PCD ASS01

Federico Bravetti
federico.bravetti2@studio.unibo.it

Tommaso Patriti
tommaso.patriti@studio.unibo.it

Alex Testa
alex.testa@studio.unibo.it

I. Analisi del Problema

Il progetto in questione riguarda la simulazione del comportamento di vari agenti all'interno di uno specifico ambiente. In particolare si focalizzerà sulla simulazione e analisi di diverse dinamiche di traffico.

A. Obiettivo

Il progetto si pone come obiettivo quello di sviluppare un simulatore concorrente a partire da una versione sequenziale presa in esempio, al fine di ottimizzare l'utilizzo dell'hardware disponibile e velocizzare l'esecuzione.

B. Concorrenza

La simulazione si compone di un certo numero di iterazioni e in ciascuna di queste, in seguito al cambiamento di stato dell'ambiente, i diversi agenti compiono un passo.

Il passo di un agente può essere rappresentato attraverso tre fasi:

- **Raccolta Dati (*sense*)**: Vengono recuperate le informazioni riguardanti gli agenti vicini. Questa fase è fondamentale per comprendere la situazione corrente.
- **Decisione e Pianificazione delle Azioni da Compiere (*decide*)**: Sulla base delle informazioni raccolte l'agente seleziona il comportamento da seguire (modifica la velocità corrente) e pianifica l'azione da compiere (definisce lo spostamento de effettuare).
- **Verifica ed Esecuzione delle Azioni (*act*)**: Dopo essere stata validata dall'ambiente viene eseguita l'azione proposta dall'agente. Il controllo da parte dell'ambiente serve ad evitare che si raggiunga uno stato illegale della simulazione (e.g. sovrapposizione degli agenti).

Sarebbe ideale parallelizzare l'esecuzione dei passi dei vari agenti, ma questo porterebbe ad errori nella simulazione. Quello che è possibile fare però è rendere concorrenti le fasi di sense e decide, dato che, non avendo effetto sull'ambiente, non alterano la percezione dei vari agenti. Una volta che tutti gli agenti hanno pianificato correttamente l'azione da eseguire si prosegue poi ad una esecuzione sequenziale della fase di act. Anche questa fase sarebbe teoricamente eseguibile in modo concorrente, ma per come è stato progettato il simulatore questo porterebbe ad errori. Infatti nel modello scelto l'ambiente ha un ruolo attivo nella gestione delle azioni ed è quindi necessario che queste vengano eseguite sequenzialmente per garantire riproducibilità.

II. Strategia Risolutiva

Essendo chiari dall'analisi i task da svolgere in ciascun passo della simulazione si è deciso di adottare il *Task Decomposition Pattern* ed in seguito ad una *Dependency Analysis* la simulazione è stata scomposta in due fasi. Nella prima è prevista l'esecuzione concorrente dei task di sense e decide. Nella seconda saranno eseguite sequenzialmente le varie act. L'alternanza di queste fasi è gestita attraverso un meccanismo di sincronizzazione. L'architettura generale è mostrata in Figura 1.

A. Architettura Concorrente

Per la gestione della concorrenza è stata adottata una semplice tecnica di *Partitioning*. Questa decisione deriva dal fatto che il numero e il tipo di task da eseguire è già noto dall'inizio della simulazione ed è quindi possibile assegnarli staticamente a dei *Worker*. Inoltre non è stato necessario neanche introdurre una figura di *Master* poiché i worker sono autonomi nella gestione dei task e nella coordinazione reciproca.

Questo è stato realizzato attraverso:

- ***TaskSplitter***: Assegna gruppi di task ai worker bilanciando il carico di lavoro.
- ***SimulationWorker***: Componente attivo che si occupa dell'esecuzione dei tasks assegnatigli

B. Gestione degli Agenti

I vari agenti sono gestiti a partire da una *Simulation* che si occupa della fase di setup. Il setup consiste di una fase di init che comprende il setting dei parametri (t0, dt, enviroment, agents, numSteps, numThread) e una successiva fase dove viene gestito l'avvio della simulazione. Questa seconda fase consiste nella creazione di una barriera, in particolare una *CyclicBarrier*, e la distribuzione dei vari tasks attraverso il TaskSplitter. La barriera è usata dai diversi SimulationWorker per sincronizzarsi ed eseguire i passi sequenzialmente. Il problema dell'esecuzione dei task sequenziali è stato risolto tramite l'aggiunta di un comando eseguibile dalla barriera (*Runnable*). Questo comando sarà eseguito ogni volta che la barriera viene "rotta", quindi in seguito all'esecuzione dei task di sense e decide e prima dell'inizio del passo successivo. Tramite questa implementazione

di fatto l'oggetto Simulation non opera attivamente all'esecuzione dei passi ma avvia solamente la simulazione.

