Alma Mater Studiorum · Università di Bologna

facoltà di scienze matematiche, fisiche e naturali

Corso di Laurea in Informatica

Progettazione, implementazione e valutazione di un meccanismo per la sincronizzazione bastato sul linguaggio Go

Tesi di Laurea in Reti di Calcolatori

Relatore:
Gabriele D'Angelo

Presentata da:
Pietro Ansaloni

Sessione II
Anno Accademico 2009/2010

Introduzione

L'Informatica ricopre un ruolo fondamentale nella società moderna sotto due punti di vista abbastanza diversi: come tecnologia è presente in gran parte degli strumenti di utilizzo comune, tanto che tutti siamo in qualche modo utenti di servizi informatici, mentre come scienza è di supporto o d'aiuto in diversi ambiti scientifici e non. La simulazione, in particolare, è una disciplina informatica di grande duttilità e può essere utilizzata per studiare il comportamento di qualunque sistema reale o ideale. La simulazione di un problema infatti è spesso il metodo migliore per comprenderlo a fondo e mostrarne la sua evoluzione temporale in relazione a diverse possibili condizioni.
Nonostante la grande variabilità degli ambiti di applicazione, in generale si può dire che una simulazione sia tanto più efficace quanto più in grado di completare la sua esecuzione nel minor tempo possibile.
La Simulazione Parallela nacque dall'idea di sfruttare sistemi multi-processore per effettuare simulazioni più efficienti dal punto di vista delle prestazioni.
In questo lavoro viene presentata un'implementazione per sistemi paralleli a memoria condivisa del Timewarp, uno dei più famosi algoritmi di sincronizzazione nell'ambito della Simulazione Parallela ad Eventi Discreti.
L'implementazione dell'algoritmo è stata realizzata usando il Go, un linguaggio di programmazione molto recente, che dedica un'attenzione particolare alla programmazione concorrente su sistemi multi-core.
Per la valutazione delle prestazioni dell'algoritmo è stato utilizzato un modello di benchmark realizzato in ambito accademico (Georgia Tech University),

chiamato PHOLD, che introduce un carico di lavoro sintetico nell'operazione di elaborazione degli eventi.
La fase di valutazione e analisi dei risultati mostra come le prestazioni ottenute dall'algoritmo siano buone per sistemi di carico sufficientemente pesanti da impegnare in maniera adeguata la simulazione. Inoltre si dimostrerà empiricamente come, in determinate condizioni, l'esecuzione più efficiente è quella distribuita su un numero di processi pari al numero di processori.
È stata infine realizzata una versione adattiva dell'algoritmo di simulazione che, adeguandosi alle condizioni dell'architettura disponibile per la simulazione, sceglie la migliore disposizione di processi possibile, dal punto di vista dell'efficienza, in maniera trasparente all'utente.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Nozioni di Base

1.1 Concorrenza e parallelismo

In Informatica, un sistema si definisce concorrente quando consente l'esecuzione simultanea di processi diversi, in grado di comunicare ed interagire tra loro. Studi sui processi concorrenti e soprattutto sulle problematiche relative alla loro sincronizzazione, risalgono già agli anni '60, ma è solo con la diffusione dei primi sistemi di calcolo paralleli negli anni '70 ed '80, che hanno trovato reale applicazione.
Si parla quindi di calcolo parallelo quando unità computazionali distinte ed indipendenti possono essere utilizzate simultaneamente per processare parti diverse di un problema, riducendo così il tempo di esecuzione.
Inizialmente il calcolo parallelo [1] ha trovato applicazione principalmente in settori come il calcolo ad alte prestazioni (High Performance Computing[1]), a causa degli alti costi associati alla produzione di calcolatori paralleli e alla complessità della realizzazione di algoritmi paralleli efficienti.
Inoltre i settori più tradizionali dell'informatica potevano ottenere una ridu-

[1]High Performance Computing è un ambito dell'informatica che usa supercomputer (calcolatori dotati di un alto grado di parallelismo e potenza di calcolo) o cluster (sistemi composti da diversi calcolatori, connessi da una rete locale, che cooperano in maniera parallela per aumentare la velocità di calcolo) per risolvere problemi di computazione avanzata.

zione dei tempi di calcolo senza sforzo, grazie agli effetti della Legge di Moore [2]: *le prestazioni dei processori e il numero di transistor ad essi relativi raddoppiano ogni due anni.*

Gordon Moore, cofondatore di Intel, teorizzò che il numero di componenti elettronici (transistor) all'interno di un chip potesse crescere, col passare del tempo, in maniera esponenziale, consentendo di produrre processori più veloci, mantenendo costante il loro prezzo.

Il parallelismo fu per anni relegato ad ambiti informatici di nicchia a causa del grande incremento di prestazioni garantito dai miglioramenti dall'hardware, che consentivano di aumentare la velocità di calcolo in maniera esponenziale, mantenendo il modello di programmazione sequenziale.

Tutto ciò ha avuto termine, in tempi recenti, quando si è raggiunto il limite della frequenza dei processori, rallentando pesantemente la crescita avuta nei decenni precedenti[2].

Il limite della frequenza di clock dipende dal modo in cui i componenti sono assemblati all'interno del chip ed è quindi di natura fisica, per cui difficilmente potrà essere risolto, almeno in tempi brevi.

D'altra parte, la legge di Moore continua a dimostrarsi valida e si è pensato di utilizzare la maggiore capacità di integrazione disponibile per inserire più unità computazionali nello stesso chip, realizzando processori multi-core.

Quindi, mentre fino al decennio scorso per migliorare le prestazioni di un programma era sufficiente aspettare che fossero disponibili nuovi processori più potenti, ora ciò non è più possibile, perché la frequenza di clock dei processori non potrà più aumentare come in passato: il miglioramento delle prestazioni di un programma dipende quindi dalla sua capacità di sfruttare il parallelismo fornito dai diversi core del processore.

Per quantificare il guadagno di tempo ottenuto con una parallelizzazione si misura un valore, detto *speedup*, che indica la scalabilità di un programma.

[2]Nel 2005, l' International Technology Roadmap for Semiconductors [3] predisse che la frequenza di clock dei microprocessori avrebbe superato i 10 GHZ nel 2008, raggiungendo i 15 GHZ nel 2010. Si noti invece che i processori prodotti da Intel sono attualmente molto distanti da quelli attesi nel 2007 secondo le predizioni più pessimistiche.

Lo speedup $S(p)$ relativo ad un'esecuzione parallela su p processori è

$$S(p) = \frac{T(1)}{T(p)}$$

dove $T(p)$ indica il tempo di esecuzione di un programma su p processori e $T(1)$ indica il tempo dell'esecuzione sequenziale dello stesso programma.
Il valore dello speedup di un'esecuzione sequenziale è sempre pari a 1, mentre quello di un'esecuzione su p processori è tanto migliore quanto più si avvicina al valore p, che, in condizioni normali, ne è un limite superiore[3].
La misura di quanto lo speedup si avvicina al limite ideale è indicata con il termine *efficienza*, che è il rapporto tra lo speedup $S(p)$ e il numero di processori p:

$$E(p) = \frac{S(p)}{p}$$

L'efficienza di un'esecuzione sequenziale è sempre pari a 1, valore che ne rappresenta il limite superiore.
L'efficienza è quindi una misura efficace per quantificare quanto tempo viene perso nel parallelizzare: più il valore dell'efficienza si discosta dal valore 1, maggiore sarà il tempo di esecuzione speso in operazioni che non vengono effettuate nell'esecuzione sequenziale.

Stabilito che lo scopo per cui utilizzare il parallelismo è diminuire il tempo di calcolo, è necessario valutare a quanto ammonti l'effettivo guadagno prestazionale ottenuto dalla parallelizzazione di un programma sequenziale.
La legge di Amdahl [4] afferma che lo speedup di una computazione parallela non può crescere in maniera indefinita, perchè è limitato dal tempo necessario per eseguire la frazione sequenziale del programma.
Sia f la frazione parallelizzabile di un programma e $1-f$ la sua parte sequenziale. Sia $T(1)$ il tempo necessario per effettuare la computazione sequenziale

[3]In rari casi possono verificarsi situazioni particolari per cui l'esecuzione sequenziale viene rallentata da fattori esterni alla computazione, che non si presentano dividendo l'eseuczione in più componenti parallele. In questi casi è quindi possibile che lo speedup di un'esecuzione su p processori superi il valore p.

del suddetto programma, ottenuto come $T(1) = T(1)(f + (1 - f))$.
Una esecuzione parallela dello stesso programma su p processori impiega un tempo

$$T(p) = T(1)\left(\frac{f}{p} + (1 - f)\right)$$

Lo speedup di questa esecuzione è

$$S(p) = \frac{1}{\frac{f}{p} + (1 - f)}$$

Cercando il limite dello speedup per $p \to \infty$ si ottiene un limite teorico per lo speedup:

$$\lim_{p\to\infty} S(p) = \lim_{p\to\infty} \frac{1}{\frac{f}{p} + (1 - f)} = \frac{1}{1 - f}$$

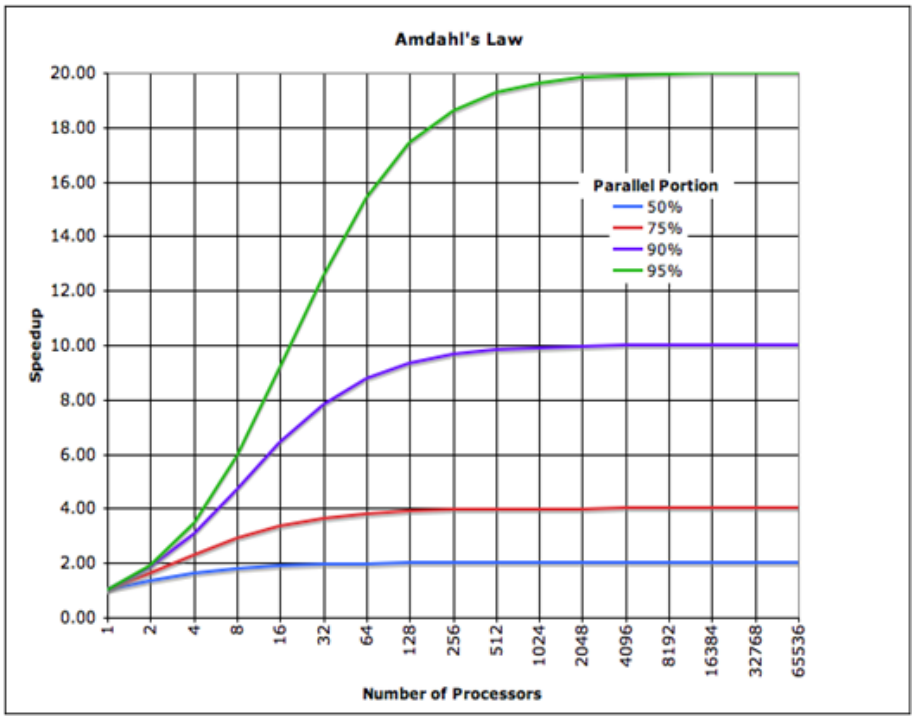

Figura 1.1: Legge di Amdahl

La legge di Amdahl mostra uno dei problemi fondamentali della computazione parallela: nella maggior parte dei casi un programma contiene al suo interno una frazione non parallelizzabile, per la quale i diversi processori devono sincronizzarsi. Per quanto piccola sia questa sezione sequenziale, al crescere del numero di processori, il miglioramento ottenuto da una parallelizzazione tenderà a calare.
Questa situazione si presenta in particolare quando i diversi processori devono sincronizzarsi per l'accesso ad elementi condivisi, per cui diventa necessario l'uso di regioni critiche.

