Relazione

Programmazione Concorrente e Distribuita

Primo assignment

A.A. 2021/2022

Eleonora Bertoni, Elisa Albertini, Denys Grushchak

Sommario

Analisi del problema

Nella fase iniziale del lavoro ci siamo concentrati sull'analisi del problema della parallelizzazione dei compiti che l'applicazione deve svolgere.
In particolare, ci siamo focalizzati sui vari punti critici dell'esecuzione del ciclo di istruzioni che viene effettuato per ogni corpo:

- calcolo delle forze
- calcolo dell'accelerazione
- aggiornamento della velocità
- aggiornamento della posizione
- controllo dei bordi
- gestione del virtual time
- *aggiornamento della GUI*

Di seguito riportiamo la *Dependency Analysis* nella quale studiamo le dipendenze tra i dati e le dipendenze temporali del sistema.

Calcolo delle forze

In lettura (corpi esterni a quello attuale):

- Posizione
- Massa

In lettura (corpo attuale):

- Velocità
- Posizione

Sapendo che in questa fase ci sono solo letture, non si presentano sostanziali problemi per il calcolo della forza di attrito e della forza di repulsione.

Calcolo dell'accelerazione

In lettura (corpo attuale):

- Massa
- Forze calcolate

Questo passaggio è dipendente da quello precedente in quanto se non sono state calcolate le forze non può essere calcolata l'accelerazione.

Aggiornamento della velocità

In lettura (corpo attuale):

- Accelerazione

In scrittura (corpo attuale):

- Velocità

Anche questo punto deve attendere l'esecuzione del passaggio precedente.

Aggiornamento della posizione

In lettura (corpo attuale):
- Velocità

In scrittura (corpo attuale):
- Posizione

Il calcolo della posizione deve attendere che tutte le forze di tutti i body siano state calcolate dato che per calcolarle viene richiesta la posizione dei body.

Controllo dei bordi

In lettura (corpo attuale):
- Velocità
- Posizione

In scrittura (corpo attuale):
- Velocità
- Posizione

Il ragionamento per questo blocco di istruzioni è lo stesso dell'aggiornamento della posizione ma in più questo punto deve essere eseguito dopo quello precedente.

Gestione di virtual time

In caso di concorrenza, virtual time deve essere condiviso fra tutti i thread, sia in scrittura che in lettura, quindi ha la necessità di essere gestito in maniera atomica. Nel caso le varie iterazioni del ciclo di aggiornamento siano fatte da più thread è importante gestire la modifica del virtual time e del contatore delle iterazioni in *sezione critica* (per gestire i possibili problemi di interleaving).
L'unico vincolo per l'aggiornamento di questo blocco di codice è di essere eseguito prima dell'aggiornamento della GUI.

Aggiornamento della GUI

L'aggiornamento può essere fatto una volta che le posizioni di tutti i body sono state ricalcolate ed è stato aggiornato Virtual Time.

Conclusioni

L'idea che abbiamo avuto per strutturare il comportamento del sistema è di suddividere il codice in porzioni divise da due barriere che richiedono il completamento delle operazioni dei singoli blocchi di operazioni per gestire le loro dipendenze.
Per quanto riguarda l'aggiornamento del virtual time e l'incremento del contatore vorremmo inserirli all'interno di un monitor per gestire la sezione critica.
I blocchi sono:

- Calcolo delle forze, dell'accelerazione e della velocità
- Calcolo della posizione e controllo dei bordi
- Aggiornamento del virtual time & aggiornamento della GUI

Strategia risolutiva e architettura proposta

Dopo aver analizzato l'applicazione seriale abbiamo deciso che strategie utilizzare e come strutturare il progetto in modo che la parallelizzazione fosse il più efficace possibile.

Il programma viene gestito da un main thread che crea tutti i thread worker. Esso aspetta che tutti i thread abbiano completato il lavoro facendo una join.

