarsi, fornendo ad essa i dati dei corpi. Al momento della creazione, il controller si registra come listener della view, la quale notificherà tramite i metodi `start()` e `stop()` la pressione dei rispettivi bottoni. L'inizio della simulazione è stato gestito tramite la classe monitor `StartSynch`. All'avvio del programma il *Master Agent* si metterà in attesa che la view comunichi al controller la pressione del bottone start. Per quanto riguarda la possibilità di mettere in pausa la simulazione, viene utilizzata la classe monitor `StopFlag`. Il

controllo sullo stato della flag è effettuato sia dal *Master Agent* ad ogni step della simulazione, che dai *Worker Agents* prima di effettuare la computazione di ogni task. Controllando lo stato della flag prima dell'esecuzione di ogni task, risulta più reattiva l'implementazione nella quale ogni task modifica un singolo body. Tuttavia la stessa reattività sarebbe stata possibile delegando il controllo della flag al task prima di aggiornare ogni body e non al *Worker Agent*. Per riprendere l'esecuzione della simulazione viene riutilizzato il meccanismo della classe *StartSynch*. La GUI deve mostrare stati consistenti della simulazione. A fronte di ciò è possibile utilizzare il metodo `invokeAndWait` della libreria *Swing*, il quale obbliga il *Master Agent* ad attendere il completamento della visualizzazione prima di riprendere la computazione. L'altra possibilità è utilizzare il metodo `invokeLater`, il quale restituisce immediatamente il controllo al chiamante, senza attendere il completamento della visualizzazione. In un primo momento è stato utilizzato `invokeAndWait`, ma, dopo aver notato che quest'ultimo diminuiva drasticamente le performance nell'aggiornamento della GUI, è stato utilizzato il metodo `invokeLater`. Alla View viene passata una copia difensiva della lista dei body aggiornata per evitare problemi di inconsistenza. Passando la lista come riferimento *Master Agent* inizierebbe ad aggiornare la lista dei corpi prima che la View termini la visualizzazione. E' stata aggiunta anche la possibilità di settare la frequenza di aggiornamento della GUI in termini di Frame Per Secondo.