C. Logic View (LV)

La gestione dell'interfaccia grafica (GUI) è stata realizzata semplificando il pattern MVC (Model-View-Controller), in cui il model e il controller sono considerati come un'unica entità (package), questa combinazione prende il nome di Logic-View. Dopo aver avviato il programma, l'utente avrà la possibilità di personalizzare i parametri della simulazione tramite la GUI; una volta avviata la simulazione, la gestione verrà trasferita alla logica del programma, la quale offrirà metodi, per poter stoppare, riprendere e riavviare la simulazione. La possibilità di utilizzare questi metodi è resa possibile grazie all'introduzione di un *SimulationRunner*, un componente attivo che gestisce l'avanzamento della simulazione mediante l'utilizzo di un apposito flag. Per rendere efficace il controllo è stato necessario aggiungere la Simulation ai partecipanti della barriera di modo da avere un punto di controllo sul flusso di esecuzione. Inoltre, la GUI, grazie all'implementazione dell'interfaccia *SimulationListener*, è in grado di ricevere notifiche riguardanti lo stato della simulazione, fornendo all'utente gli aggiornamenti sullo stato attuale.

III. Comportamento del Sistema

Il comportamento del sistema è rappresentato dalla Petri-Net in Fig. 2.

IV. Elementi di Randomicità

Nei contesti di simulazione appaiono spesso elementi di randomicità, per cui è stata introdotta la possibilità di creare agenti i cui parametri sono estratti casualmente da un fissato delta di variazione. Per garantire riproducibilità l'estrazione è effettuata a partire da un *seed* fornito in fase di creazione dell'agente.

V. Identificazione di proprietà di correttezza e verifica

Per garantire il corretto funzionamento del sistema sono state di seguito effettuate sia verifiche specifiche rispetto al comportamento di componenti chiave (*monitor*) sia verifiche generali rispetto ai risultati ottenuti in seguito all'esecuzione di esempi di simulazioni.

A. Barriera

La barriera essendo l'unico strumento con cui viene gestita la sincronizzazione dei worker è il componente su cui si basa il corretto funzionamento del sistema. Di seguito è riportato il codice di test utilizzato per verificare che la rottura e il reset funzionassero correttamente. A supporto di questo test è stato utilizzato un counter Thread-Safe.

Listing 1. Barrier Code

```
public class BarrierTest {

   public static void main(String[]
       args) {
      int participants = 2; //works
          with 2, breaks with 3 (not
          enough threads to hit the
          barrier)
      Barrier barrier = new
          CyclicBarrier(participants);
      Counter counter = new Counter();

      new SimpleSimulationWorker
      (barrier,participants,counter)
      .start();
      new SimpleSimulationWorker
      (barrier,participants,counter)
      .start();
   }

   private static class
       SimpleSimulationWorker extends
       Thread{

      private final Barrier barrier;
      private final Counter counter;
      private final int participants;

      public SimpleSimulationWorker
      (Barrier barrier,
      int participants,
      Counter counter){
         this.barrier = barrier;
         this.participants =
             participants;
         this.counter = counter;
      }

      @Override
      public void run() {
         try {
            assert
                this.counter.getValue()
                == 0;
            this.barrier.hitAndWaitAll();
            this.counter.inc();
            this.barrier.hitAndWaitAll();
            assert
                this.counter.getValue()
                == participants;
         } catch (InterruptedException
             e) {
            throw new
                RuntimeException(e);
         }
      }
   }

}
```