Prima fase

Per quanto riguarda la prima versione dell'applicazione (quella senza la GUI) abbiamo scelto di utilizzare una *shared space architecture* dove ogni thread lavora su un intervallo ben definito di corpi. Abbiamo preferito questo tipo di architettura in quanto il problema permette di avere una equa distribuzione del lavoro su ogni thread.

Abbiamo suddiviso, dunque, il lavoro in due blocchi: nel primo calcoliamo e aggiorniamo la velocità dei corpi e nel secondo aggiorniamo la sua posizione e controlliamo i bordi.
Per garantire la corretta esecuzione del secondo blocco di operazioni dobbiamo essere sicuri che tutti i thread abbiano finito l'esecuzione del primo blocco. Affinché avvenga questo abbiamo deciso di utilizzare una *barrier* che li sincronizzi.
Successivamente abbiamo utilizzato una seconda *barrier* che controlli che tutti i thread abbiano eseguito anche il secondo blocco di operazioni, per fare in modo che i thread possano riniziare il ciclo di istruzioni.

Parlando del virtual time e dell'aggiornamento del counter delle iterazioni abbiamo scelto di implementare un monitor specifico in quanto i dati che devono essere modificati in questo caso sono condivisi da tutti i thread e quindi va garantito che essi siano acceduti in maniera atomica.

