Progetto d'esame per il corso di Sistemi di calcolo paralleli e distribuiti

Luca Rickler

DD settembre 2016

Tema

Simulazione di un generico sistema ad agenti comunicanti tramite canali broadcast

Indice

1 Introduzione

Nei corsi di modellizzazione ad agenti disponibili per la laurea magistrale in fisica sono proposti come strumenti per lo sviluppo dei progetti d'esame alcuni software di varia complessità. Questi, tra cui spicca la libreria JADE[1], sviluppata da Telecom Italia Lab, sono prevalentemente scritti in java, linguaggio che offre molti vantaggi, ma può portare spesso a programmi lenti ed inadatti a trattare simulazioni con un numero molto elevato di agenti. In questo lavoro mi propongo di iniziare lo sviluppo di uno strumento alternativo in C++ capace di simulare agevolmente sistemi con un elevato numero di agenti su macchine relativamente poco potenti.

Sebbene in sé possa sembrare un compito eccessivamente ambizioso, il nocciolo del programma può essere riassunto in poche parole. Infatti, l'obiettivo di questo lavoro è simulare un generico sistema ad agenti comunicanti tramite canali broadcast. A questo si può inoltre aggiungere l'assunzione che nessun agente possa eseguire più di una azione per intervallo di tempo della simulazione. Anche se può sembrare limitante, questa ipotesi semplifica il design e lo sviluppo del programma.

2 Design

2.1 Esecuzione a turni

Una qualunque simulazione ad agenti necessita di un qualche tipo di controllo sulla sua struttura temporale. Questo può essere ottenuto in vari modi, ad esempio costruendo un sistema di timer per sincronizzare l'esecuzione della sequenza di compiti dei singoli agenti. Tuttavia, il modo più semplice per controllare la struttura temporale è suddividerla in intervalli, da qui in avanti chiamati *turni*. Questi sono rappresentativi di un generico intervallo di "tempo simulato", la cui entità varia a seconda della specifica simulazione.

Data questa struttura temporale, il modo più semplice per sincronizzare l'esecuzione dei compiti degli agenti è imporre che questi non possano agire più di una volta durante lo stesso turno. In questo modo, la sequenza di compiti viene suddivisa in generiche *azioni*, eseguite una per turno per ogni agente.

Bisogna inoltre considerare che, nella maggioranza dei casi, non si conosce il dettaglio della catena di azioni. Per questo, si può supporre che, affinché una di queste catene non si interrompa rendendo inattivo un agente per il resto della simulazione, ogni azione prenoti l'esecuzione della seguente lungo la catena per il turno successivo. Questo inoltre consente un certo grado di flessibilità, permettendo all'agente di scegliere quale azione prenotare a seconda delle circostanze.

Per implementare tutto questo si è scelto di usare due code FIFO a due lock, costruite secondo il modello di Michael and Scott (1996), nei quali immagazzinare le azioni. Ogni turno verranno eseguite le azioni da una delle due code ed immagazzinate le azioni per il turno successivo nella seconda, per poi invertire i ruoli il turno seguente. Per comodità, queste possono essere incluse in un array in cui, per discernere quale coda svolga quale funzione, si utilizza un indice ciclico, incrementato a ogni turno.

[1] **www.jade.tilab.com**

2.2 Sotto-fasi

In alcuni casi, è utile considerare un'ulteriore suddivisione dei turni in sotto-fasi. Infatti, per svolgere in parallelo le azioni di ogni turno, si suppone che queste non dipendano le une dalle altre e possano essere eseguite in qualunque ordine. Tuttavia, può darsi che uno o più agenti, per poter agire, abbiano bisogno di un messaggio proveniente da un altro agente. In questo caso non è più vero che queste azioni possano essere eseguite in qualunque ordine, poiché l'agente mittente deve sempre agire prima degli altri. Questo problema può facilmente essere risolto suddividendo il turno in sotto-fasi e classificando le azioni in una scala di priorità di esecuzione. Quindi si può imporre che le azioni con più alta priorità vengano eseguite in una sotto-fase precedente alle azioni con più bassa priorità.

