Alma Mater Studiorum · Università di Bologna

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA E SCIENZE INFORMATICHE

campus di Cesena

Relazione Assignment-1 PCD

Brini Tommaso
Matricola 0001125134

Rambaldi Riccardo
Matricola 0001125022

Anno Accademico 2023-2024

Indice

Capitolo 1

Analisi del problema

L'obiettivo di questo assignment è realizzare un sistema che esegua simulazioni agent-based, al fine di massimizzare le prestazioni di esecuzione. La simulazione consiste nel determinare, a partire da un tempo logico iniziale t0, come evolve il sistema a passi discreti di tempo dt, aggiornando a ogni step lo stato dell'ambiente e degli agenti. Nello specifico, l'esempio fornito riguarda la simulazione del traffico a partire dall'istante 0, con un numero fissato di veicoli (agenti) e di step.

A partire dal codice fornito (che implementa la simulazione in modo sequenziale) il nostro obiettivo è stato sfruttare la concorrenza per massimizzare le prestazioni, minimizzando il tempo di esecuzione. Lo step di ogni agente, che ne definisce il comportamento, è suddiviso in tre parti:

- **SENSE:** l'agente determina le percezioni correnti dall'ambiente.
- **DECIDE:** l'agente determina quale sia la prossima azione da fare, dato lo stato corrente e le percezioni rilevate dalla fase precedente.
- **ACT:** l'agente esegue sull'ambiente l'azione decisa nella fase precedente.

Nella soluzione sequenziale fornita come punto di partenza, l'esecuzione della simulazione impiega molti secondi in quanto ogni agente svolge sequenzialmente il proprio step interno, poi l'environment svolge lo step dell'ambiente, e cosi via in loop fino a raggiungere il numero di step selezionato.

L'aspetto principale da tenere conto è che ogni agente esegue il proprio step (quindi riceve percezioni, decide la prossima mossa e successivamente la esegue) a partire dallo stato dell'ambiente, modificato al passo precedente. Non è quindi importante l'ordine di esecuzione degli agenti in uno specifico

step, in quanto ognuna di queste esecuzioni è condizionata dal precedente step dell'environment (e non da quello attuale).

Per questo motivo, dopo uno step dell'ambiente, ogni agente deve compiere il proprio step indipendentemente dall'ordine di esecuzione. L'aspetto fondamentale è la sincronizzazione tra l'ambiente e i vari agenti: ogni volta che l'ambiente esegue uno step, prima di eseguire il successivo deve attendere che tutti gli agenti abbiano terminato il loro step.

Capitolo 2

Strategia Risolutiva

Inizialmente abbiamo pensato a creare ogni veicolo (agente) come thread, permettendogli di eseguire la parte di SENSE e DECIDE in modo parallelo rispetto alle altre macchine. La fase di ACT, invece, doveva obbligatoriamente essere eseguita in modo concorrente, in quanto esegue l'azione sull'ambiente andando a cambiare (potenzialmente) il suo stato al passo successivo.

Tuttavia, questa soluzione non ci ha convinto a livello di performance, in quanto nelle simulazioni massive (quindi con un numero molto elevate di veicoli) venivano mandati in esecuzioni migliaia di thread che andavano a sovraccaricare molto i core della CPU.

La nostra soluzione proposta per risolvere questo problema è di dividere gli agenti in sottogruppi di ugual numero e far sì che questi si sincronizzino tra loro.

Sfruttando infatti il numero n di processori della macchina su cui eseguiamo il codice, possiamo lanciare n thread (worker) che si occuperanno di gestire l'esecuzione degli step di un sottoinsieme della lista di agenti totali.

Per permettere la sincronizzazione tra i vari worker, abbiamo introdotto un monitor barriera in modo che, ad ogni step della simulazione, venga eseguito prima lo step dell'ambiente e poi lo step di ogni agente.

L'esecuzione della simulazione è gestita da un thread principale, che si occupa di eseguire il primo step dell'ambiente e successivamente di lanciare gli n worker (thread). Ogni worker è sincronizzato sulla barriera condivisa. Dopo aver eseguito lo step per ogni agente assegnato, il worker si mette in wait sulla barriera. Quando l'ultimo worker termina l'esecuzione del proprio step, la barriera esegue il successivo step dell'ambiente tramite un oggetto Runnable (passato nel costruttore) e sveglia tutti i worker permettendo il prossimo step degli agenti.