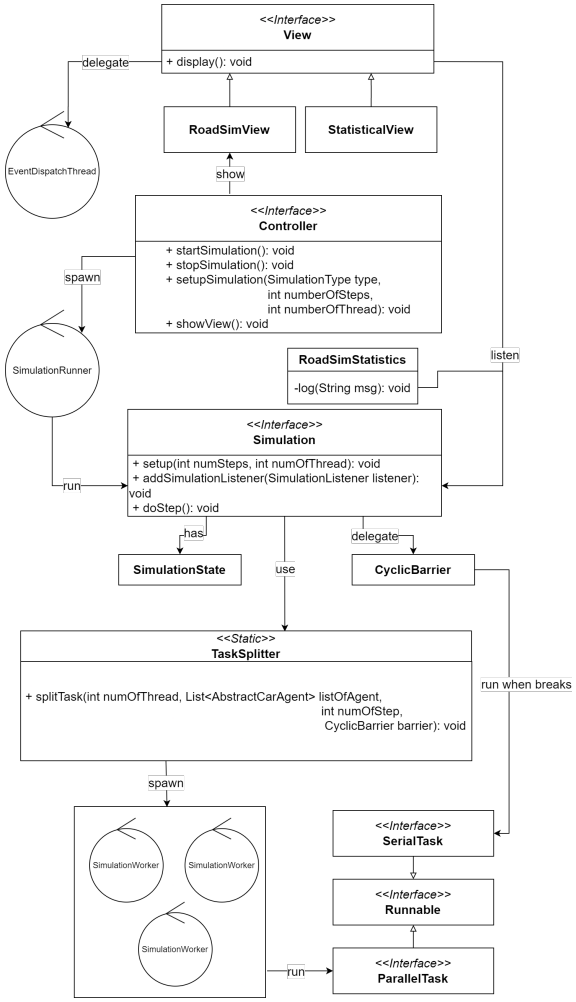

Figure 1. UML

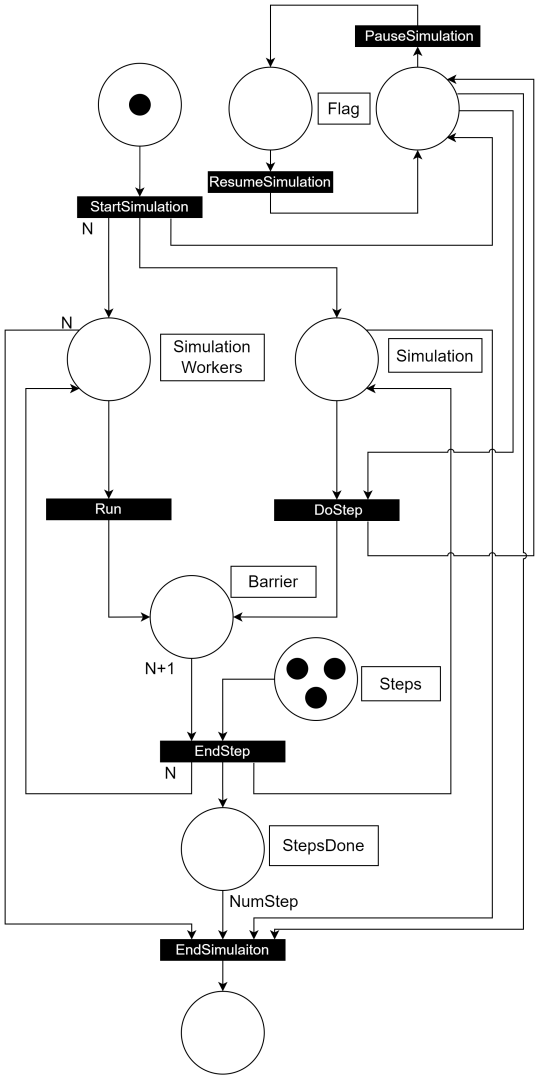

Figure 2. Petri Net

Dai risultati di JPF si può notare che non ci sono problemi riguardo le asserzioni richieste, quindi la sincronizzazione è rispettata, e anche la rottura avviene correttamente poiché impostando un numero di partecipanti maggiore rispetto ai thread creati si genera *deadlock*.

```
Barrier b = new CyclicBarrier(2);
==================
no errors detected
================== statistics
elapsed time:       00:00:01
states:             new=942,
                    visited=1050,
                    backtracked=1992,
                    end=25
...
Barrier b = new CyclicBarrier(3);
==================
error #1:
"deadlock encountered:
    thread ass.BarrierTest$Si..."
```