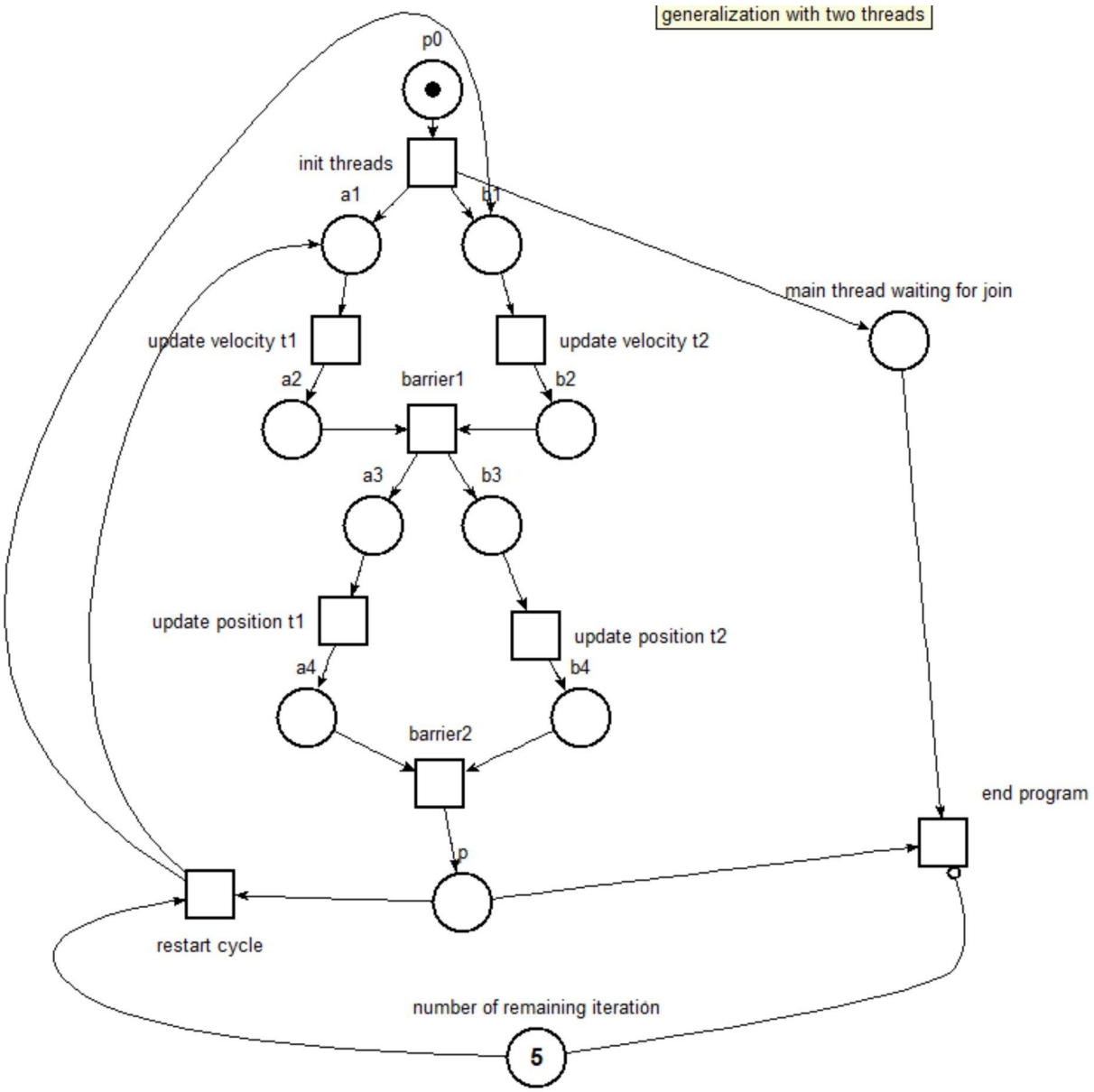
Seconda fase

Per quanto riguarda la seconda versione dell'applicazione (quella con la GUI) abbiamo inserito la GUI passando il *viewer* alla simulazione e incapsulando il suo aggiornamento all'interno del monitor che si occupa del counter delle iterazioni.

In fine sono state realizzate le funzionalità di Start e Stop.
La Start è stata realizzata inserendo il programma in un ciclo infinito che va in wait una volta che la simulazione termina. Quando viene premuto il tasto Start la simulazione viene ricreata da capo.
La Stop, invece, imposta un flag della *barrier* e quando tutti i thread sono bloccati nella *barrier* l'attuale iterazione viene considerata come l'ultima per la simulazione. In questo modo i thread worker terminano la loro esecuzione in maniera naturale.

Comportamento del sistema

Sistema completo (semplificato)



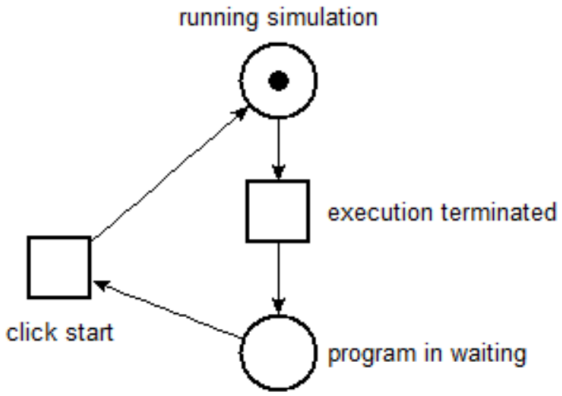
Rete di Petri semplificata che generalizza il comportamento del sistema nel caso di 2 thread e 5 step

Il thread principale, una volta creati i worker thread, attende che tutti facciano *join* per poi terminare il programma.
I due thread eseguono parallelamente le varie operazioni, come calcolare e aggiornare la velocità, nello specifico, per poi bloccarsi sulla prima *barrier* in attesa che l'altro abbia finito.
Successivamente, quando entrambi i thread hanno eseguito il primo blocco di istruzioni, passano ad eseguire il secondo, sempre parallelamente, dove dovranno aggiornare la posizione del corpo e controllare i bordi della simulazione per poi bloccarsi alla seconda *barrier*.

Una volta superata anche l'ultima barriera inizia l'iterazione successiva. Quando le iterazioni sono concluse il programma termina.
E' da notare che la condizione per fare uscire i thread dalla barriera viene abilitata solo quando tutti i thread del sistema si trovano al suo interno. In questo modo si evita che alcuni di essi possano rimanere bloccati in wait.