Testando la nostra soluzione tramite la simulazione "massive", ovvero con 5000 veicoli e 100 step, possiamo notare come i tempi di esecuzione sono stati notevolmente ridotti.

```
> Task :app:RunTrafficSimulationMassiveTest.main()
[ SIMULATION ] Running the simulation: 5000 cars, for 100 steps ...
[ SIMULATION ] Completed in 38518 ms - average time per step: 385 ms
```

Figura 2.1: Risultato della simulazione sequenziale

```
> Task :app:RunTrafficSimulationMassiveTest.main()
[ SIMULATION ] Running the simulation: 5000 cars, for 100 steps ...
[ SIMULATION ] Completed in 8838 ms - average time per step: 0 ms
```

Figura 2.2: Risultato della simulazione concorrente

Abbiamo realizzato una versione della GUI a partire da quella fornita nell'esempio. Quando viene lanciata la gui, la simulazione viene settata con un un numero di agent definito ma i worker thread si mettono in attesa dello start. Tramite un semplice slider è possibile selezionare il numero di step da eseguire. Abbiamo implementato i seguenti bottoni:

- **Start:** serve per lanciare il SimulationThread che si occupa di fare partire la simulazione senza essere bloccante sulla gui. Può essere premuto una volta sola, a inizio simulazione.
- **Stop:** termina definitivamente la simulazione lanciando un interrupt su tutti i thread.
- **Pause/Resume:** ferma/riprende la simulazione all'istante t eseguendo una wait/signal su tutti i worker.

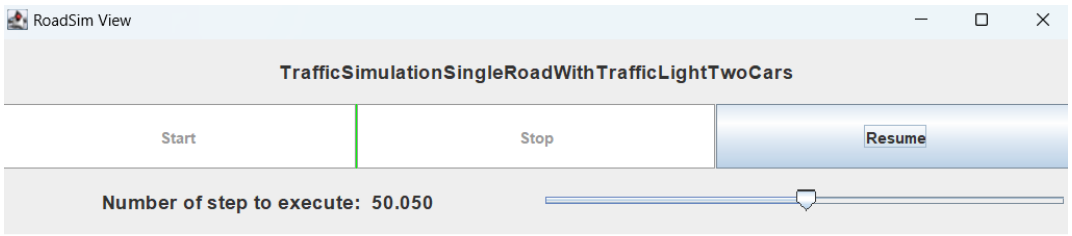

Figura 2.3: Screenshot della GUI sviluppata

Capitolo 3

Verifica con Java Path Finder

Abbiamo testato, utilizzando Java Path Finder, il comportamento dei vari worker tramite una versione semplificata della soluzione adottata, di cui riportiamo i ritultati e la parte rilevante del codice.

```
====================================================== search started: 07/04/24 16.50

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

====================================================== statistics
elapsed time:       00:00:28
states:             new=12700,visited=6212,backtracked=18912,end=2
search:             maxDepth=6213,constraints=0
choice generators:  thread=12700 (signal=0,lock=11,sharedRef=5201,threadApi=6213,reschedule=1275), data=0
heap:               new=52291,released=31857,maxLive=606,gcCycles=18901
instructions:       200089081
max memory:         512MB
loaded code:        classes=94,methods=2126

====================================================== search finished: 07/04/24 16.51
```

Figura 3.1: Output ottenuto da Java Path Finder

```java
public class CyclicBarrier {
    3 usages
    private final int nPartecipants;
    6 usages
    private int counter = 0;
    2 usages
    private Runnable action;
    4 usages
    private boolean isPass = false;

    1 usage
    public CyclicBarrier(int nPartecipants, Runnable action) {...}

    1 usage
    public synchronized void await() throws InterruptedException{
        while(isPass){
            wait();
        }
        counter++;
        if(counter == nPartecipants){
            isPass = true;
            action.run();
            counter = 0;
            notifyAll();
        }
        while(!isPass){
            wait();
        }
        counter++;
        if(counter == nPartecipants){
            reset();
        }
    }

    1 usage
    private void reset() {
        isPass = false;
        counter = 0;
        notifyAll();
    }
}
```