Ciò può essere ottenuto aggiungendo una coda FIFO extra per ogni sotto-fase dopo la prima al modello discusso in precedenza.

Nella corrente implementazione, la priorità di esecuzione è definita alla creazione dell'agente e non può essere cambiata durante l'esecuzione. Inoltre, il numero di sotto-fasi è definito durante l'inizializzazione della simulazione indipendentemente dalla priorità dei singoli agenti.

2.3 Programma sequenziale

La struttura discussa nei paragrafi precedenti è condensata nel semplice diagramma di flusso mostrato nella figura 1. Il tutto comincia con l'inizializzazione delle strutture dati fondamentali della simulazione, discusse nel paragrafo 3.1, ed è subito seguita dalla costruzione degli agenti. Quindi si procede a riempire le code di esecuzione, il cui numero è dettato dal criterio discusso nel paragrafo 2.2, secondo la priorità di esecuzione degli agenti. Queste due fasi possono essere anche accorpate, eseguendole in un unico passaggio per ogni agente. Finiti questi passi preliminari, inizia la vera e propria simulazione, con l'esecuzione delle azioni immagazzinate nelle code FIFO. Questo passaggio viene ripetuto finché non si è raggiunto il limite massimo di turni specificato durante la fase di inizializzazione. Questo segna la fine del programma.

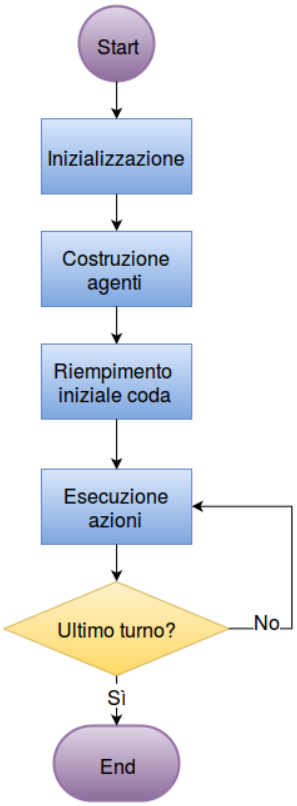

Figura 1: *Diagramma di flusso del programma sequenziale*

Questo programma sequenziale è stato utilizzato come riferimento per valutare l'efficacia della parallelizzazione discussa nel paragrafo successivo.

2.4 Programma parallelo

Il programma sequenziale discusso nel paragrafo precedente può essere parallelizzato separando l'esecuzione delle azioni ed il controllo sulla struttura temporale. Infatti si può costruire una variante del paradigma *master-worker*, in cui il master controlla soltanto il passaggio da un turno all'altro, non assegnando direttamente i compiti ai worker. Per poter implementare una struttura del genere è necessario utilizzare un'architettura *shared memory*, in quanto la coda FIFO da cui i worker ricevono le azioni da compiere deve essere condivisa.

Applicando queste modifiche allo schema precedente si ottiene il diagramma di flusso mostrato nella figura 2. Le fasi di inizializzazione non vengono toccate dalle modifiche, che interessano esclusivamente la fase esecutiva. Infatti, subito dopo il riempimento iniziale della coda, vengono creati i thread worker. Il master (mostrato a sinistra nella figura) rimane quindi in attesa che i worker (nella parte destra della figura) eseguano le azioni prenotate per quel turno. Giunti alla prima sincronizzazione, i worker si mettono in attesa, mentre il master controlla se vi sono turni rimanenti da eseguire. In caso affermativo, prepara il turno successivo spostando l'indice ciclico dell'array di code FIFO. Quindi il programma arriva alla seconda sincronizzazione ed il ciclo continua. La chiusura avviene quando i worker ricevono un segnale di uscita da parte del master.