1.2 Simulazione

Per Simulazione si intende quel settore dell'Informatica che si occupa di progettare e realizzare un modello in grado di imitare il comportamento di un sistema reale nella sua evoluzione temporale.
Teoricamente qualsiasi sistema reale o ideale può essere oggetto di una simulazione e per questo esistono diversi schemi di simulazione: ad eventi discreti, per processi, per attività.
Dall'inizio degli anni '80, con la diffusione di sistemi paralleli, anche nell'ambito della simulazione si iniziò a dedicare attenzione ad algoritmi paralleli, con lo scopo di ridurre il tempo di calcolo: nacque così la Simulazione Parallela e Distribuita.
Lo schema più utilizzato è quello ad eventi discreti, che in ambiente parallelo ha quindi dato origine alla Simulazione Parallela ad Eventi Discreti (PDES).
In questo ambito il modello simulato deve essere suddiviso in più componenti, chiamati Logical Process (LP), ciascuno dei quali viene eseguito in maniera parallela ed indipendente dagli altri, per quanto consentito dalle specifiche del modello stesso.
I LP comunicano tra loro scambiandosi messaggi.
La simulazione è guidata dagli eventi, che rappresentano qualunque tipo di operazione, avvenimento o cambiamento nello stato del sistema. Ognuno dei

quali può essere processato da un solo LP, quello responsabile dell'evento in questione, che non necessariamente coincide con quello creatore.
Ogni evento ha quindi 2 caratteristiche fondamentali: un LP di riferimento e un timestamp che indica l'istante temporale in cui deve essere processato.
Tutti i LP rispettano il *Vincolo di Causalità Locale* (VCL), cioè processano gli eventi che li riguardano in ordine cronologico, non decrescente rispetto al timestamp: questa condizione è sufficiente per garantire che la causalità degli eventi sia rispettata.
Nella simulazione il tempo degli eventi non ha niente a che fare con il tempo reale, ma è un tempo logico, simulato.

1.3 Il concetto di tempo nel parallelismo

In ambiente parallelo i processi sono entità logicamente separate che comunicano scambiandosi messaggi, il cui ritardo di trasmissione potrebbe anche essere non trascurabile ai fini del risultato dell'esecuzione.
Secondo il punto di vista canonico l'ordinamento temporale tra gli eventi[4] che occorrono nel sistema dipende da un orologio *reale* e corrisponde alla comune nozione *A è avvenuto prima di B*.
Tuttavia, in ambiente parallelo, non è sempre possibile garantire che gli orologi reali dei diversi processi siano perfettamente sincronizzati, quindi, dati due eventi, non è sempre possibile determinare quale è avvenuto prima e stabilire quindi un ordinamento totale tra tutti gli eventi del sistema. Per questo è necessario un meccanismo che stabilisca un ordinamento tra eventi senza fare ricorso ad orologi fisici, ma utilizzando un'astrazione del tempo realizzata tramite *orologi logici* [5].
La prima considerazione da fare riguarda processi che non interagiscono tra loro nell'esecuzione e che sono quindi indipendenti: i loro orologi non han-

[4]In questa sezione il termine evento non ha nulla a che fare con gli eventi discreti della PDES, ma indica un'operazione qualunque effettuata da un processo in un sistema parallelo.

no alcuna necessità di essere sincronizzati ed i processi sono quindi liberi di operare in modo concorrente senza alcun rischio di interferenze.
Per processi non indipendenti invece, è necessario garantire che concordino sull'ordinamento degli eventi, anche se non è necessario che condividano in ogni momento la stessa nozione di tempo reale.
Assumiamo che gli eventi di ogni processo siano totalmente ordinati (questa condizione non è restrittiva) e che l'invio e la ricezione di un messaggio siano considerati come eventi di un processo.
È possibile allora definire una relazione di ordinamento tra eventi A $\Rightarrow$ B che letteralmente significa *A è avvenuto prima di B*. La relazione $\Rightarrow$ sull'insieme di eventi di un sistema è la minima relazione che soddisfa le seguenti condizioni:

- se A e B appartengono allo stesso processo e A viene prima di B, allora A $\Rightarrow$ B;
- se A è la spedizione di un messaggio e B è la ricezione dello stesso da parte di un altro processo, allora A $\Rightarrow$ B;
- se A $\Rightarrow$ B e B $\Rightarrow$ C, allora A $\Rightarrow$ C, cioè vale la proprietà transitiva;
- non vale la proprietà riflessiva, quindi A $\not\Rightarrow$ A.

Quindi la relazione $\Rightarrow$ è un ordinamento parziale non riflessivo dell'insieme di tutti gli eventi del sistema e stabilisce che due eventi siano considerati concorrenti se $A \not\Rightarrow B$ e $B \not\Rightarrow A$.
Si stabilisce che ogni processo P_i abbia un proprio orologio logico C_i, che ha il compito di assegnare un tempo logico C_i(A) ad ogni evento A del processo. Dati due eventi A e B, si può stabilire la condizione per cui se $A \Rightarrow B$ allora $C(A) < C(B)$.
L'ultima affermazione è valida sia nel caso locale, in cui A e B sono eventi dello stesso processo, sia nel caso in cui rappresentano spedizione e ricezione di un messaggio.
A questo punto, ricorrendo all'astrazione appena formalizzata, è possibile

fare a meno degli orologi fisici per rappresentare il tempo degli eventi che coinvolgono i processi paralleli.

Applicando l'astrazione appena descritta alla simulazione, si vuole sottolineare come il concetto di tempo logico sia utilizzato per realizzare il tempo discreto degli eventi simulati.
In una simulazione esistono però diverse concezioni di tempo:

- tempo fisico: si riferisce al tempo del sistema modellato dalla simulazione;
- tempo simulato: è il tempo logico assegnato agli eventi che compaiono nella simulazione, astrae dal tempo fisico ma ne è una rappresentazione sensata e matematicamente corretta;
- wallclock time: si riferisce al tempo reale durante l'esecuzione della simulazione.

Capitolo 2

Simulazione Parallela ad Eventi Discreti

In ambito PDES la simulazione è condotta dai Logical Process (LP), che fanno avanzare il tempo simulato gestendo gli eventi che li riguardano.
Come già detto, la condizione sufficiente per un'esecuzione causalmente corretta in ambiente parallelo è che il vincolo di causalità locale (VCL) non sia violato: questa condizione garantisce che il risultato di una simulazione parallela sia identico rispetto a quello di una equivalente esecuzione sequenziale, ma è la causa di uno dei principali problemi in PDES, chiamato *problema della sincronizzazione* [6]. Questo è dovuto al fatto che gli eventi che un LP deve processare non sono tutti locali, ma provengono in parte da altri processi.
Per evitare la violazione del VCL ogni LP dovrebbe processare un evento solo se è certo che in futuro non potrà riceverne altri dall'esterno con timestamp antecedente. Eventi di questo tipo sono detti *safe*.
In letteratura esistono diversi algoritmi che si occupano di risolvere il problema della sincronizzazione e possono essere raggruppati in 2 famiglie, che utilizzano come soluzione idee diametralmente opposte.

2.1 Approccio Conservativo

Il primo gruppo di algoritmi prende il nome dall'approccio utilizzato per la soluzione del problema della sincronizzazione, che è di tipo conservativo [6].
L'idea di base è la più semplice ed intuitiva: per evitare la violazione del VCL è sufficiente processare soltanto gli eventi *safe* e, se non ce ne sono, aspettare finché non si hanno informazioni sufficienti per stabilire quali eventi possono essere processati in sicurezza.
Gli algoritmi con approccio conservativo hanno bisogno che il sistema di simulazione garantisca una serie di condizioni, senza le quali non è possibile determinare se un evento è safe o meno:

- il numero di LP e i canali di comunicazione esistenti tra di essi sono definiti inizialmente e non possono essere modificati durante la simulazione.

- Ogni LP spedisce messaggi[1] ordinati rispetto al timestamp agli altri processi.

- La rete di comunicazione garantisce affidabilità e preserva l'ordinamento dei messaggi.

- Il meccanismo di spedizione di un evento da un LP ad un altro prevede che il tempo simulato di ricezione sia strettamente maggiore di quello di invio: la differenza di tempo (simulato) tra i due eventi viene definita *lookahead*, ed è solitamente una caratteristica intrinseca del sistema reale.

Da queste assunzioni si può ricavare che l'ultimo messaggio ricevuto tramite un determinato canale è un limite inferiore del timestamp dei messaggi che è possibile ricevere dal relativo LP.

[1] Nel corso del lavoro si utilizzeranno i termini evento e messaggio per indicare un evento spedito da un processo ad un altro.

È quindi logico pensare che, per uno specifico LP P_a, un evento E_i con timestamp T_i può essere considerato safe solamente se, $\forall$ LP P_k, con $P_k \neq P_a$, è stato ricevuto almeno un evento E_j con tempo T_j e vale $T_i > T_j$.
Ogni LP può quindi processare solo gli eventi che sono considerati safe, ma in caso non ce ne siano, non può limitarsi ad aspettare indefinitamente, perché c'è il rischio che la simulazione vada in deadlock, una situazione di stallo nella quale nessuno dei LP può processare un evento e si viene a creare una condizione di attesa circolare: ogni processo aspetta che uno degli altri compia un'azione (processare un evento o inviare un messaggio) e nessuno è autorizzato a procedere nella computazione.
Poiché non ci sono eventi esterni che possono sbloccare la situazione o effettuare un'operazione di recovery è necessario prevenire la possibilità che si verifichino casi di deadlock.
Per evitare che si verifichi questa condizione ogni LP, quando non ha più eventi safe da processare, invia dei messaggi vuoti (null message) agli altri LP con cui è connesso, con lo scopo di comunicare il timestamp minimo dei possibili eventi che eventualmente invierà agli altri processi [7].
In questo modo si impedisce il verificarsi della condizione di attesa circolare e quindi del deadlock. Si noti come il meccanismo di invio dei null message sia interamente basato sul concetto di lookahead: fissato questo valore a L, un LP può processare un evento con tempo T_i solo se è sicuro che non arriveranno eventi dagli altri LP con tempo $T_j < T_i$, quindi se sono arrivati a processare eventi con timestamp $T_j - L$; solo in questo modo i LP possono avanzare il proprio tempo simulato rimanendo in uno stato safe.
Se non ci fosse il lookaheahd, o se fosse nullo, si eviterebbe il deadlock perché i LP dovrebbero comunque gestire invio e ricezione dei null message, ma il sistema sarebbe in una condizione di attesa attiva, in cui tutti i processi sono impegnati ma non stanno effettivamente procedendo nella computazione (livelock): una volta entrati in una fase della simulazione in cui sarebbe necessario l'invio dei null message per procedere nella computazione, ogni LP infatti non potrebbe dichiarare nessun evento safe, perché i null message in

arrivo dagli altri LP non consentirebbero di avanzare il tempo simulato.
La dipendenza dal lookahead è il principale aspetto negativo dell'approccio conservativo: più questo valore è basso, maggiore sarà il numero di messaggi vuoti che è necessario inviare per arrivare al termine della simulazione, maggiore sarà il tempo reale speso dai LP per occuparsi della comunicazione, a discapito della computazione.