Figura 3.2: Versione della barriera utilizzata

```java
2 usages
public class Worker extends Thread {
    2 usages
    private int step;
    2 usages
    private int id;
    2 usages
    private List<Agent> agents;
    2 usages
    private CyclicBarrier barrier;

    1 usage
    public Worker(int id, int step, List<Agent> agents, CyclicBarrier barrier) {
        this.step = step;
        this.id = id;
        this.agents = agents;
        this.barrier = barrier;
    }

    public void run() {
        for (int i = 0; i < step; i++) {
            for (var agent : agents) {
                agent.step(id);
            }
            try {
                barrier.await();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }

}
```

Figura 3.3: Codice del nostro worker thread

Capitolo 4

Descrizione con Rete di Petri

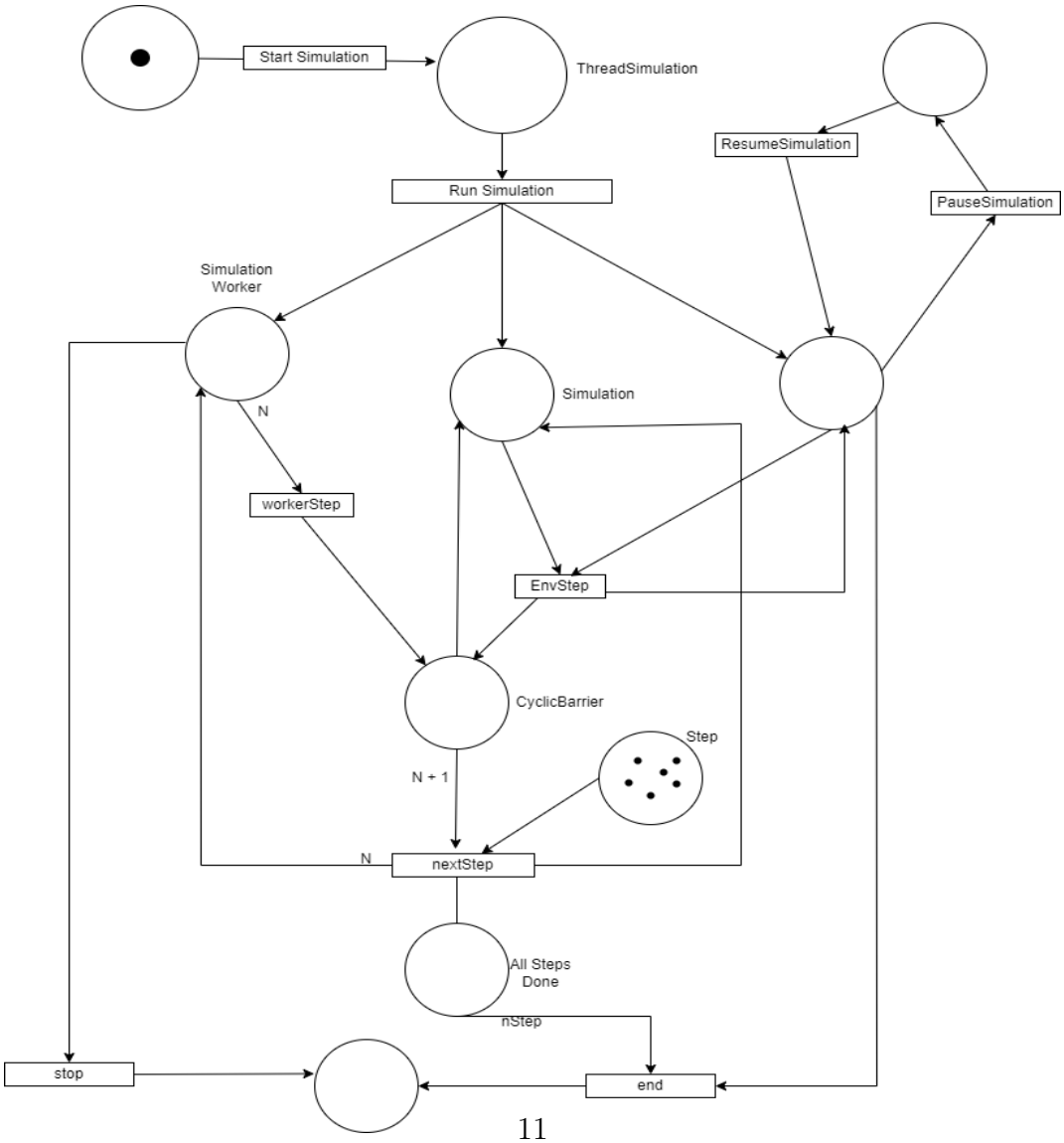

Figura 4.1: Rete di Petri