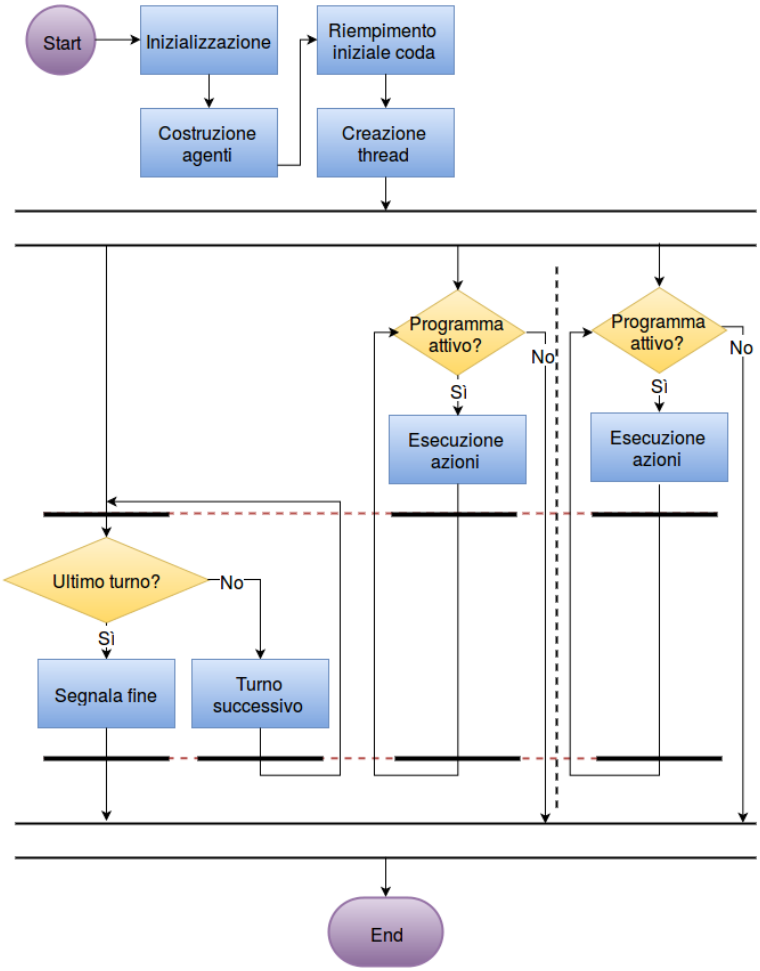

Figura 2: *Diagramma di flusso del programma parallelo*

Questa struttura può essere facilmente modificata per includere le sotto-fasi discusse nel paragrafo 2.2. Infatti basta ripetere il ciclo di esecuzione una volta in più per ogni sotto-fase aggiuntiva dopo la prima.

2.5 Messaggi broadcast

Il tipo di messaggio di più semplice implementazione è il messaggio broadcast. Infatti, per poter inviare un tale messaggio, un agente non deve avere una conoscenza specifica degli altri agenti, ma è sufficiente che abbia accesso ad un elemento esterno che conosca tutti gli agenti e possa consegnare il messaggio. Tale elemento esterno è la classe *Runtime* discussa nel paragrafo 3.1. Il messaggio viene quindi consegnato in una casella di dimensioni illimitate, implementata usando una coda FIFO con le stesse caratteristiche di quelle discusse in precedenza. Sebbene sia possibile parallelizzare questo meccanismo, per semplicità si è preferito mantenere un approccio sequenziale, in cui la consegna del messaggio viene eseguita per tutti gli agenti da un unico worker, lo stesso che ha preso in carico lo svolgimento dell'azione che lo ha inviato.

3 Implementazione

Il programma è stato scritto in *C++11*, usando la libreria *pthread* per la parallelizzazione.

3.1 Classe *Runtime* e funzine *ThreadExec*

La classe fondamentale del programma è la classe *Runtime*. Questa si occupa dell'inizializzazione della simulazione e della gestione del flusso di esecuzione. Infatti, contiene le istanze di tutte le barrier ed i lock necessari a sincronizzare l'esecuzione dei thread. Inoltre, si occupa di istanziare gli agenti, gestisce l'array di code FIFO contente le azioni da eseguire ogni turno e sotto-fase e crea i thread worker. Questi a loro volta eseguono la funzione *ThreadExec*, schematizzata nella figura 2. Infine, il Runtime si occupa della consegna dei messaggi sotto richiesta degli agenti.