2.2 Approccio Ottimistico

La seconda tipologia di algoritmi di sincronizzazione nacque come soluzione alternativa a quella conservativa [6], un meccanismo nel quale la cpu rimane inattiva per la maggior parte del tempo, in attesa di messaggi dagli altri processi.
Il punto debole di questo meccanismo è che generalmente i tempi di elaborazione sono molto più veloci di quelli di comunicazione, quindi un meccanismo efficiente potrebbe impiegare in maniera costruttiva il tempo speso nell'aspettare messaggi dagli altri processi, cosa che non avviene nell'approccio conservativo.
L'idea su cui si fonda l'approccio ottimistico è quindi lasciare i processori inattivi per il minor tempo possibile, visto che le operazioni di computazione sono molto veloci, se confrontate con quelle di comunicazione.
Ogni LP può quindi procedere nella simulazione processando gli eventi di cui dispone in ordine di timestamp, senza preoccuparsi se questi sono safe o meno. Quando un LP riceve dall'esterno un evento che ha timestamp anteriore rispetto al suo tempo simulato, viene riscontrata una violazione del VCL e il sistema deve effettuare un'operazione di recovery per ritornare in uno stato in cui ogni evento, compreso quello trasgressore, può essere processato rispettando il VCL.
L'unica caratteristica che il sistema deve avere è che i canali di comunicazione garantiscano affidabilità: non è consentita la perdita di messaggi, ma non è strettamente necessario che l'ordine di arrivo sia uguale a quello di spedi-

zione.
L'algoritmo principale per questa categoria è il Timewarp, ideato da David R. Jefferson [8].
Per far sì che i LP possano tornare indietro, l'algoritmo prevede che ognuno di essi mantenga informazioni sugli eventi processati, sui messaggi inviati e sugli stati locali, in modo da poter ripristinare la situazione del LP, se necessario, nelle esatte condizioni in cui si trovava ad un determinato tempo simulato.
La violazione del VCL è riparata con una operazione chiamata rollback, che consiste in un "riavvolgimento" del tempo simulato: tutte le operazioni compiute dal LP devono essere annullate, per tornare ad uno stato in cui l'evento trasgressore può essere processato in maniera causalmente corretta.
Quando un LP riceve un evento trasgressore con timestamp T_t tutti gli eventi da lui processati ad un tempo $T_i > T_t$ devono essere annullati.
Gli eventi locali, che coinvolgono solo il LP in questione, possono essere annullati eseguendo le operazioni inverse rispetto a quelle svolte per processarlo, eliminando l'evento dall'insieme di quelli processati e reinserendolo tra gli eventi ancora da elaborare.
Per quanto riguarda gli eventi non locali, come l'invio di un messaggio ad un altro LP, la situazione è un po' più complessa e per questo è stato introdotto il concetto di anti-messaggio, che consiste nell'opposto logico di un messaggio. Per annullare l'operazione di spedizione di un messaggio, un LP deve quindi inviare il relativo anti-messaggio.
Quando un messaggio incontra il suo opposto viene effettuata un'operazione di annichilimento, che consiste nella cancellazione di entrambi. Il LP che deve effettuare questa operazione deve comportarsi diversamente in base alla situazione in cui si trova l'evento da cancellare:

- Se l'evento non è ancora stato ricevuto non è possibile effettuare l'annichilimento: l'anti-messaggio deve essere conservato per svolgere la cancellazione quando si riceverà il relativo evento.

- Se l'evento è stato ricevuto, ma non ancora processato, basta eseguire

l'annichilimento cancellando l'evento dall'insieme di quelli futuri.

- Se l'evento è già stato processato è necessario svolgere un rollback ad un tempo precedente all'elaborazione del messaggio, per poi effettuare la cancellazione di entrambi.

Un LP, nell'effettuare un'operazione di rollback, si può trovare nella condizione di dover inviare anti-messaggi, che a loro volta potrebbero essere causa di rollback negli altri LP.
In conclusione, è possibile che la ricezione di un messaggio che trasgredisce al VCL provochi una catena di rollback che potrebbe coinvolgere, nel caso peggiore, tutti i LP.
Si vuole sottolineare che la cascata di rollback, anche nel caso peggiore, non sarà mai in grado di far regredire la simulazione in maniera indeterminata. Infatti, con la ricezione di un evento trasgressore di timestamp T, i LP potranno tornare indietro al massimo ad un tempo $T_i \geq T$.
Il Timewarp ha bisogno di un meccanismo di controllo globale che stabilisca un limite inferiore al di là del quale non è possibile tornare tramite operazioni di rollback, per poter effettuare operazioni irrevocabili (tipicamente di I/O) e per liberare parte della memoria occupata dalla strutture dati di supporto. Questo limite inferiore prende il nome di Global Virtual Time (GVT) [9] e consiste nel minimo timestamp di ogni messaggio o anti-messaggio, parzialmente o totalmente processato, all'interno del sistema in un dato momento. Calcolare il GVT sarebbe un'operazione banale se fosse possibile avere una visione globale della simulazione, ma essendo il Timewarp un algoritmo per la sincronizzazione in ambiente parallelo, questo non è possibile.
Anche se esistono diversi algoritmi di calcolo, lo schema generale è il seguente:

- ogni LP riceve un messaggio che comunica l'inizio del procedimento di valutazione;
- ogni LP calcola il proprio minimo locale;
- si calcola il minimo globale a partire dai minimi locali calcolati dai LP.

Nel procedimento di valutazione del GVT bisogna tenere conto anche dei messaggi transienti, cioè di quelli che durante l'operazione di calcolo sono in viaggio da un LP all'altro. L'unica soluzione per questo problema è fare in modo che ogni evento nel sistema sia tenuto in considerazione da almeno un LP.
Questo procedimento non è comunque sufficiente per garantire l'esattezza del valore calcolato, a causa del *simultaneous reporting problem*: se due processi ricevono la richiesta di calcolo del minimo locale in tempi (reali) diversi, può capitare che si perda un evento che è in viaggio tra i due, perché entrambi i LP ritengono che sia competenza dell'altro.
Per ovviare a questo problema sono stati proposti diversi metodi; in questo lavoro è stato utilizzato l'algoritmo di Samadi per il calcolo del GVT [9], che prevede l'invio di acknowledgement (ack) per confermare la ricezione di un messaggio: finché un LP non riceve la conferma di ricezione per un messaggio, deve conteggiarlo nel calcolo del GVT.
Inoltre il simultaneous reporting problem viene risolto marcando gli ack inviati dopo aver comunicato il minimo locale: in questo modo se un processo riceve un acknowledgement marcato deve tener conto del relativo messaggio nella valutazione del minimo locale.

Capitolo 3

Linguaggio Go

3.1 Caratteristiche principali

Il Go [10] è un linguaggio general-purpose molto recente proposto da Google nel Novembre 2009: è un progetto open-source, nato con l'intento di produrre un linguaggio moderno, di semplice utilizzo ma dalle prestazioni simili a quelle degli altri linguaggi compilati.
Secondo Google un nuovo linguaggio di programmazione è necessario per adattarsi al mondo dell'informatica che nell'ultimo decennio è cambiato in maniera profonda, mentre i linguaggi di programmazione sono rimasti sostanzialmente gli stessi. Queste sono le principali tendenze riscontrate:

- enorme incremento nelle prestazioni dei computer, rapportato a tempistiche di sviluppo del software sostanzialmente invariate.
- Impatto significativo della gestione delle dipendenze nell'ambito dello sviluppo del software; i tipici header file dei linguaggi C-like sono in contrasto con un'analisi delle dipendenze pulita e una compilazione veloce.
- Rifiuto progressivo degli utenti nei confronti dei pesanti sistemi di tipi come quelli di Java e C++, che hanno spinto gli utenti verso linguaggi dinamicamente tipati come Python e JavaScript.

- Inadeguato supporto, da parte dei principali linguaggi di programmazione, di concetti fondamentali come garbage collection e computazione parallela.

- Emergere dei sistemi multicore, generalmente visto come un problema e non come un'opportunità.

Per rispondere a queste esigenze gli sviluppatori del Go hanno pensato valesse la pena produrre un nuovo linguaggio, seguendo queste linee guida:

- compilazione veloce, anche per programmi di grandi dimensioni.

- Un modello per la costruzione del software che renda veloce l'analisi delle dipendenze, evitando buona parte dell'overhead prodotto dalle operazioni di inclusione degli headers dei linguaggi C-like.

- Nessuna gerarchia di tipi: in questo modo, nella compilazione, si evitano le perdite di tempo prodotte definendo le relazioni che intercorrono tra tipi diversi. Inoltre, sebbene il sistema di tipi sia statico, il linguaggio deve renderlo più leggero rispetto ai tipici linguaggi di programmazione object oriented.

- Nessun meccanismo di deallocazione esplicita: la memoria non utilizzata viene recuperata tramite garbage collection.

- Necessità di provvedere un supporto per l'esecuzione concorrente e la comunicazione tra processi, ritenuto fondamentale per la realizzazione di sistemi software su macchine multi-core.

In conclusione il Go è un tentativo di combinare la facilità di programmazione dei linguaggi interpretati e dinamicamente tipati con l'efficienza e la stabilità dei linguaggi compilati e staticamente tipati. Mira inoltre ad essere un linguaggio moderno, che garantisca una garbage collection efficiente, fornisca supporto per la computazione multi-core e sia veloce nella compilazione.

La sintassi di base del Go è simile a quella del C, con qualche concetto preso dalla famiglia di linguaggi Pascal/Modula/Oberon. Per quanto riguarda la concorrenza sono state utilizzate idee provenienti dai linguaggi ispirati dal CSP di Hoare [11], come Newsqueak e Limbo.

3.2 Programmazione concorrente

Nella definizione del linguaggio è chiaramente espressa l'intenzione di dedicare una particolare attenzione alla programmazione concorrente effettuata in ambiente parallelo, sfruttando l'architettura multi-core ampiamente diffusa al giorno d'oggi.
Il parallelismo è realizzato tramite goroutines: particolari funzioni indipendenti tra loro che possono essere eseguite concorrentemente su processori diversi (se possibile), ma che utilizzano lo stesso spazio degli indirizzi e possono quindi comunicare tramite memoria condivisa.
Qualunque funzione può essere eseguita come goroutine, anteponendo il comando `go` alla chiamata: in questo modo la funzione viene lanciata in maniera asincrona e indipendente dal chiamante.
L'idea di base che sta dietro al parallelismo delle goroutines è quella di distribuire queste particolari funzioni su un certo set di thread del sistema operativo: quando una di queste si blocca (per una system call per esempio) ne viene caricata un'altra attiva, cercando di utilizzare al massimo la potenza di calcolo della cpu e di non lasciarla mai inutilizzata.
Quello che gli sviluppatori del linguaggio vogliono mettere in evidenza è che tutto questo viene realizzato in maniera trasparente al programmatore, che deve soltanto preoccuparsi di gestire le chiamate ad alto livello.
Una variabile d'ambiente, `GOMAXPROCS`, definisce il numero di processori che possono essere utilizzati simultaneamente, garantendo la possibilità di lanciare funzioni concorrenti su processori diversi in parallelismo reale. Viene però fatto notare che lanciare diverse goroutines non sempre garantisce prestazioni migliori: se il numero di funzioni concorrenti è superiore al numero di pro-

cessori a disposizione, ci si può ragionevolmente aspettare un lieve degrado delle prestazioni.
Il Go fornisce un meccanismo di comunicazione di alto livello (chan), che consente a due funzioni diverse di scambiarsi informazioni e, in caso di chiamata bloccante, di sincronizzarsi.
Il chan deriva direttamente da uno dei modelli più funzionali per la comunicazione tra processi concorrenti: il linguaggio formale CSP (Communicating Sequential Processes), ideato da Hoare [11].
Nelle specifiche del Go viene espressamente consigliato di gestire la sincronizzazione tra goroutines comunicando tramite i chan: di default l'invio e la ricezione di un messaggio attraverso un chan sono chiamate sincrone, quindi una receive si blocca finché non viene effettuata una send sul relativo chan e viceversa.

Il Go definisce quindi un meccanismo ad alto livello per la programmazione concorrente: i processi possono essere realizzati tramite goroutines ed eseguiti in parallelo su processori diversi, se l'architettura lo consente, mentre la comunicazione tra di esse può essere effettuata utilizzando chan condivisi tra queste. Viene naturale pensare ad una possibile implementazione di una simulazione parallela utilizzando questi meccanismi.

3.3 Test sullo speedup: confronto con C/OpenMP

Per verificare l'effettiva efficienza del Go sono state effettuate una serie di prove con un programma di tipo strongly cpu-bound, cioè senza comunicazione ma con forte utilizzo del processore: in particolare è stato implementato un metodo Monte Carlo per la stima di π [12], che risulta particolarmente funzionale agli scopi, perché è facilmente scalabile e ha come unico compito quello di eseguire calcoli, per cui si può studiare il suo comportamento al variare del numero di processori utilizzati.
L'algoritmo prevede che vengano lanciati un certo numero (parametrizzabile

al momento della chiamata) di processi concorrenti, ognuno dei quali genera pseudo-casualmente una serie di numeri nell'intervallo [-1,1], che vengono a due a due interpretati come coordinate in un piano cartesiano.
Per ogni punto generato, si controlla la posizione rispetto alla circonferenza goniometrica[1] e si conta il numero di punti che cadono all'interno di essa.
Il numero di punti caduti all'interno della circonferenza fornisce una stima del valore della sua area A_c, in relazione al numero totale di punti lanciati, che rappresenta una stima di A_q, area del quadrato di raggio 2 centrato nell'origine.
Quindi, definiti $A_q = (2r)^2 = 4r^2$ e $A_c = \pi r^2$, è possibile ricavare il valore

$$\pi = \frac{4A_c}{A_q}$$

Al termine della computazione è quindi sufficiente contare il numero totale di punti e stabilire quanti di essi sono caduti all'interno della circonferenza. Si può così calcolare una stima di π:

$$\pi \simeq \frac{4count}{tot}$$

Dove *count* rappresenta il numero di punti caduti all'interno della circonferenza, e $tot = nproc * np$, con *nproc* che indica il numero di processi e *np* il numero di punti lanciati da ognuno di essi.