B. Flag di Controllo

È a tutti gli effetti un semplice monitor, avendo tutti i metodi pubblici *syncronized*. Quindi è garantita la mutua esclusione e non ci sono problemi di lost update. Non essendoci inoltre punti nel codice dove si vuole controllare lo stato del flag e modificarlo non si creano neanche possibili problemi di *check-and-act*

C. Criticità

L'uso della barriera con Runnable non è in generale Thread-Safe. Osservando il codice di esempio infatti si nota che è presente un problema di lost update. In questo caso semplificato sono possibili soluzioni banali (e.g. aggiungere la keyword synchronized al metodo inc() oppure utilizzare come wrapper della variabile "i" un counter Thread-Safe) ma in generale in presenza di componenti non Thread-Safe

Listing 2. Runnable Code

```
public class TestRunnable {
private static int i = 0;
private static void inc(){i++;}
public static void main(String[]
    args) throws
    InterruptedException {
   int iterations = 500000;
   Barrier barrier = new
       CyclicBarrier(1,
       TestRunnable::inc);
   Thread thread = new Thread(()->{
         for (int j = 0; j <
             iterations; j++) {
            try {
               barrier.hitAndWaitAll();
            } catch
                (InterruptedException
                ignored){}
         }
      }
   );
   thread.start();
   for (int j = 0; j < iterations;
       j++) {
      inc();
   }
   thread.join();
   System.out.println(i);
   // Expected: 1000000,
   // Example Actual: 989701
}
}
```

può portare ad errori. Nello sviluppo della versione concorrente del simulatore si è voluto semplicemente aggiungere un livello di gestione della concorrenza senza però andare a modificare i componenti forniti inizialmente (se non attraverso semplici refactor), per cui essendo l'ambiente e gli agenti dei semplici componenti passivi il loro utilizzo in più componenti attivi va gestito in modo opportuno. La fase di inizializzazione è eseguita su un singolo thread quindi non crea problemi. Per quanto riguarda la fase di esecuzione ogni componente o è acceduto dai partecipanti alla barriera (SimulationWorkers e Simulation) o dal Runnable assegnato alla barriera. Poiché ad ogni SimulationWorker è assegnato un componente passivo diverso e la barriera crea separazione temporale per l'uso di oggetti condivisi tra Runnable e workers non si vengono a creare situazioni critiche dove si possono generare errori.

D. Test

Per avere un veloce ed efficace metodo per confermare il determinismo e la correttezza del simulatore parallelo, sono stati scritti dei test. La procedura prevede la registrazione dei log delle simulazioni sequenziali e la loro successiva comparazione con i log corrispondenti delle simulazioni parallele. Il superamento dei test avviene quando i log delle due modalità di simulazione risultano essere identici, confermando così il determinismo e la correttezza del simulatore parallelo.

VI. Prove di Performance e considerazioni relative

Per la raccolta dei dati, è stata implementata la classe *RunTrafficSimulationMassive*. La simulazione viene gestita su diversi thread, con un intervallo che varia da 2^0 a 2^{10} thread. Sono stati considerati differenti numeri di auto nella simulazione: 1000, 2000, 5000, e 10000 veicoli. Ogni simulazione ha avuto una durata di 100 step. Le simulazioni, sono state effettuate su tre diverse tipologie di processori: M1, M3 e i5-1035G1.

Per garantire una rappresentazione completa delle prestazioni del sistema, sono stati calcolati i seguenti indicatori medi:

- **Tempi medi di esecuzione:** la media dei tempi di esecuzione delle simulazioni per ogni configurazione di thread, numero di auto e processore. Fig. 3
- **Speedup medio:** il rapporto tra il tempo di esecuzione della simulazione seriale e il tempo di esecuzione su un numero variabile di thread, in media su tutte le configurazioni di numero di auto e processore. Fig. 4
- **Efficienza media:** il rapporto tra lo speedup medio e il numero di core fisici, in media su tutte le configurazioni di numero di auto e processore. Fig. 5

Questi indicatori forniscono una visione chiara delle prestazioni del sistema in vari scenari di carico e configurazioni hardware. Tab. I