3.2 Classi *Agent*, *Action* e *Message*

Gli agenti sono costruiti a partire dalla classe *Agent*. Questa contiene tutti i metodi necessari ad interagire con il Runtime per prenotare le azioni (metodo *ScheduleAction*), inviare e ricevere messaggi (metodi *Send* e *ReadMessage*). L'inizializzazione è contenuta nel metodo *Setup*, il quale deve essere implementato dagli specifici agenti della simulazione. È interessante notare che questa funzione è sempre la prima ad essere inserita nelle code di esecuzione subito dopo la costruzione dell'agente. Di conseguenza, ogni simulazione è sempre preceduta da un turno aggiuntivo di inizializzazione.

Gli agenti sono inoltre dotati di un ID unico, inizializzato dal Runtime al momento della costruzione, e contengono una lista delle loro azioni. Queste sono tutte istanze della classe *Action*. La funzione fondamentale di questa classe è il metodo *Act* ed è l'unico che verrà eseguito dai worker. L'immagazzinamento nelle code FIFO è effettuato attraverso un puntatore ad un oggetto *std::function*, contenuto dalla classe ed inserito dal Runtime nelle code sotto richiesta dell'agente proprietario. Un altro elemento importante è il puntatore all'agente proprietario *my_agent*, con il quale l'azione può accedere ai metodi necessari a prenotare azioni per il turno successivo, oltre che inviare e ricevere messaggi.

I messaggi sono oggetti della classe *Message* e fungono prettamente da contenitore passivo di informazioni. Contengono soltanto tre variabili: il

contenuto del messaggio, un puntatore all'ID dell'agente mittente ed una stringa contenete l'ontologia del messaggio. Tutte queste sono variabili private, modificabili esclusivamente tramite il costruttore ed i loro valori sono ottenibili attraverso funzioni *Get*. Ciò permette di evitare che uno dei destinatari possa in qualche modo alterare il messaggio prima che tutti lo abbiano letto.

La distribuzione dei messaggi è effettuata dal Runtime, il quale inserisce un puntatore nelle inbox dei vari agenti. Queste sono costituite da code FIFO dello stesso tipo di quelle discusse nei paragrafi precedenti. Inoltre è interessante sottolineare che il messaggio restituito dalla funzione *ReadMessage* è una copia locale, separata dal messaggio originario, il quale viene distrutto dal Runtime una volta che tutti gli agenti l'hanno ricevuto.

4 Test

Tutti i test successivi sono stati effettuati su di un portatile Acer Aspire VN7-791G-74HT, dotato di un processore Intel i7-4710HQ.

4.1 Scaling in funzione del numero di thread

Come test iniziale, è stato valutato lo scaling del codice in funzione del numero di thread, escludendo l'uso della messaggistica. Per simulare un ipotetico carico, si è assegliato agli agenti il compito di calcolare il più grande numero primo minore di 1000 con un semplice algoritmo. Ciò è stato ripetuto da 2000 agenti per tutta la simulazione, della durata totale di 3 turni, incluso il turno di inizializzazione. Questa è stata effettuata sia con la versione sequenziale del programma che con quella parallela, variando il numero di thread worker da 1 a 11 (thread totali attivi da 2 a 12). Per eliminare le interferenze di altri processi sulle misure del wall-clock time, la simulazione è stata ripetuta 7 volte per ogni configurazione e si è mediato sui risultati.

Sebbene possa sembrare una simulazione eccessivamente breve, si è optato per la durata di soli tre turni per facilitare i confronti con i test sui messaggi discussi successivamente.