I test sono stati eseguiti su una macchina del cluster di simulazione dell'Università di Bologna: chernobog, una macchina a 64 bit con processore quad-core Intel Xeon.
Per avere un metro di giudizio sulle prestazioni del Go sono state svolte le stesse prove con un analogo test scritto in C che utilizza OpenMP per le chiamate di gestione dei thread.

[1] Per circonferenza goniometrica si intende una circonferenza nello spazio cartesiano con centro nell'origine e raggio di dimensione 1. L'area di tale circonferenza vale $A = \pi r^2 = \pi$.

Threads	1	2	3	4
Speedup C/OpenMP	1.000	1.985	2.951	3.874
Speedup Go	1.000	1.987	2.927	3.788

Tabella 3.1: Stima di π, confronto Go - C/OpenMP

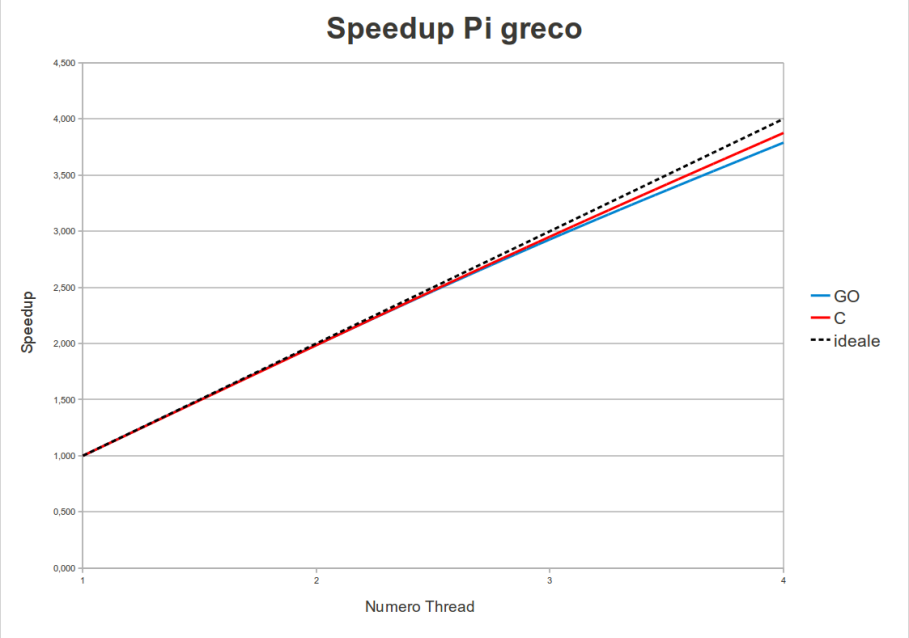

Figura 3.1: Stima di π, confronto GO - C/OpenMP

Il valori da confrontare sono stati ottenuti lanciando ripetutamente ognuno dei due test, per poi calcolare la media, dalla quale sono stati ricavati i valori dello speedup.
Come si può vedere da grafico e tabella, i valori del C e quelli del Go sono molto simili tra loro, ed entrambi si avvicinano parecchio ai valori ideali di scalabilità.
Questo significa che il Go, anche se un po' più lento del C, fa registrare ottimi valori per quanto riguarda lo speedup, cioè l'indice di miglioramento prestazionale che si ottiene in una simulazione parallela rispetto ad una sequenziale, e può essere quindi considerato, sotto questo aspetto, al livello dei

migliori linguaggi di programmazione.
È stata effettuata una ulteriore prova, sfruttando il test per la stima di π, per confrontare i tempi medi di esecuzione al variare del numero di processi in un'esecuzione su un solo processore.
Sono quindi stati lanciati nuovamente gli stessi test in Go rispetto al caso precedente, tranne per il fatto che la variabile d'ambiente `GOMAXPROCS` è stata mantenuta costantemente pari a 1.

N^o processi	1	2	3	4
Speedup	1.000	0.995	0.987	0.982

Tabella 3.2: Stima di π, speedup con esecuzione monoprocessore

Come prevedibile i valori ottenuti dalla simulazione eseguita su un solo processore non migliorano all'aumentare del numero di processi utilizzati. La tabella mostra come lo speedup si mantenga praticamente costante, il che indica che non c'è stato alcun incremento di prestazioni dovuto all'uso di più goroutines. Questa è quindi una controprova che nel caso precedente i miglioramenti prestazionali non sono dovuti all'aumento del numero di processi utilizzati, ma alla loro esecuzione parallela.

Capitolo 4

Implementazione del Timewarp

L'algoritmo Timewarp non rappresenta l'intera simulazione, ma fornisce il supporto necessario per realizzarla: una simulazione infatti ha lo scopo di imitare il comportamento di un sistema qualsiasi nella sua evoluzione temporale.
Per consentire alla simulazione di adattarsi alle caratteristiche del sistema da riprodurre, la si può immaginare divisa in due parti distinte:

- modello di simulazione, che rappresenta il sistema da simulare. In particolare dovrà occuparsi dell'elaborazione degli eventi per reagire a ciascuno di essi in maniera adeguata al sistema imitato.

- Meccanismo di simulazione, che fornisce il supporto per la gestione del modello. Questa parte sarà diversa in base al tipo di algoritmo utilizzato e servirà per lo svolgimento della simulazione, nascondendo al modello i dettagli dell'algoritmo.

In questo capitolo viene descritta esclusivamente l'implementazione del meccanismo di simulazione, basato sull'algoritmo Timewarp, mentre quella del modello sarà trattata nel Capitolo 6.

4.1 Descrizione variabili e strutture dati

Le principali strutture dati implementate nella realizzazione del Timewarp sono

- `Event`: rappresenta gli eventi del sistema. È contraddistinto da un identificatore univoco all'interno del sistema `Id`, un timestamp `Time`, che rappresenta il tempo (simulato) in cui dovrà essere processato l'evento, e un'ulteriore struttura a disposizione del modello, che permette di diversificare gli eventi di cui il sistema necessita.

- `Message`: consente l'invio di eventi da un LP all'altro. Contiene un `Event`, un mittente e un destinatario.

- `Heap`: è una coda con priorità realizzata tramite heap[1] minimo, nel quale ogni nodo è un array di eventi ed è quindi possibile memorizzare eventi contemporanei nello stesso nodo. In questo modo la gestione dello heap risulta molto più semplice, poichè non è possibile trovare nodi con lo stesso timestamp. Questa struttura è utilizzata da ogni LP per contenere gli eventi futuri.

- `List`: è una lista "bilinkata" (double linked), nel quale ogni nodo è collegato sia al precedente che al successivo. La struttura è realizzata sfruttando il tipo `list` del Go ed è mantenuta ordinata per migliorare l'efficienza dell'operazione di ricerca.

- `LocalState`: è una struttura locale ad ogni LP che contiene le informazioni rilevanti a proposito del processo. Ogni struttura contiene

  - il tempo simulato attuale del processo;
  - un identificatore univoco per il LP;

[1] Gli elementi di una coda con priorità possono essere disposti in un vettore, detto heap, che può essere interpretato come albero binario. Le particolari proprietà con cui gli elementi della coda son disposti sui nodi dell'albero rendono relativamente veloci le operazioni di inserimento ed estrazione del minimo.

  - una struttura `Heap` contenente gli eventi ancora da processare;
  - una `List` contenente gli eventi processati, per poterli reinserire tra quelli ancora da processare in caso di rollback;
  - una `List` per i messaggi inviati agli altri LP, per poter generare i relativi anti-messaggi ed inviarli, in caso sia necessario annullare l'operazione di spedizione;
  - una `List` per gli anti-messaggi per i quali non è ancora stato ricevuto il relativo messaggio, nel quale vengono salvati in attesa di poter effettuare l'annichilimento;
  - una `List` che contiene i messaggi in uscita, cioè quelli che sono stati spediti ma per cui non è ancora stato ricevuto l'acknowledgement. Questa struttura è utile nel calcolo del GVT per tenere conto dei messaggi transienti.

Ci sono inoltre variabili condivise tra tutti i LP:

- `Lpnum`, che indica il numero di LP.
- `EndTime`, che indica il tempo in cui la simulazione deve terminare.
- `State[]`, un array di dimensione `Lpnum` che contiene informazioni sullo stato pubblico dei LP. Un LP può trovarsi infatti in diversi stati:
  - `running`, cioè funzionante in maniera corretta;
  - `stopped`, dopo aver completato la simulazione;
  - `idle`, se ha raggiunto il tempo simulato `EndTime` ma non ha ancora terminato (questa situazione si verifica se ci sono altri LP ancora `running`);
  - `evaluating`, se sta effettuando il calcolo del GVT.
- `Gvt`, che contiene l'ultimo valore calcolato per il GVT.

4.2 Descrizione funzioni e implementazione dell'algoritmo

All'inizio della simulazione ogni LP, dopo aver effettuato una chiamata di inizializzazione, svolge la chiamata `Simulate()`, che rappresenta l'intero ciclo di vita del processo, dal quale uscirà solo a simulazione terminata.
All'interno della funzione `Simulate` vengono effettuate, nell'ordine, le seguenti chiamate:

- `receiveAll()`, che si occupa della ricezione di tutti i messaggi in arrivo da altri LP;
- `manageMessage()` che, per ogni messaggio ricevuto, si occupa della sua gestione;
- `manageEvent()`, che estrae l'evento con timestamp minimo dallo heap e lo processa.

L'operazione più complessa è quella di gestione dei messaggi, perché rappresenta il punto di interazione con gli altri processi. In particolare le operazioni delicate sono quelle da svolgere in caso venga riscontrata una violazione del VCL: se un LP riceve un messaggio che ha timestamp minore del tempo simulato attuale deve effettuare un'operazione di rollback, se invece riceve un anti-messaggio, quindi la violazione è stata rilevata da un altro LP, deve operare un annichilimento, che potrebbe a sua volta causare un rollback.
Queste sono, in ordine, le operazioni che un LP deve svolgere quando deve effettuare un rollback al tempo T:

- impostazione del tempo simulato attuale del processo al valore T.
- Eliminazione di tutti gli eventi processati con timestamp $T_e > T$ e inserimento di questi tra gli eventi da processare.
- Creazione e spedizione degli anti-messaggi relativi a messaggi inviati a tempi $T_m > T$.

- Eliminazione di tutti i messaggi inviati a tempi $T_m > T$.

Dopo aver ricevuto e gestito tutti i messaggi in arrivo, la funzione `Simulate` si occupa della gestione del primo evento tra quelli ancora da processare estraendo l'evento con timestamp minimo dallo heap di eventi futuri. Se il timestamp dell'evento è superiore al tempo simulato attuale, questo valore viene aggiornato al tempo dell'evento, in caso contrario l'evento va gestito al tempo corrente, per cui il tempo simulato non deve essere avanzato.
Al termine di queste operazioni, l'evento può passare alla fase di gestione vera e propria, che è però compito del modello e sarà quindi trattata nel capitolo 6.
La simulazione termina quando tutti i LP raggiungono un tempo simulato prestabilito, parametro iniziale della computazione, stampando alcune informazioni rilevanti dal punto di vista statistico.
Il calcolo del GVT è realizzato dai LP in maniera asincrona, cioè non interrompe il normale flusso d'esecuzione.
La fase di valutazione ha inizio quando una delle strutture dati associate ad un LP raggiunge una dimensione limite: il LP in questione deve semplicemente comunicare a tutti i processi che è necessario calcolare il GVT per liberare spazio in memoria, proseguendo poi nella simulazione.
Ogni LP, quando riceve il messaggio di valutazione del GVT, deve calcolare il proprio minimo locale, inserirlo in una struttura condivisa e verificare quali altri processi hanno effettuato questa operazione: nel caso in cui tutti gli altri abbiano già valutato il proprio minimo locale, il processo in questione ha il compito di valutare il minimo globale, settandolo poi in un'altra variabile condivisa.
Tutti i processi potranno quindi accedere al nuovo valore del GVT calcolato.
La valutazione del GVT è quindi realizzata in un solo *turno*, cioè è necessario effettuare un solo giro di messaggi. Per evitare che più processi calcolino il minimo globale, tale operazione viene effettuata all'interno di una sezione critica.