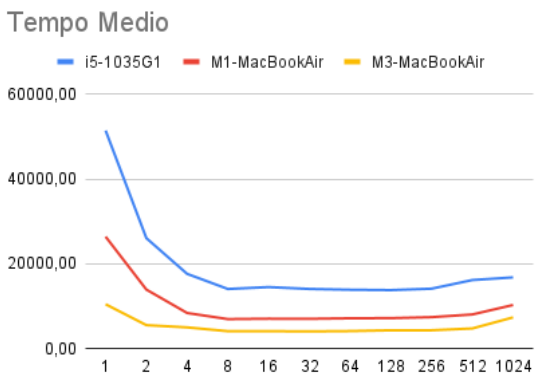

Figure 3. Tempo di esecuzione medio

I risultati mostrano un notevole miglioramento delle prestazioni all'aumentare del numero di thread. Il picco di efficienza si raggiunge con l'impiego di 8 thread, corrispondente al numero di core dei processori usati. Superati gli 8 thread l'efficienza e lo

Table I
TABELLE DELLE PERFORMANCE MEDIE

i5-1035G1			
Threads	Tempo Medio	SpeedUp Medio	Efficienza Media
1	51416,00	0,97	0,97
2	26056,75	1,83	0,92
4	17640,75	2,68	0,67
8	14075,25	3,41	0,43
16	14521,50	3,40	0,43
32	14083,50	3,54	0,44
64	13897,25	3,47	0,43
128	13806,75	3,36	0,42
256	14146,75	3,06	0,38
512	16181,25	2,50	0,31
1024	16810,50	2,03	0,25
MacBookAir M1			
Threads	Tempo Medio	SpeedUp Medio	Efficienza Media
1	26403,50	0,97	0,97
2	13933,00	1,82	0,91
4	8413,75	3,17	0,79
8	6975,50	3,73	0,47
16	7071,00	3,69	0,46
32	7043,75	3,74	0,47
64	7167,25	3,49	0,44
128	7202,00	3,35	0,42
256	7435,25	3,02	0,38
512	8069,00	2,36	0,30
1024	10303,50	1,63	0,20
MacBookAir M3			
Threads	Tempo Medio	SpeedUp Medio	Efficienza Media
1	10478,75	1,12	1,12
2	5549,75	1,98	0,99
4	5024,75	2,77	0,69
8	4127,50	3,57	0,45
16	4130,25	3,70	0,46
32	4078,50	3,70	0,46
64	4152,75	3,55	0,44
128	4337,75	3,29	0,41
256	4345,25	2,77	0,35
512	4771,00	2,02	0,25
1024	7380,75	1,15	0,14

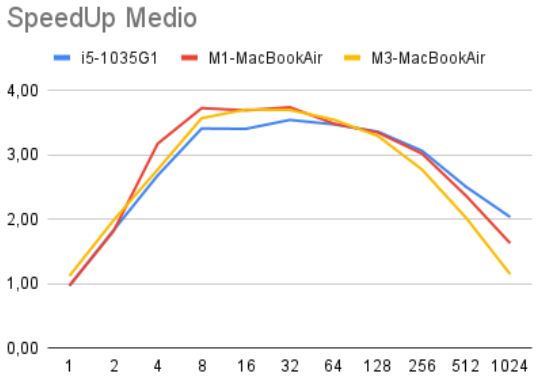

Figure 4. SpeedUp medio

speedup rimangono pressoché costanti fino a circa 64 e poi calano. Tipicamente l'aumento della granularità consente una maggior efficacia della distribuzione del carico tra i core, ma quando questa risulta eccessiva si ha una perdita di performance dovuta all' overhead di gestione dei thread (e.g. Context Switch).

A. *Performace su Apple Silicon*

Durante i test con i chip Apple Silicon (M1 e M3), si sono ottenuti risultati poco consistenti. In particolare, le implementazioni del programma in modalità seriale hanno mostrato tempi di esecuzione molto variabili, con differenze anche fino a 2,5 volte. Ad esempio, la simulazione massiva seriale con 10.000 automobili e 100 step su M3 variava tra i 30 e i 70 secondi. Questa diversità è dovuta al fatto che i chip Apple Silicon sono dotati di quattro core ad alta efficienza e quattro core ad alte prestazioni, e se non viene specificato alcun parametro, il sistema operativo decide in base alle condizioni della macchina quale tipo di core utilizzare durante l'esecuzione del programma seriale [1]. Tuttavia, con l'aumentare del numero di thread e l'adozione della versione parallela del programma, questa variabilità scompare. Per l'acquisizione dei dati, sono state considerate solo le esecuzioni seriali su core ad alte prestazioni, mentre le altre sono state scartate al fine di evitare valori anomali di speedup ed efficienza.