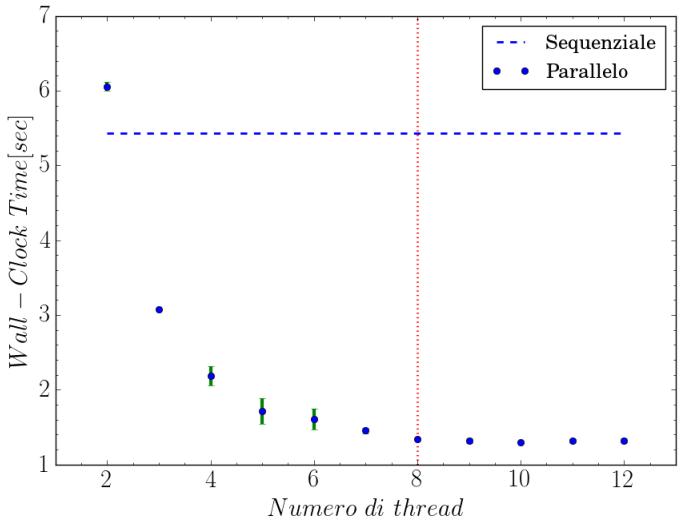

Figura 3: *Wall-clock time del test di scaling con 2000 agenti per la durata di 3 turni, con i risultati relativi ai run paralleli (punti), quello sequenziale (linea blu) ed il numero di processori logici della macchina (linea rossa).*

La figura 3 mostra le misure del wall-clock time relative a questo test. Come si può vedere, il programma parallelo risulta 4 volte più veloce nel caso di un alto numero di thread esecutori, anche se il loro numero supera quello dei processori logici della macchina (linea rossa). È interessante notare che nel caso di un solo thread worker, il programma sequenziale risulta leggermente più veloce.

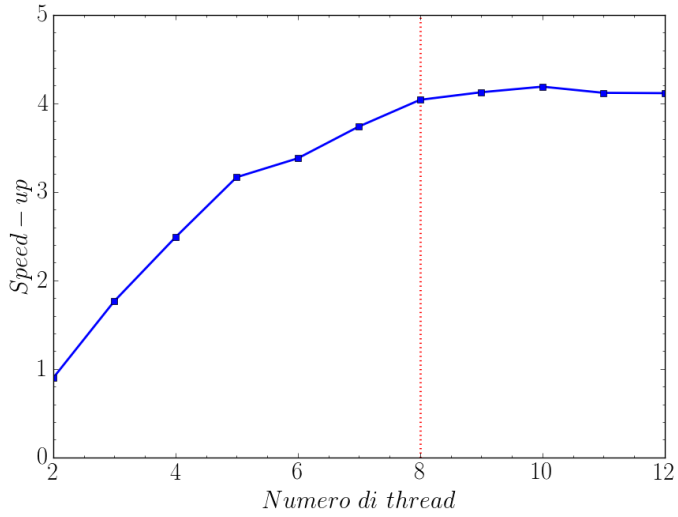

Figura 4: *Speed-up del test di scaling con 2000 agenti per la durata di 3 turni (curva blu), con il numero di processori logici della macchina (linea rossa).*

Questo è confermato dalla valutazione dello speed-up, calcolato come

$$S(n) = \frac{t_s}{t_p}$$

dove n è il numero di thread attivi, t_s è il wall-clock time del programma sequenziale e t_s quello del programma parallelo. Come si può notare dalla figura 4 , il guadagno massimo è pari ad un fattore 4. Tuttavia, si nota che la curva presenta un cambio di pendenza intorno ai 5 thread totali.

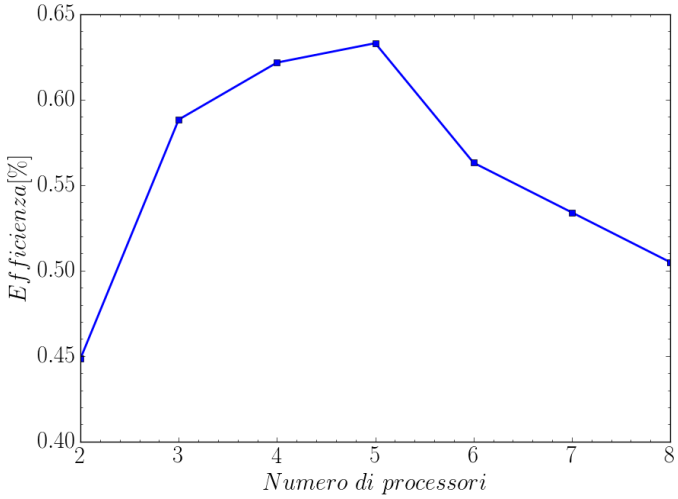

Figura 5: *Efficienza relativa al test di scaling con 2000 agenti per la durata di 3 turni.*