4.3 Funzionalità del Go utilizzate

Di seguito sono riportate le particolari funzionalità del linguaggio Go dedicate all'ambiente parallelo e multi-core, che sono state utilizzate per la parte di simulazione riguardante parallelizzazione e comunicazione:

- il supporto fornito dalle goroutines è stato utilizzato per lanciare i LP in maniera indipendente l'uno dall'altro. In questo modo i vari LP possono essere attivati parallelamente all'interno della chiamata di inizio simulazione.

- Il tipo di dato `chan` è stato sfruttato, come suggerito dagli stessi sviluppatori del linguaggio, per realizzare la comunicazione tra LP. Sono però state impiegate chiamate asincrone, per evitare che ad ogni invio o ricezione ogni processo si bloccasse indefinitamente aspettando l'esecuzione dell'operazione complementare da parte di che un altro LP. In questo modo è stata realizzata la comunicazione tra processi, ma non la sincronizzazione.

- Non è necessario effettuare alcun tipo di sincronizzazione tra i LP, ma può capitare che debbano accedere a variabili condivise ed aggiornarle, come avviene per la valutazione del GVT. In questo caso è necessario garantire l'accesso esclusivo ad alcune variabili, per il quale viene usato un semplice semaforo Mutex[2], fornito direttamente dal linguaggio.

- La variabile GOMAXPROCS è stata impiegata per modificare il numero di thread su cui distribuire le goroutines. In questo modo è stato possibile lanciare i LP in parallelismo reale, poiché, quando possibile, si è fatto in modo che ogni LP venisse lanciato su un processore diverso.

[2]Con il termine semaforo Mutex si intende un semaforo binario utilizzato per realizzare mutua esclusione: in questo modo si garantisce l'accesso esclusivo ad una sezione di codice da parte di diversi processi concorrenti.

Capitolo 5

Modello PHOLD

Un modello di benchmark adeguato è fondamentale per realizzare uno studio significativo su un algoritmo PDES. Si è scelto di utilizzare un modello formale chiamato PHOLD [13], ideato appositamente per testare le prestazioni del Timewarp.
Il PHOLD è una versione parallela del modello HOLD [14], usato per la valutazione delle prestazioni di code di eventi.

5.1 Funzionamento

Il modello PHOLD è composto da un insieme di entità simulate, suddivise equamente tra i LP: le entità sono i veri soggetti della simulazione, le strutture a cui fanno riferimento gli eventi, mentre sono i processi a doversi incaricare della gestione degli eventi e della comunicazione. Ogni LP si comporta come da specifiche per PDES con approccio ottimistico ed elabora gli eventi in ordine di timestamp.
Ogni volta che un LP elabora un evento relativo ad una delle entità che lo compongono, viene generato e spedito un nuovo evento ad una entità diversa, scelta in maniera casuale tra quelle vicine, tramite un meccanismo chiamato *movement function*.
In questo modo il numero di eventi non processati nel sistema è costante per

tutta la durata della simulazione e rappresenta uno dei parametri del modello.
La scelta della destinazione di un evento dipende fortemente dal modo in cui sono disposte le entità nel sistema: possono essere sparse, raggruppate in maniera casuale o disposte in schemi di posizionamento.
Nella creazione di un evento viene utilizzato un altro parametro importante del modello: il *timestamp increment*, che definisce la differenza temporale tra l'elaborazione e la creazione di un nuovo evento.
Infine, non meno importante degli altri, l'ultimo parametro del benchmark è il *workload*, cioè il carico di lavoro sintetico che viene svolto dal LP nel processare un evento: questo valore simula le operazioni svolte nell'elaborazione di un evento in un sistema reale e deve essere parametrizzabile per consentire lo studio del comportamento del Timewarp in diverse condizioni di carico.
Ricapitolando, il PHOLD è caratterizzato dalle operazioni svolte da ogni LP nel processare un evento:

- generazione di un nuovo evento con caratteristiche scelte in base a
  - timestamp increment, per il tempo dell'evento;
  - movement function, per l'entità di destinazione.
- Operazioni di lavoro sintetico in base al parametro di carico.

5.2 Implementazione

L'implementazione del PHOLD può essere descritta spiegando come sono state implementate le tre principali caratteristiche.
Poichè la componente casuale è decisamente importante nella generazione di un nuovo evento, per prima cosa è stato realizzato un generatore di numeri pseudocasuali [15] LCG 16807[1].

[1]Il generatore LCG 16807 fa parte della classe Linear Congruential Generator ed ha coefficiente $coeff = 16807$ e modulo $mod = 2^{31} - 1$. Ogni valore Fl generato pseudocasualmente è ottenuto a partire dal precedente ($prev$): $Fl = \frac{(coeff * prev)\%mod}{mod}$

Nella generazione di un evento, il destinatario è scelto in maniera uniforme tra tutte le entità del sistema.
Il timestamp increment è calcolato sommando un valore pseudocasuale con distribuzione esponenziale al tempo dell'evento processato, in modo che il timestamp del nuovo evento sia maggiore o uguale al tempo attuale e rispettando così il principio di causalità.
Il carico computazionale è stato realizzato sotto forma di operazioni floating point: in questo modo il processo spende effettivamente tempo in operazioni di calcolo e la quantità di lavoro è facilmente controllabile modificando il numero di operazioni floating point (nFPops) da effettuare nel processare gli eventi.
Le operazioni effettuate sono iterazioni del metodo di Newton per calcolare $\sqrt{2}$, che possono quindi essere eseguite indefinitamente, senza il rischio di incorrere in problemi numerici.
Variando il parametro nFPops si modifica in maniera molto diretta il comportamento del PHOLD: più è alto tale valore più la simulazione sarà di tipo computational-bound.

La versione del PHOLD implementata è parametrizzabile al momento della chiamata, in modo che l'utente possa scegliere diversi casi da simulare.
Il numero di LP e quello di entità sono parametri passati alla simulazione da linea di comando, mentre il workload, la densità degli eventi e il tempo di fine simulazione vengono assegnati tramite un file di configurazione.

Capitolo 6

Fase di Testing

L'intera fase di test è stata eseguita sulla macchina del cluster di simulazione precedentemente citata: chernobog.
Dopo aver accertato che il linguaggio di programmazione è adeguato a supportare computazioni parallele, sono state effettuate alcune prove, in diverse situazioni di carico del modello PHOLD, per studiare il comportamento dell'algoritmo Timewarp: in particolare si è cercato di verificare se la parallelizzazione consente un incremento delle prestazioni.
In tutti i casi analizzati sono state svolte diverse prove, dalle quali sono state poi ricavate informazioni statisticamente valide.
I due valori osservati in questa fase di test riguardano il Wall Clock Time (WCT), ovvero il tempo reale impiegato dalla simulazione, che sarà espresso in millisecondi, e lo speedup, che è già stato descritto precedentemente.

6.1 Numero di processori

Il primo test è stato svolto con lo scopo di verificare il comportamento dell'algoritmo al variare del numero di processori utilizzati nella simulazione: si è scelto un caso particolare per i parametri del PHOLD e si è analizzato il comportamento dell'algoritmo al variare di GOMAXPROCS, quindi del

numero di cpu utilizzate.
I parametri scelti per il PHOLD rappresentano un caso di carico intermedio:

- 0.5 per la densità degli eventi;
- tempo di fine simulazione pari a 1000;
- 10000 operazioni floating point per l'elaborazione di un evento;
- 1500 entità simulate, da suddividere nei diversi LP.

Si è quindi studiato il comportamente del Timewarp nell'esecuzione con 1,2,3 e 4 Logical Process.

N^o processori	1	2	3	4
1 LP	2964	2966	2965	2965
2 LP	374208	1915	1915	1905
3 LP	689377	282825	1627	1628
4 LP	1214844	713825	206127	1513

Tabella 6.1: WCT Timewarp al variare del numero di cpu (ms)

N^o processori	1	2	3	4
1 LP	1.000	1.000	1.000	1.000
2 LP	0.008	1.549	1.548	1.556
3 LP	0.004	0.010	1.822	1.821
4 LP	0.002	0.004	0.014	1.960

Tabella 6.2: Speedup Timewarp al variare del numero di cpu

Come si può osservare dalle tabelle, c'è una separazione netta nello schema di prestazioni del Timewarp: quando il numero di processori impiegati per la simulazione è minore del numero di LP, i tempi di calcolo aumentano in

maniera enorme.

N^o processori	1	2	3	4
1 LP	0	0	0	0
2 LP	47201	22	25	21
3 LP	116713	51921	122	125
4 LP	187266	104681	62965	257

Tabella 6.3: Numero di rollback al variare del numero di cpu

Per capire le ragioni di questo comportamento basta guardare il numero di rollback complessivi effettuati dai LP nel sistema, al variare del numero di cpu impiegate. Come per il WCT, anche per il numero di rollback c'è una netta distinzione: se il Timewarp viene lanciato su un numero di processori non inferiore al numero di LP, la quantità di rollback effettuati è relativamente bassa, mentre negli altri casi diventa altissima, causando un pesante degrado delle prestazioni.
Questo comportamento è comprensibile se si pensa a quello che succede se il numero di LP p è superiore al numero di cpu c: in ogni istante della simulazione ci possono essere al massimo c processi in esecuzione, che procedono nella simulazione, mentre gli altri $p - c$ sono fermi.
I processi in esecuzione, come da specifiche per il Timewarp, si comporteranno in maniera ottimistica, processando gli eventi che li riguardano senza preoccuparsi se questi sono safe o meno, avanzando nella simulazione. Ad un certo punto della computazione si avrà uno switch e uno dei LP attivi verrà sostituito da uno inattivo, che si inserirà nella simulazione.
Il nuovo processo si troverà ad un tempo simulato inferiore agli altri e non appena invierà un messaggio ad uno di essi, il ricevente sarà costretto ad effettuare un rollback, che probabilmente coinvolgerà tutti i processi attivi.
Questa situazione, nel caso in cui $p > c$, si verificherà ogni volta che un LP verrà attivato e impedirà alla simulazione di procedere in maniera lineare.

I dati ricavati mostrano come una simulazione effettuata su un numero di processori inferiore al numero di LP sia altamente inefficiente a causa di caratteristiche intrinseche del Timewarp. Inoltre si può osservare come aumentando il numero di cpu disponibili, ma non il numero di LP, le prestazioni rimangano sostanzialmente invariate.
Per questo motivo, i test successivi verranno effettuati su un numero di processori pari al numero di LP impiegati nella simulazione.

6.2 Tempo di fine simulazione

Uno dei parametri del modello PHOLD è rappresentato da `EndTime`, che indica a quale tempo simulato la simulazione deve arrestarsi: è quindi un valore che influisce direttamente sulla durata dell'elaborazione.
Lo scopo di questa sezione è mostrare come il valore di `EndTime` non influisca sulla velocità di esecuzione dell'algoritmo[1], a patto che:

- sia sufficientemente grande da consentire al sistema di entrare in uno stato stabile;
- non sia troppo alto, in modo da garantire che la simulazione termini senza che insorgano problemi non direttamente collegati al Timewarp, come la fine dello spazio di memoria o l'entrata in funzione del meccanismo di garbage collection, che potrebbero influire pesantemente sulle prestazioni.

