Assignement01

Marco Raggini
Matricola:0001141793
Email: marco.raggini2@studio.unibo.it

Francesco Carlucci
Matricola:0001136938
Email: francesco.carlucci6@studio.unibo.it

8 aprile 2024

Indice

Capitolo 1

Analisi del problema

Il progetto è costituito da simulazioni che riguardano il comportamento di macchine che percorrono strade, contenenti anche semafori. In particolare i problemi riscontrati per rendere la simulazione concorrente, partendo da una simulazione sequenziale, sono:

- **La comunicazione tra le macchine e l'environment:** uno dei problemi principali è stato capire come far comunicare le macchine con l'ambiente. Poiché tutte le macchine devono decidere quale azione intraprendere in base alla situazione dell'ambiente senza modificarlo, dopodiché ciascuna può eseguire la propria azione. Al termine di tutte le azioni si ripete questo procedimento finché la simulazione non termina.
- **Il numero dei thread:** l'idea iniziale consisteva nel rendere ogni macchina un thread. Successivamente, si è capito che i thread fisici non bastavano per grandi quantità di macchine, questo ha reso l'effettiva implementazione più impegnativa.
- **La GUI:** L'aggiunta dei bottoni deve permettere lo stop e la ripartenza della simulazione. Questo può creare problemi di concorrenza poiché più thread possono accedere alla stessa informazione condivisa.

Capitolo 2

Strategia risolutiva e architettura

Per risolvere i problemi precedentemente identificati è stata adottata la strategia risolutiva di tipo **specialist parallelism** che consiste nell'assegnare a ciascun agente uno specifico compito in cui il parallelismo è dato da tutti gli agenti che sono attivi simultaneamente.

2.0.1 Gestione dei thread

L'implementazione proposta presenta tre tipi di thread differenti:

- **Thread manager:** si occupa di gestire il resto dei thread del sistema, nello specifico: li genera, li avvia e ne controlla l'esecuzione;
- **Thread per le macchine:** si occupa di gestire i CarAgent ad esso assegnati, infatti ogni AgentThread controlla più macchine nel caso in cui fossero più del numero di thread logici del processore su cui viene eseguito il programma;
- **Thread per i semafori:** si occupa di gestire i semafori presenti sulle strade;

Inizialmente ogni thread gestiva un solo agente (macchina o semaforo), ma ci si è preso resi conto che non potendo sfruttare i thread virtuali questa decisione non era corretta, quindi si è optato per una soluzione in cui in base al numero di thread disponibili nel processore su cui viene eseguito il programma e agli agenti da dover gestire, si è assegnato a ciascun thread un pool di agenti pari al rapporto tra gli agenti e i thread disponibili.

2.0.2 Gestione della concorrenza

Per gestire la concorrenza tra i thread sono state utilizzate barrier e monitor. Le prime servono ad assicurarsi che per iniziare a svolgere le proprie funzioni ciascun thread deve aspettare attraverso una `wait()` che tutti vengano creati e solo a quel punto vengono svegliati attraverso una `notifyAll()` dall'ultimo thread creato. Successivamente, tramite un'altra barrier, i thread aspettano che tutte le macchine abbiano svolto le proprie fasi di `sense` e `decide` (percezione dell'ambiente e decisione dell'azione da svolgere), in questo modo le macchine possono avere la stessa percezione dell'ambiente, senza influenzarlo. A questo punto le varie macchine possono eseguire l'azione decisa in precedenza (fase `act`) e quando tutte l'hanno terminata, controllato anch'esso da una barrier, finisce uno step di comportamento degli agenti.

Per quanto riguarda i monitor, essi sono stati implementati per gestire i comandi mandati dall'utente tramite la GUI. In particolare, abbiamo utilizzato il modello readers/writers per gestire la pausa e la ripresa della simulazione, infatti viene utilizzata una variabile condivisa che viene modificata solo dalla GUI (writer), ma che viene letta da tutti gli agenti (readers).