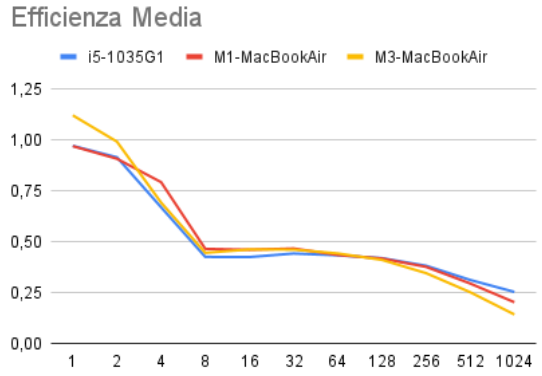

Figure 5. Efficienza media

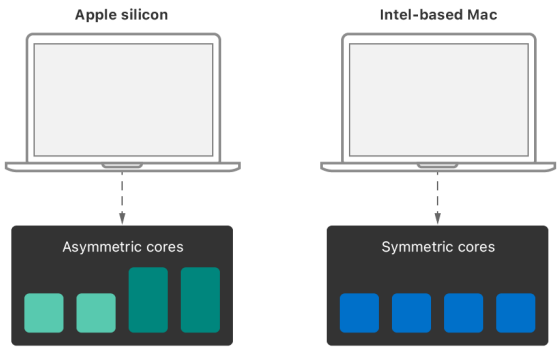

Figure 6. Aseimmetria tra core in Apple Silicon [1]

.

Per capire come nell'effettivo i chip Apple Silicon eseguono lo scheduling di un processo single core sul processore, il programma è stato eseguito in due modalità:

- `taskpolicy -B -p PPID` Fig. 7 A
- `taskpolicy -b -p PPID` Fig. 7 B

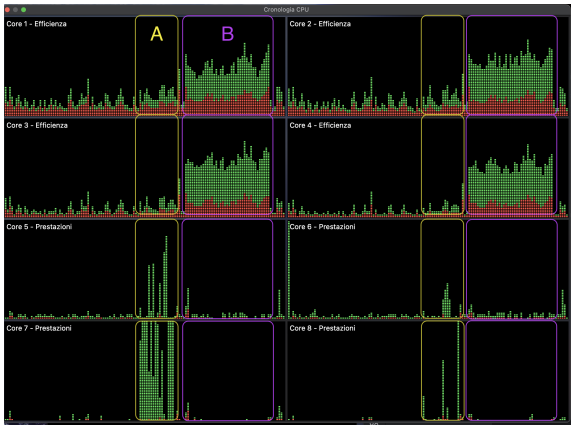

Figure 7. Uso CPU su chip M3 con diverse taskpolicy

La gestione della Qualità del Servizio (QoS) è fondamentale per assicurare che le applicazioni su Apple Silicon siano reattive ed efficienti dal punto di vista energetico. Le classi QoS assegnate alle attività indicano l'importanza di ciascuna operazione per il sistema, influenzando la priorità e la pianificazione delle stesse [1]. Su Apple Silicon, la classe QoS influisce su quale tipo di core (Performance o Efficiency) viene utilizzato per eseguire un'attività. I compiti di background tendono ad essere eseguiti sui core Efficiency per massimizzare la durata della batteria. È importante notare che tramite parametri come la taskpolicy (usata per eseguire il test in Fig. 7), è possibile solo indicare se un processo deve prediligere core efficienti o meno, senza poter forzare una priorità QoS maggiore. È il sistema operativo che prende la decisione finale su quale core utilizzare, basandosi sulle impostazioni di QoS e sulle esigenze del sistema [2] .

REFERENCES

[1] "Tuning your code's performance for apple silicon." [Online]. Available: https://developer.apple.com/documentation/apple-silicon/tuning-your-code-s-performance-for-apple-silicon

[2] "How you can't promote threads on an m1," Jan 2022. [Online]. Available: https://eclecticlight.co/2022/01/24/how-you-cant-promote-threads-on-an-m1/