Si è scelto quindi un caso particolare del PHOLD, fissando tutti i parametri del modello, tranne quello riguardante il tempo di fine simulazione, ad un valore intermedio:

[1] É chiaro come il valore di fine simulazione ne modifichi direttamente la durata. Ciò che si vuole mostrare è che questo parametro del modello non ha alcun effetto sulle prestazioni sulla simulazione, intese come velocità di esecuzione.
Nello specifico, ci si aspetta che una simulazione con `EndTime` $= T$ duri la metà di una con `EndTime` $= 2T$.

- densità degli eventi a 0.5;
- 2400 entità simulate, da suddividere nei diversi LP;
- 10000 operazioni floating point come carico computazionale.

Si è valutato il comportamento del Timewarp con `EndTime` pari a 500, 1000 e 2000.

N^o LP	Endtime = 500	Endtime = 1000	Endtime = 2000
1 LP	2378	4757	9496
2 LP	1480	2975	5937
3 LP	1215	2416	4758
4 LP	1114	2156	4220

Tabella 6.4: WCT Timewarp: prestazioni al variare di EndTime (ms)

Nel caso analizzato si può osservare come il tempo impiegato per completare la simulazione cresca linearmente con il valore `EndTime`. Questo indica che l'aumento del parametro `EndTime` non introduce alcun fattore in grado di modificare le prestazioni del Timewarp.
Avendo scelto in maniera indeterminata l'esempio su cui effettuare la valutazione, ci si aspetta che i risultati osservati siano validi anche negli altri casi e che il valore assegnato al parametro `EndTime` non abbia alcuna influenza sulla capacità di scalare dell'algoritmo. Per questo motivo, d'ora in poi, non verrà più segnalato al lettore il valore usato come tempo di fine simulazione, dato che questo non ha alcuna influenza sulla capacità di scalare dell'algoritmo Timewarp.

6.3 Scalabilità in base al Workload

Il primo test sulla scalabilità è stato realizzando modificando adeguatamente il paramentro workload del modello PHOLD, che gestisce la quantità di operazioni floating point effettuate dai LP nel processare ogni evento.
Si è scelto quindi un caso particolare del PHOLD, con densità degli eventi pari a 0.5 e 2400 entità simulate, e lo si è analizzato al variare del carico computazionale.
Per stabilire quale range di valori utilizzare si è preso spunto da alcuni articoli sul Timewarp [16, 17] e si è scelto di valutare tre casi di carico: 1000, 10000 e 100000 operazioni floating point.
Negli articoli considerati il carico di lavoro era dell'ordine delle centinaia di migliaia di operazioni per evento, ma si è scelto di utilizzare valori più piccoli e meno omogenei per poter realizzare un'analisi più dettagliata e significativa delle prestazioni del Timewarp.

N^o LP	1000 FPops	10000 FPops	100000 FPops
1 LP	1319	4757	40664
2 LP	1151	2975	21009
3 LP	1947	2416	14528
4 LP	1957	2156	11215

Tabella 6.5: WCT Timewarp: scalabilità in base al workload (ms)

N^o LP	1000 FPops	10000 FPops	100000 FPops
1 LP	1.00	1.00	1.00
2 LP	1.15	1.60	1.94
3 LP	0.68	1.97	2.80
4 LP	0.67	2.21	3.63

Tabella 6.6: Speedup Timewarp: scalabilità in base al workload

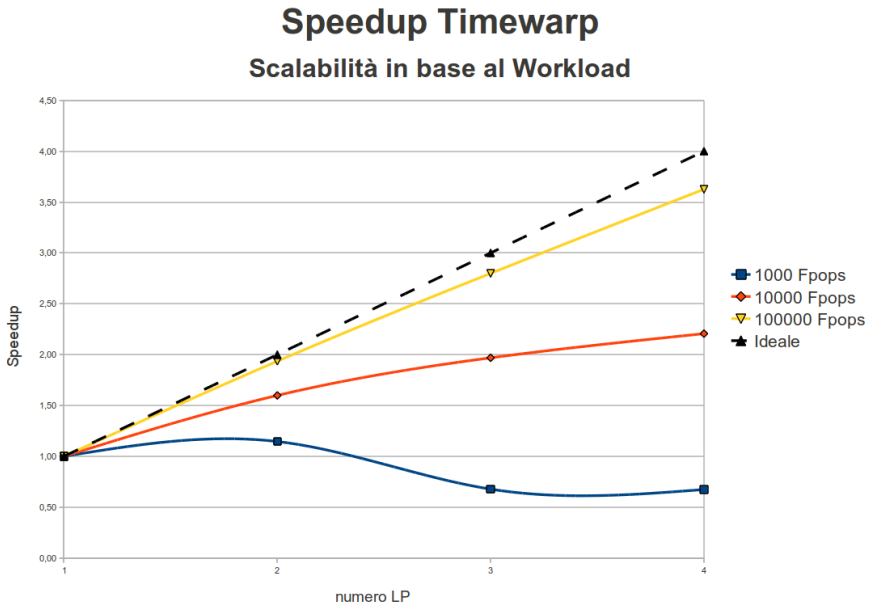

Figura 6.1: Speedup Timewarp: scalabilità in base al workload

Dall'analisi di grafico e tabelle si può osservare come lo speedup sia nettamente migliore, rispetto agli altri casi, nell'esecuzione con 100000 operazioni floating point, cioè quando il carico computazionale è massimo. In questo caso infatti le prestazioni dell'algoritmo scalano in maniera quasi ottimale.
Nel caso intermedio, con 10000 operazioni per evento, le prestazioni migliorano al crescere delle cpu utilizzate e quindi anche lo speedup tende a crescere, ma in maniera meno significativa rispetto al caso precedente. Il guadagno temporale ottenuto con la parallelizzazione è comunque significativo e per questo, anche in queste condizioni, l'algoritmo può essere considerato efficiente.
Nel terzo caso invece il WCT diminuisce, rispetto all'esecuzione sequenziale, solo nell'esecuzione con due LP, mentre in quella con tre e quattro processi il tempo impiegato dalla simulazione è più alto.

In conclusione si osserva che il comportamento dell'algoritmo è abbastan-

za diverso nei tre casi esaminati. Inoltre si è mostrato come il Timewarp scali in maniera migliore quando il carico computazionale è più alto. Questo dipende dal fatto che le operazioni di computazione fanno parte della sezione parallelizzabile del programma, il cui tempo di calcolo, in un'esecuzione parallela, viene suddiviso tra i diversi processori.
É chiaro quindi che più alto è il tempo dedicato alla computazione, maggiore sarà il miglioramente ottenuto con una parallelizzazione.
Nella parte restante della fase di valutazione verranno omesse le informazioni riguardanti il carico computazionale e il modello verrà utilizzato con `nFPops` pari a 10000. In questo modo è possibile analizzare il comportamento dell'algoritmo in maniera indipendente dal workload.

6.4 Scalabilità in base alla Densità

Come secondo test sulla scalabilità del Timewarp, si è osservato il comportamento dell'algoritmo al variare del paramentro densità d del modello PHOLD, che definisce il numero di eventi nel sistema (n_{ev}) in base al numero di entità simulate (n_{ent}): $n_{ev} = n_{ent}d$.
Come nel test precedente, si è scelto un caso particolare del PHOLD, fissando il numero di entità simulate a 2400 e tutti gli altri parametri del modello ai valori standard definiti precedentemente. Si è scelto quindi di studiare il comportamento del Timewarp in quattro casi di densità: 0.1, 0.5, 0.9, 1.5.

N^o LP	d = 0.1	d = 0.5	d = 0.9	d = 1.5
1 LP	901	4757	8653	14778
2 LP	703	2975	5263	8783
3 LP	709	2416	4107	6687
4 LP	755	2156	3560	5680

Tabella 6.7: WCT Timewarp: scalabilità in base alla densità (ms)

N^o LP	d = 0.1	d = 0.5	d = 0.9	d = 1.5
1 LP	1.00	1.00	1.00	1.00
2 LP	1.41	1.60	1.64	1.68
3 LP	0.40	1.97	2.11	2.21
4 LP	0.31	2.21	2.43	2.60

Tabella 6.8: Speedup Timewarp: scalabilità in base alla densità

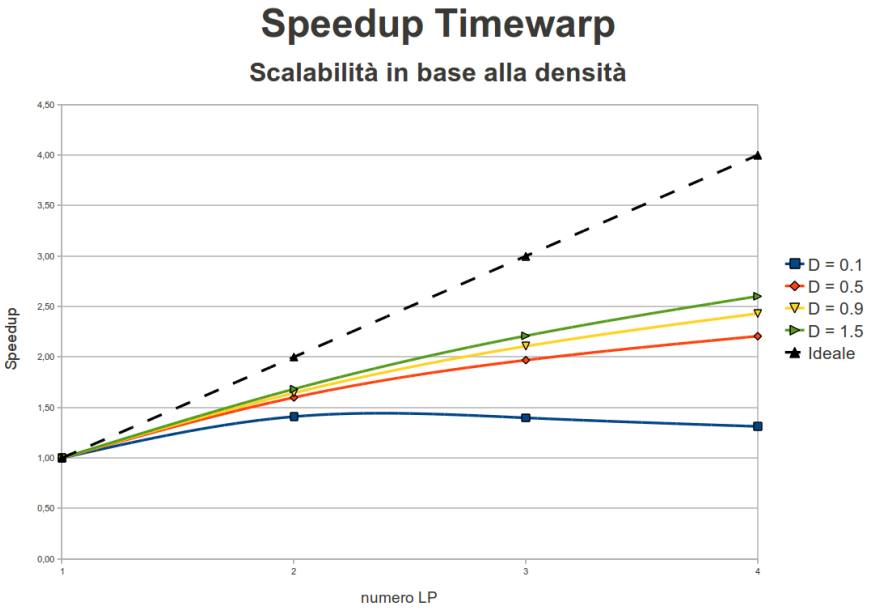

Figura 6.2: Speedup Timewarp: scalabilità in base alla densità

Il grafico mostra come i valori dello speedup ottenuti tendano generalmente a migliorare con il crescere della densità.
Si osserva che il WCT, nel caso con densità 0.1, diminuisce nel passare da 1 a 2 LP, mentre aumenta nelle esecuzioni con 3 e 4 processi, pur rimanendo inferiore a quello dell'esecuzione sequenziale.

In questo caso, con densità pari a 0.1, le prestazioni ottenute dal Timewarp non sono sufficientemente buone per giustificare l'uso del parallelismo, ma in tutti gli altri casi la scalabilità è buona.

6.5 Scalabilità in base al numero di entità

Nel terzo test sono state valutate le prestazioni del Timewarp osservandone il comportamento al variare del numero di entità coinvolte nella simulazione.
Il modello è stato parametrizzato in un caso intermedio, fissando la densità degli eventi a 0.5 e osservando i casi con 900, 1800, 3600 e 7200 entità.