Il fenomeno è ancora più evidente nel grafico dell'efficienza, calcolata dividendo lo speed-up per il numero di processori usati. Questa è visibile

nella figura 5. Una possibile spiegazione può essere la presenza di lunghi tempi di attesa verso la fine di ogni turno, quando i worker finiscono di svolgere i propri compiti ed arrivano alla prima barrier.

É interessante notare che aumentando il carico per turno, semplicemente chiedendo che il numero primo da calcolare sia il più grande minore di 10^4, l'efficienza tende a diminuire più drasticamente, mentre aumenta il massimo, sempre situato sui 5 thread totali, come si può vedere dalla figura 6. Ciò rafforza l'ipotesi che vi siano lunghi tempi di attesa alla fine di ogni turno.

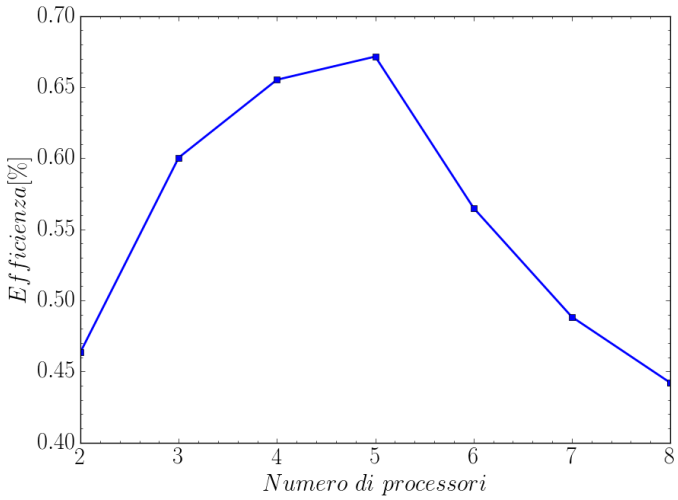

Figura 6: *Efficienza relativa al test di scaling con 2000 agenti per la durata di 3 turni. In questo caso, il numero primo da calcolare era il massimo minore di* 10^4.

4.2 Efficienza delle comunicazioni

Per testare le comunicazioni interne tra agenti è stato usato un semplicissimo problema. Gli agenti sono suddivisi in due gruppi di diversa priorità di esecuzione. Il gruppo a priorità maggiore è composto da un solo agente che genera casualmente un numero e lo comunica agli altri. Questi sono tutti nel secondo gruppo e hanno ognuno un proprio numero, diverso per ogni agente. Se uno di questi riceve un numero diverso dal proprio non fa niente e si rimette in ascolto. In caso contrario, comunica agli altri di aver ricevuto il numero giusto, ed il programma si ferma.

Contando anche il turno di inizializzazione, il tutto si conclude in soli tre turni: inizializzazione, generazione del numero, chiusura. Inoltre, sono inviati solo due messaggi in tutta la simulazione.

Per ottenere dei risultati misurabili e non eccessivamente perturbati da altri processi attivi sulla macchina, il numero di agenti è stato aumentato dai 2000 del test precedente a $2 \cdot 10^7$.

Come si può vedere dalla figura 7, che mostra il wall-clock time relativo al test sia in parallelo (punti) che in sequenziale (curva blu), vi è un piccolo guadagno in termini di tempo di esecuzione tra i 3 ed i 6 thread totali. Tuttavia, per numeri superiori ed inferiori vi è una netta perdita, soprattutto superati gli 8 thread totali.