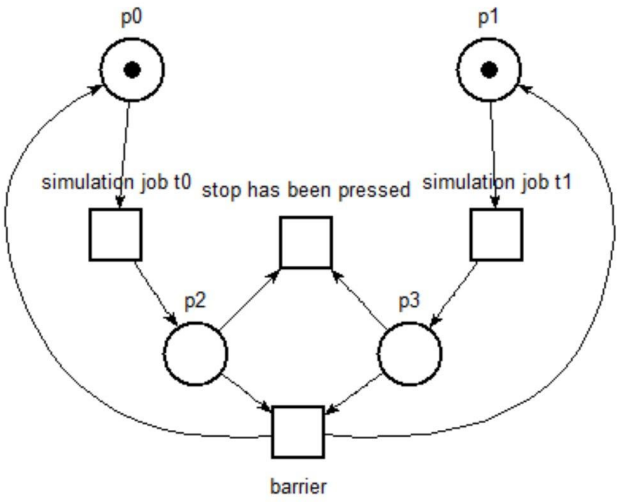
Start & Stop

Per schematizzare in maniera più esemplificativa si è deciso di realizzare una rete di Petri che descrivesse lo Start a un livello più astratto rispetto allo Stop.



Rete di Petri dello Start

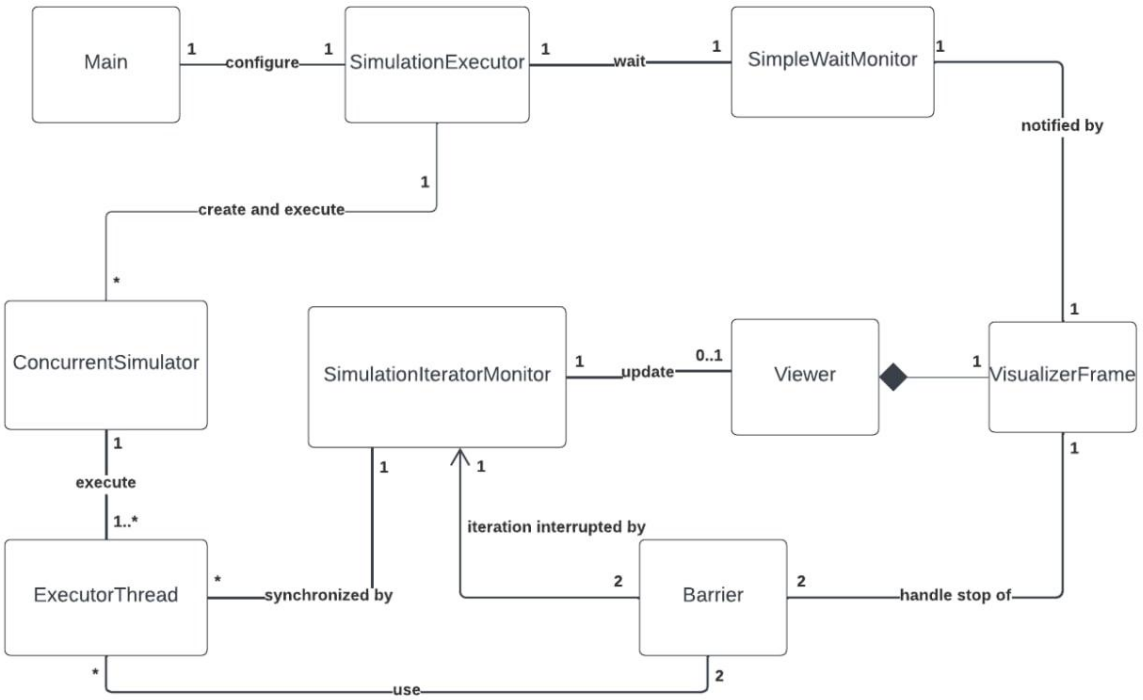
Lo Start viene premuto quando il sistema si trova in waiting (dopo che è stato premuto il tasto Stop o quando l'esecuzione è terminata) e fa ripartire la simulazione da capo.



Rete di Petri dello Stop
Per semplicità, le due *barrier* siano state collassate in una singola

Questo schema mostra come il sistema, nel caso specifico 2 thread, possa andare in wait sia quando l'esecuzione è terminata sia quando viene premuto il tasto Stop.