N^o LP	900 entità	1800 entità	3600 entità	7200 entità
1 LP	1796	3548	7145	14765
2 LP	1191	2265	4415	8775
3 LP	1073	1860	3489	6678
4 LP	1039	1717	3028	5740

Tabella 6.9: WCT Timewarp: scalabilità in base al numero di entità (ms)

N^o LP	d = 0.1	d = 0.5	d = 0.9	d = 1.5
1 LP	1.00	1.00	1.00	1.00
2 LP	1.51	1.57	1.62	1.68
3 LP	1.67	1.91	2.05	2.21
4 LP	1.73	2.07	2.36	2.57

Tabella 6.10: Speedup Timewarp: scalabilità in base al numero di entità

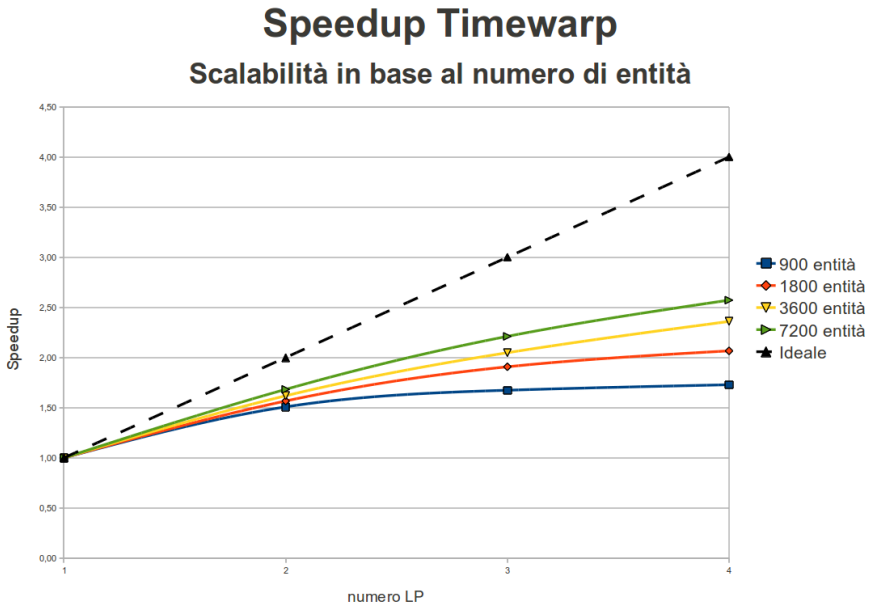

Figura 6.3: Speedup Timewarp: scalabilità in base al numero di entità

I risultati ottenuti mostrano come i valori dello speedup ottenuti tendano generalmente a migliorare con il crescere della densità.
Si osserva che il WCT, nel caso con densità 0.1, diminuisce nel passare da 1 a 2 LP, mentre aumenta nelle esecuzioni con 3 e 4 processi, pur rimanendo inferiore a quello dell'esecuzione sequenziale.
In questo caso, con densità pari a 0.1, le prestazioni ottenute dal Timewarp non sono sufficientemente buone per giustificare l'uso del parallelismo, ma in tutti gli altri casi la scalabilità è buona.

Capitolo 7

Profiling dell'algoritmo

7.1 Introduzione

È stata effettuata una fase di profiling del Timewarp, nella quale si è studiato come viene impiegato il tempo nella simulazione.
La versione dell'algoritmo che è stata realizzata è in grado di mantenere informazioni riguardanti le operazioni svolte durante la simulazione ed il tempo speso nell'esecuzione delle diverse parti. In questo modo è stato possibile realizzare una traccia dell'esecuzione e stabilire in che modo i processori impiegano il tempo della simulazione.
Ovviamente l'operazione di profiling introduce un overhead, che in alcuni casi risulta abbastanza consistente, dovuto alle operazioni che è necessario svolgere per stabilire come viene impegnato il processore e per mantenere informazioni. Per questo motivo l'analisi effettuata non è completamente attendibile e la precisione delle stime non può essere ottimale.
Partendo da questo presupposto, è comunque possibile svolgere un'analisi approssimata sul funzionamento dell'algoritmo.
Sono state effettuate 3 prove, in differenti condizioni di carico (1000, 10000 e 100000 operazioni floating point), per quantificare l'overhead introdotto dall'operazione di profiling.

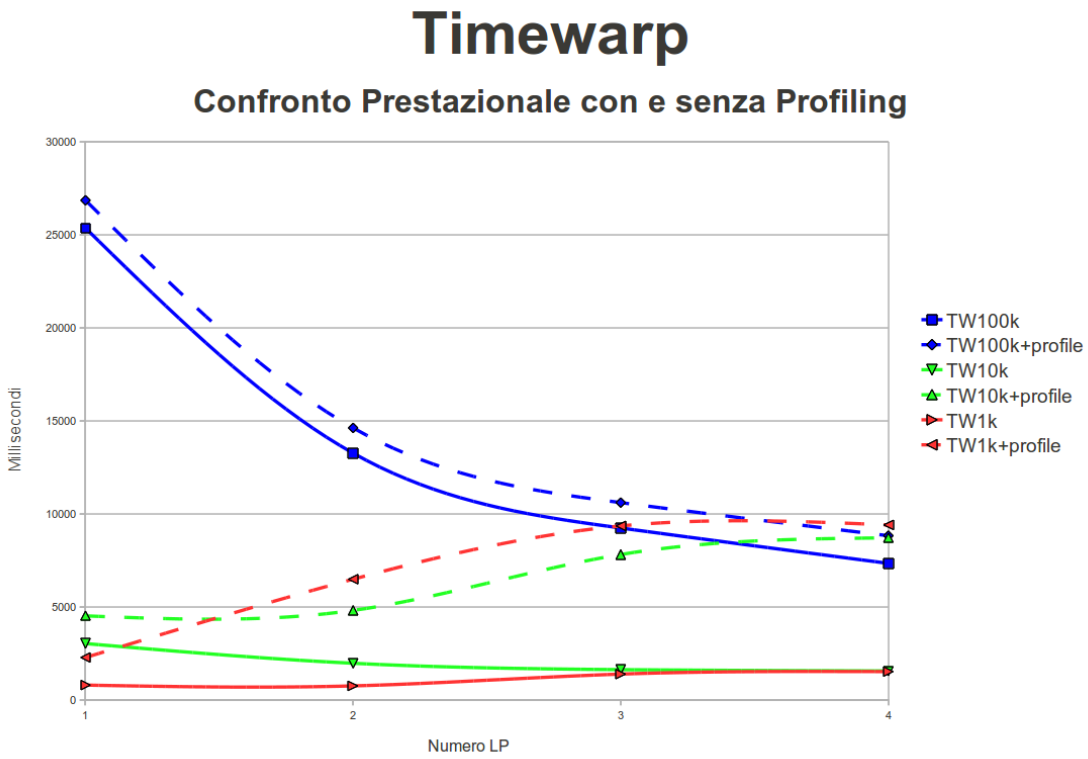

Figura 7.1: Confronto Prestazioni con e senza Profiling

Il grafico 7.1 mostra un confronto tra i tempi di esecuzione del Timewarp con e senza l'operazione di profiling: risulta evidente come nei casi con 1000 e 10000 operazioni l'inserimento del profiler modifichi pesantemente le prestazioni, mentre nel caso di `nFPops` = 100000 l'overhead introdotto appesantisce lievemente la computazione, ma non la modifica in maniera sostanziale.
In conclusione l'operazione di profiling influisce direttamente sulle prestazioni del Timewarp, ma può comunque dare un'idea di come viene impiegato il tempo durante la computazione.

7.2 Profiling, caso nFPops = 1000

Il primo caso analizzato è stato quello con carico computazionale minore: 1000 operazioni floating point.
Il tempo d'esecuzione dell'algoritmo è stato suddiviso in sette categorie.

- *Comunicazione* rappresenta la durata delle operazioni di invio e ricezione di messaggi sui canali.
- *Valutazione GVT* indica il tempo impiegato nelle operazioni per il calcolo del Global Virtual Time.
- *Computazione* rappresenta l'elaborazione di un evento ed è costituita dal tempo impiegato nella computazione sintetica.
- *Heap* rappresenta il tempo speso nelle operazioni di inserimento, ricerca ed estrazione effettuato sullo heap.
- *List* indica il tempo impiegato nella gestione delle diverse liste.
- *Idle Time* è il tempo in cui i LP sono rimasti inattivi.
- *Rollback* rappresenta la durata delle operazioni di rollback.

Funzione	1 LP	2 LP	3 LP	4 LP
Comunicazione	3%	23%	27%	27%
Valutazione GVT	3%	7%	9%	10%
Computazione	37%	18%	16%	14%
Heap	45%	30%	24%	25%
List	9%	8%	8%	7%
Idle Time	4%	5%	5%	5%
Rollback	0%	9%	12%	13%

Tabella 7.1: Profiling Timewarp, caso nFPops = 1000

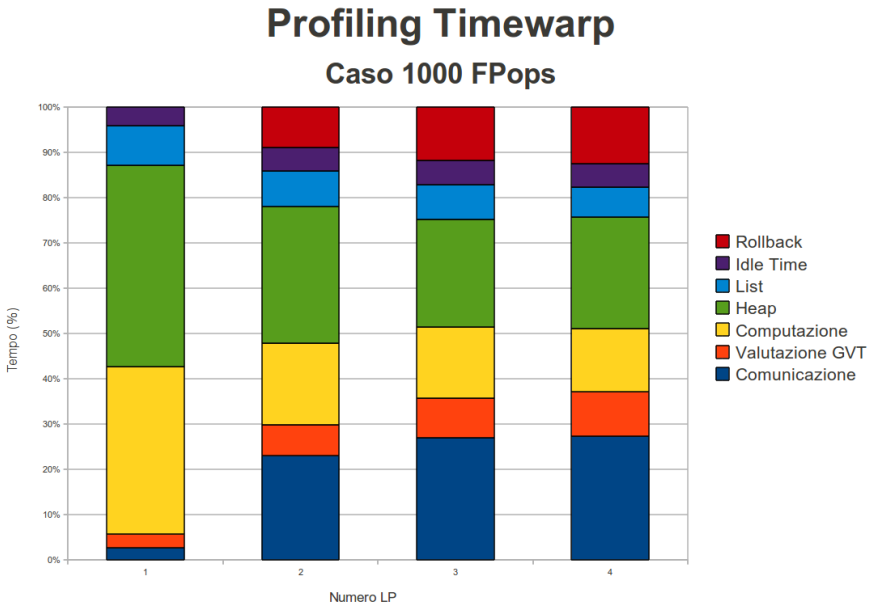

Figura 7.2: Profiling Timewarp, caso nFPops = 1000

In questo caso la percentuale di tempo utilizzato dalle diverse componenti della simulazione varia in maniera piuttosto evidente con l'aumento del numero di LP.
In particolare è interessante osservare come il tempo dedicato a comunicazione, valutazione del GVT e operazioni di rollback aumenti con il crescere del parallelismo, mentre la durata delle operazioni di computazione e gestione delle strutture dati diminuisca.
Questo significa che la parallelizzazione ha portato dei benefici evidenti, riducendo i tempi di calcolo e di gestione, ma ha introdotto alcune operazioni di supporto, il cui costo cresce con l'aumento del livello di parallelismo.

7.3 Profiling, caso nFPops = 10000

Il secondo caso analizzato è stato quello con carico computazionale pari a 10000 operazioni floating point.

Funzione	1 LP	2 LP	3 LP	4 LP
Comunicazione	2%	20%	29%	31%
Valutazione GVT	2%	3%	4%	5%
Computazione	73%	53%	37%	28%
Heap	12%	12%	13%	15%
List	9%	5%	5%	6%
Idle Time	2%	3%	3%	4%
Rollback	0%	4%	8%	12%

Tabella 7.2: Profiling Timewarp, caso nFPops = 10000

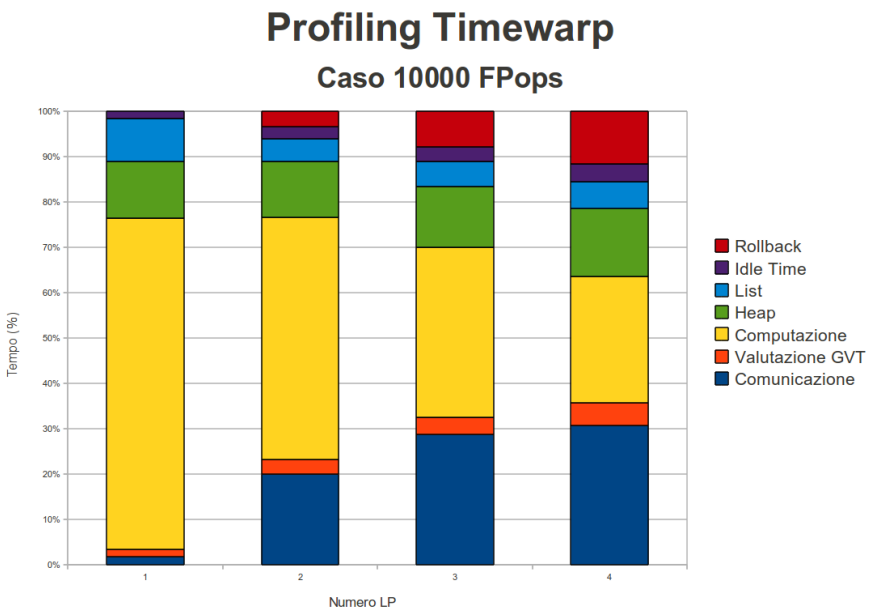

Figura 7.3: Profiling Timewarp, caso nFPops = 10000

Nel secondo caso preso in esame il profiling fornisce informazioni molto simili al caso precedente: con la parallelizzazione diminuisce il tempo speso in operazioni di computazione e gestione delle strutture dati, mentre aumenta quello impiegato per la computazione e lo scambio di messaggi.