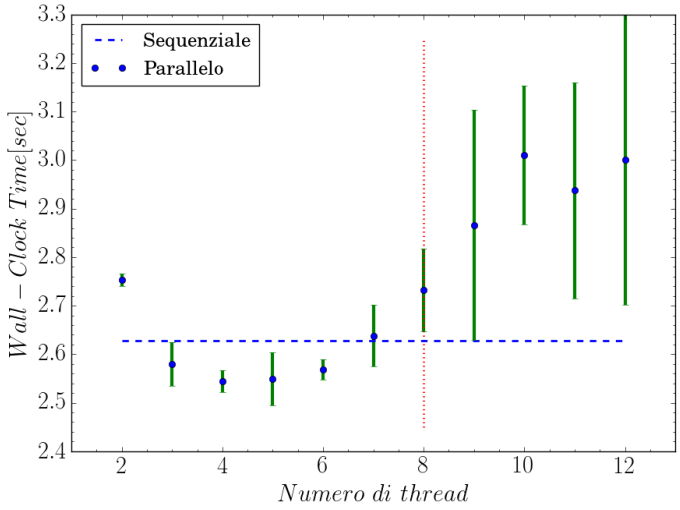

Figura 7: *Wall-clock time del test di scaling delle comunicazioni con* $2 \cdot 10^7$ *agenti per la durata di 3 turni, con i risultati relativi ai run paralleli (punti), quello sequenziale (linea blu) ed il numero di processori logici della macchina (linea rossa).*

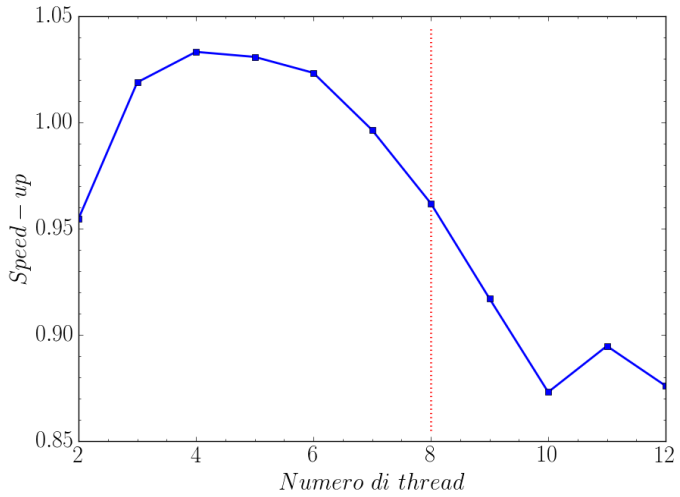

Figura 8: *Speed-up del test di scaling delle comunicazioni con* $2 \cdot 10^7$ *agenti per la durata di 3 turni (curva blu), con il numero di processori logiciq della macchina (linea rossa).*

Ciò si può più direttamente notare dal grafico dello speed-up, mostrato nella figura 8. Infatti, vi è un piccolo guadagno per un basso numero di thread ed una forte perdita per alti valori.

Però, è molto interessante osservare l'efficienza, rappresentata nella figura 9: infatti, essa diminuisce fortemente all'aumentare del numero di thread. Questo, in aggiunta alla bassa efficienza massima, pari al 48%,

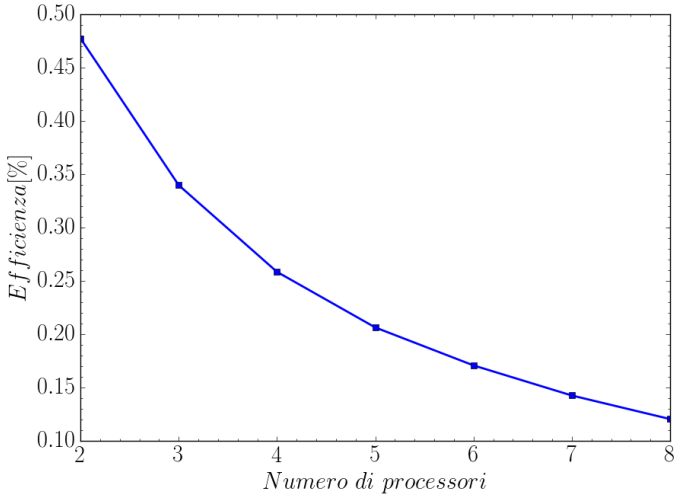

Figura 9: *Efficienza del test di scaling delle comunicazioni con* $2 \cdot 10^7$ *agenti per la durata di 3 turni.*