Implementazione del sistema



UML del sistema

Questo schema UML rispecchia la struttura (ad alto livello) del sistema.
In particolare è importante notare che la barriera ha al suo interno *SimulationIteratorMonitor* e registra il segnale di stop tramite un Event Handler (inviato da *VisualizerFrame* - bottone). Una volta che tutti i worker thread sono nella barriera, tramite *IteratorMonitor*, vengono interrotte le iterazioni.

Modularità del codice

- L'esecuzione con e senza GUI può essere completamente scollegata dal sistema (passandola o meno alla classe che gestisce la simulazione).
- Le barriere possono essere riutilizzate in altri programmi.
- SimulationIteratorMonitor può essere riadattato facilmente per altri programmi.

Difficoltà incontrate

Per evitare una riprogettazione completa della GUI abbiamo deciso di implementare in modo statico le chiamate agli handler dei tasti per le funzionalità di Start e Stop.
In futuro, questa particolare implementazione, potrebbe essere migliorata.

Prove di performance

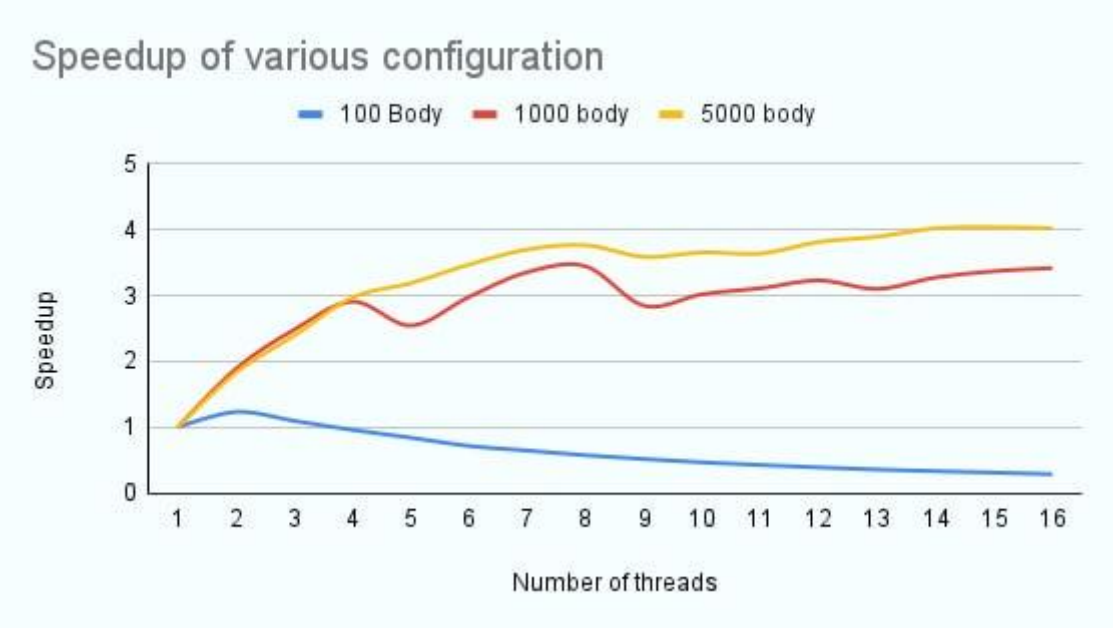


Grafico degli speed-up

Per testare lo speed-up abbiamo eseguito più volte il sistema (senza GUI) e poi abbiamo calcolato il tempo medio delle varie esecuzioni. Il test è stato eseguito tramite un PC con quattro core fisici (otto con hyperthreading).
Il grafico sopra riportato è frutto delle misurazioni ottenute tramite numerosi test.

Facendo le misurazioni abbiamo notato che il tempo di esecuzione, con lo stesso numero di body ma con step diversi, è proporzionale (lo speedup non cambia). Per questo motivo abbiamo scelto step ragionevoli in modo da non avere esecuzioni né troppo brevi né troppo lunghe.