La differenza più evidente rispetto al caso precedente riguarda il tempo impiegato nella computazione che, nell'esecuzione con 10000 operazioni floating point, occupa la maggior parte del tempo, arrivando ad occuparne il 73% nell'esecuzione su un solo LP.
Appare evidente come, aumentando il carico computazionale, cresca anche il tempo dell'operazione di elaborazione degli eventi. Questo fatto mostra anche come il guadagno ottenuto con la parallelizzazione sia maggiore se il workload è più alto, poichè l'operazione di computazione può essere eseguita in maniera parallelela.

7.4 Profiling, caso nFPops = 100000

Infine è state effettuta l'operazione di profiling nel caso con `nFPops` pari a 100000.

Funzione	1 LP	2 LP	3 LP	4 LP
Comunicazione	0%	3%	5%	8%
Valutazione GVT	0%	1%	1%	1%
Computazione	97%	92%	89%	86%
Heap	1%	2%	2%	2%
List	1%	1%	1%	1%
Idle Time	0%	0%	0%	0%
Rollback	0%	1%	2%	2%

Tabella 7.3: Profiling Timewarp, caso nFPops = 100000

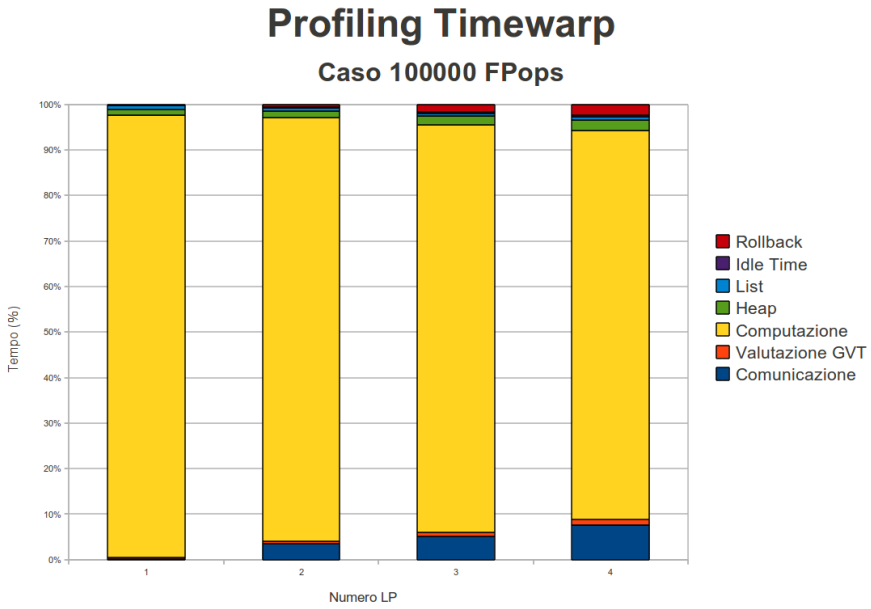

Figura 7.4: Profiling Timewarp, caso nFPops = 100000

In questo caso risulta evidente come le operazioni di computazione utilizzino i processori per quasi tutto il tempo d'esecuzione.
Si può comunque osservare come il tempo speso per la comunicazione cresca con l'aumento del nuemro di LP utilizzati, anche se costituisce solo una minima parte del tempo totale.

Capitolo 8

Analisi dei risultati

Nella fase di valutazione si è osservato come la scalabilità del Timewarp tenda a migliorare in condizioni di carico intenso, cioè nei casi in cui il modello ha una quantità maggiore di lavoro da svolgere[1].
Non è stato possibile stabilire quale sia il lavoro utile per migliorare lo speedup, perchè il guadagno ottenuto con la parallelizzazione cresce, in tutti i casi studiati, cioè all'aumentare del numero di entità, della densità e del carico computazionale.
Il parametro di workload del PHOLD sembra essere separato logicamente dagli altri, perchè rappresenta la quantità di lavoro da svolgere nell' operazione di elaborazione degli eventi, che ha poco in comune con il numero di eventi o entità.
La densità degli eventi d e il numero di entità N_{ent} invece, sono strettamente collegati, poichè cooperano nello stabilire il numero di eventi N_{ev}.
Probabilmente è quindi possibile integrare questi due parametri in uno solo: il numero di eventi. Per fare questo è però necessario verificare se il Timewarp si comporta nello stesso modo quando il numero di eventi è lo stesso, ma cambiano i due parametri d e N_{ent}.

[1] In questo caso con i termini lavoro e carico non si intende il carico computazionale (workload), ma la quantità di operazioni complessive che il modello deve svolgere.

8.1 Verifica: numero di eventi

É stata realizzata quindi una ulteriore fase di testing, con lo scopo di verificare se è possibile unificare la densità e il numero di eventi in un unico parametro: il numero di eventi.
A tale scopo si è scelto un caso intermedio (`EndTime` = 1000, `nFPops` = 10000) e sono state quindi effettuate due prove:

- per quanto riguarda la prima si è scelto il caso $N_{ent} = 3000$, $d = 0.5$;
- per la seconda si è fissato $N_{ent} = 1500$ e $d = 1.0$.

In questo modo il numero di eventi, calcolato come $N_{ev} = N_{ent}d$, è uguale in entrambi i casi.

N^o LP	$d = 1.0, N_{ent} = 1500$	$d = 0.5, N_{ent} = 3000$
1 LP	5932	5977
2 LP	3692	3699
3 LP	2939	2925
4 LP	2608	2611

Tabella 8.1: Confronto: numero di eventi

La tabella 8.1 mostra come le prestazioni, nei due casi analizzati, siano quasi identiche. Questo significa che, dal punto di vista della simulazione, le prestazioni di un'esecuzione con densità d e numero di entità e sono uguali a quelle ottenute, per esempio, con una densità $2d$ e numero di entità $\frac{e}{2}$.
Tuttavia, dal punto di vista della modellazione, non è possibile unificare i due casi dell'esempio appena mostrato, poichè rappresentano due situazioni completamente diverse, che devono essere mantenute distinte.
Le due prove rappresentano quindi casi logicamente differenti e che sono modellati in maniera diversa, ma mostrano un comportamento quasi identico perchè i parametri del PHOLD risultano uguali, in quanto il valore realmente

significativo è il numero di eventi.
In conclusione, potrebbe risultare interessante eliminare il parametro densità e sostituirlo con qualcosa che rappresenti il numero di eventi in maniera indipendente dal numero di entità, ma questa operazione sarebbe sbagliata dal punto di vista logico, poichè è evidente come il numero di eventi nel sistema debba essere direttamente dipendente dal numero di entità.

8.2 Carico di lavoro

Riassumendo quindi, i test effettuati parametrizzando adeguatamente il PHOLD mostrano come l'algoritmo scali in maniera ottimale solo quando il carico di lavoro è rilevante.
Il concetto di carico di lavoro, a questo punto, ha assunto un duplice significato. In particolare indica:

- il carico computazionale (workload), inteso come il carico di lavoro da svolgere nel processare un evento;
- la dimensione del problema modellato, rappresentata dal numero di eventi nel sistema.

In conclusione si può affermare che, se il sistema è sufficientemente carico, è possibile ottenere un miglioramento prestazionale tramite la parallelizzazione, che sarà tanto maggiore quanto più è alto il livello di parallelismo.
Si è mostrato quindi, come l'esecuzione più efficiente sia quella nella quale il numero di Logical Process è massimo, ed ognuno di essi può essere eseguito indipendentemente su un processore.

Conclusioni

La fase di valutazione ha mostrato chiaramente come questa implementazione del Timewarp consenta di ottenere un miglioramento prestazionale tramite un'operazione di parallelizzazione, a condizione che i parametri di carico del modello siano sufficientemente alti.
É stato messo in evidenza come il parallelismo sia realizzato esclusivamente a livello dei LP e garantisca prestazioni ottimali solo nel caso in cui il numero di cpu utilizate è almeno pari al numero di processi.
In base ai test effettuati in diverse condizioni si è ricavato, inoltre, che l'esecuzione più efficiente dal punto di vista prestazionale è quella effettuata utilizzando il numero massimo di LP.
Si è quindi implementata la possibilità di lanciare la simulazione in maniera ottimale, automatizzando la scelta del numero di LP in base alle caratteristiche dell'architettura a disposizione. In questo modo la simulazione può adattarsi al sistema utilizzato, sfruttandone tutte le potenzialità e garantendo che il tempo di esecuzione sia, in queste condizioni, il più piccolo possibile. Questo significa che il numero di processi da utilizzare nella simulazione non è più un problema dell'utente, poichè la simulazione può essere eseguita automaticamente in maniera ottimale.
É stato possibile realizzare questo lavoro di ottimizzazione solo grazie alle particolari attenzioni che il linguaggio Go dedica all'ambiente multicore e al parallelismo.
In conclusione, il Go ha dimostrato di essere una valida opzione da utilizzare come linguaggio in una simulazione parallela a memoria condivisa: a

differenza della maggior parte degli altri linguaggi di programazione, supporta in maniera diretta il parallelismo, garantendo una buona scalabilità e prestazioni ai livelli di altri linguaggi di programmazione già affermati e largamente utilizzati in questo ambito.
Per quanto riguarda l'ambiente distribuito, al momento il Go non offre nulla in più degli altri linguaggi e per questo non è stato effettuato nessun tipo di test.
In ogni caso, essendo un linguaggio giovane e ancora in via di sviluppo, non è detto che funzionalità ulteriori non possano essere aggiunte in seguito.

Bibliografia

[1] Goth, G., Entering a Parallel Universe, Communications ACM (September 2009) 52 (9), 15-17

[2] Moore, Gordon E., Cramming more components onto integrated circuits, Electronics Magazine. (1965) pp. 4

[3] International Technology Roadmap for Semiconductors, Executive Summary, 2005 and 2007; http://public.itrs.net/

[4] Amdahl, G., Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities, (1967) AFIPS Conference Proceedings (30): 483-485

[5] Lamport, L. Time, Clocks and the Ordering of Events in a Distributed System, Communications of the ACM 21, 7 (July 1978), 558-565

[6] Fujimoto, R.M. Parallel and Distributed Simulation Systems, Wiley-Interscience

[7] Chandy, K.M., Misra, J. Deadlock absence proofs for networks of communicating processes Inf. Process. Lett. 9, 4 (Nov. 1979), 185-189

[8] Jefferson, D.R. Virtual time, ACM Transactions on Programming Languages and Systems, 7(3):404-425, July 1985

[9] Samadi, B., Muntz, R.R., and Parker, D.S. A Distributed Algorithm to Detect a Global State of a Distributed Simulation System In Proc. IFIP Conference on Distributed Processing, Amsterdam, North-Holland, 1987

[10] The Go Programming Language, http://golang.org/

[11] Hoare, C. A. R. Communicating Sequential Processes, Prentice Hall International Series in Computer Science, 1985

[12] Introduction to Parallel Computing, https://computing.llnl.gov/tutorials/parallel_comp/#ExamplesPI

[13] Fujimoto, R.M. Performance of Time Warp under synthetic workloads, Proceedings of the SCS Multiconference on Distributed Simulation (1990), 23-28

[14] Jones, D. W. , An empirical comparison of priority-queue and event-set implementations. Communications ACM (April 1986) 29 (4), 300-311

[15] Zeeb C. N., Burns P. J. Random Number Generator Recommendation

[16] Bing Wang, Yiping Yao, Himmelspach, J., Ewald, R., Uhrmacher, A.M. Experimental Analysis of Logical Process Simulation Algorithms in James II, Winter Simulation Conference (WSC), Proceedings of the 2009, 1167-1179

[17] Gupta A., Akyildiz, I., Fujimoto, R.M. Performance analysis of Time Warp with homogeneous processors and exponential task times. SIGMETRICS Perform. Eval. Rev. 19, 1 (April 1991), 101-110