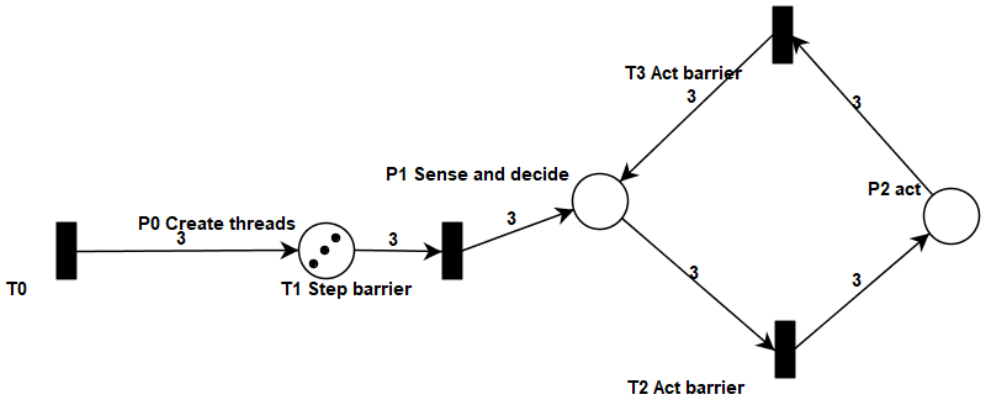

Figura 2.1: Descrizione del comportamento del sistema con Reti di Petri

Capitolo 3

Performance e correttezza

Il tempo ottenuto dall'esecuzione dalla simulazione sequenziale era di 38 secondi, mentre quello con la simulazione concorrente è di 10 secondi, quindi tramite l'implementazione concorrente si è ottenuto un tempo 4 volte inferiore.

```
C:\Users\39392\.gradle\jdks\eclipse_adoptium-21-amd64-windows\jdk-21.0.2
[ SIMULATION ] Running the simulation: 5000 cars, for 100 steps ...
[ SIMULATION ] Completed in 38139 ms - average time per step: 381 ms

Process finished with exit code 0
```

Figura 3.1: Risultati ottenuti con la simulazione sequenziale

```
[ SIMULATION ] Running the simulation: 5000 cars, for 100 steps ...
Finish: 100
Completed in 9647ms
```

Figura 3.2: Risultati ottenuti con la simulazione concorrente

Per quanto riguarda la correttezza abbiamo provato ad utilizzare JPF, ma dopo tutte le conversioni del codice a Java 8 abbiamo riscontrato un problema che non riuscivamo a comprendere, di seguito riportato.

Nonostante ciò, tutte le simulazioni fornite originariamente vengono eseguite in maniera corretta.

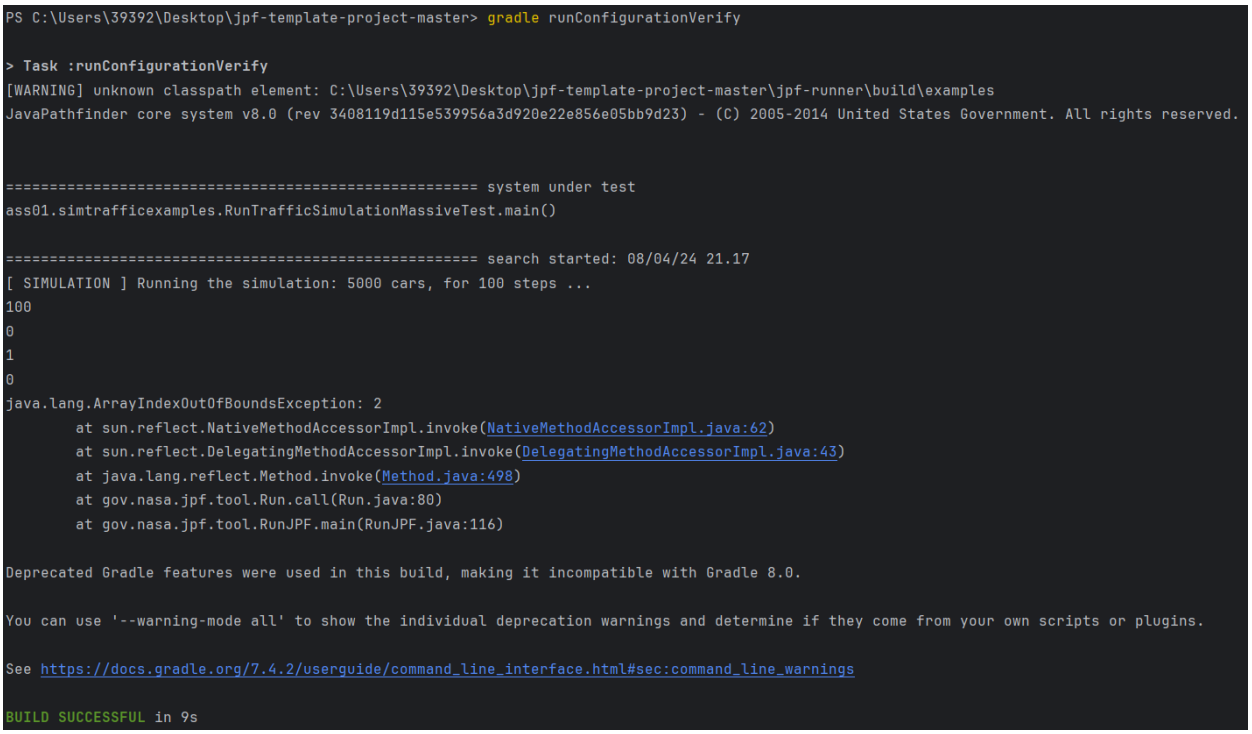

Figura 3.3: Errore di JPF