Leggendo il grafico abbiamo notato che:

- con 100 body, aumentando il numero dei thread, lo speed-up cala. Questo è dovuto dall'overhead della gestione del thread → le risorse necessarie per gestire una divisione del lavoro in modo concorrente sono molto più alte e costose rispetto al vantaggio ricevuto.
- con 1000 body vediamo che il lavoro viene suddiviso in modo più efficiente e non notiamo più l'overhead perché ogni thread fa una quantità di lavoro sufficiente per compensarlo e, allo stesso tempo, aumentare le prestazioni.
- con 5000 body il sistema si comporta come con 1000 body ma la gestione del tempo, fatta dai thread, è ancora migliore e dunque l'overhead si nota ancora meno → è, in percentuale, ancora più piccolo rispetto al tempo di lavoro utile al sistema.

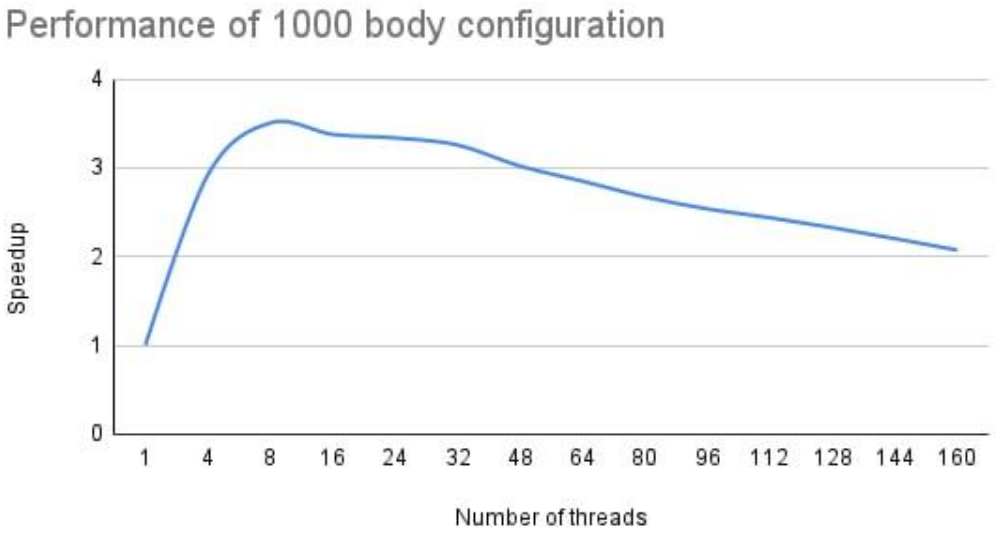


Grafico che visualizza la degradazione delle performance del sistema con un numero di thread eccessivo

Alla luce della precedente analisi e di questo grafico il numero di thread ottimali, per la macchina su cui è stato fatto lo studio, è 8.
Si può però notare che per i multipli di 8 (fino a 32) le performance sono pari, o a volte maggiori, rispetto al caso migliore.

Identificazione di proprietà di correttezza e verifica

Per verificare la correttezza del nostro sistema abbiamo utilizzato JPF.
All'inizio delle nostre prove abbiamo riscontrato alcuni problemi (come deadlock) che ci hanno portato a modificare il nostro progetto in modo che fosse corretto.

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

====================================================== statistics
elapsed time:       00:14:05
states:             new=2468880,visited=6015959,backtracked=8484839,end=4
search:             maxDepth=693,constraints=0
choice generators:  thread=2468880 (signal=33484,lock=94934,sharedRef=2208142,threadApi=201,reschedule=132119),
 data=0
heap:               new=13133800,released=754935,maxLive=772,gcCycles=8115532
instructions:       198629653
max memory:         784MB
loaded code:        classes=91,methods=1896
```

Screenshot del test finale con JPF

La nostra scelta per il testing è stata quella di semplificare il sistema (togliendo ad esempio le funzionalità di Start e Stop tutto quello che riguarda la GUI) e poi fare alcune prove con un diverso numero di thread e step (partendo da un numero più esiguo fino ad arrivare a numeri più consistenti ma che permettessero comunque un'esecuzione abbastanza breve).