conferma che l'algoritmo scelto per la distribuzione e la lettura dei messaggi non è adatto alla parallelizzazione. Tuttavia, è importante osservare che metodi alternativi non porterebbero a grandi vantaggi, in quanto il problema risiede nella struttura scelta per le inbox degli agenti. Questa infatti costringe ad allocare dinamicamente memoria ogni qual volta che arriva un messaggio e, per grandi numeri di agenti, ciò comporta un forte rallentamento del processo di distribuzione.

Bisogna tuttavia osservare che il messaggio di tipo broadcast non è in realtà quello più comunemente usato da simulazioni complesse, dove spesso vengono usati messaggi tra singoli agenti. L'implementazione di questi tuttavia presenta ulteriori complicazioni che sono fuori dallo scopo di questo lavoro.

Infine, conviene evidenziare che il tempo necessario affinché $2 \cdot 10^7$ agenti ricevano e leggano un singolo messaggio broadcast è di poco superiore al secondo. Questo, in una simulazione di grandi dimensioni che non faccia eccessivo uso di questo strumento, è un tempo totalmente trascurabile. Ciò è rafforzato dall'alto numero di agenti considerati, superiore a quello della maggior parte delle simulazioni.

5 Conclusioni

In questo lavoro è stato trattato un modo per realizzare una generica simulazione ad agenti. Sebbene esistano delle limitazioni all'approccio utilizzato, le quali, ad esempio, rendono più impegnativo svolgere simulazioni in "tempo reale", permettendo agli agenti di interagire con il mondo esterno, questo presenta diversi vantaggi, primo tra tutti la facile parallelizzazione.

Si è inoltre visto che questo approccio porta a notevoli guadagni in termini di tempi di esecuzione, con uno speed-up massimo misurato pari ad un fattore 4. Tuttavia, bisogna notare che l'efficienza massima non coincide con il massimo dello speed-up nei test eseguiti.

Per quanto riguarda la messaggistica, si è osservato che l'assegnazione del compito di distribuire i messaggi ad un singolo thread worker comporta una perdita in tempo di esecuzione. Questa però è generalmente trascurabile se comparata al guadagno complessivo precedentemente misurato.

6 Possibili sviluppi futuri

Il programma allo stato attuale non presenta molte funzionalità potenzialmente utili. Alcune di queste sono, inoltre, di particolare interesse e potrebbero fortemente aumentare la varietà di simulazioni realizzabili.

- La prima di queste è la possibilità di inviare messaggi ad un singolo agente. Ciò richiede che gli agenti abbiano a disposizione un sistema di indirizzi ed un metodo per ottenere tali indirizzi. Una possibile soluzione potrebbe essere un sistema di *pagine gialle* a cui gli agenti possano registrarsi e richiedere gli indirizzi degli agenti con certe caratteristiche. Questa è la soluzione adottata dalla libreria JADE.
- Un altro utile strumento potrebbe essere un sistema di log su file flessibile e facilmente adattabile alle varie simulazioni. Idealmente, gli agenti potrebbero indicare a tale sistema quali variabili salvare e con quale frequenza. La complicazione principale di tale sistema è la necessità, intrinseca in tutto il programma, di essere generale. Questo comporta un grande numero di possibili formati di output da implementare.
- Un meccanismo di *azioni cicliche*, che automaticamente si prenotano nuovamente, potrebbe essere utile per l'implementazione di cicli di ricezione dei messaggi, specialmente in simulazioni dove la ricezione e la risposta ai messaggi vengono considerate due azioni separate.

Riferimenti bibliografici

Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275. ACM, 1996.