Relazione sull'implementazione dell'algoritmo Work

Alberto Franco, Mirco Polato, Lorenzo Tessari

14 novembre 2011

1 Introduzione

L'algoritmo Work rappresenta un notevole risultato nella ricerca legata al problema dei k-server. Questo algoritmo infatti permentte di rispondere alle richieste del problema in modo $(2k-1)$-competitivo che non è il limite inferiore ma ci si avvicina molto.

Ci è stato chiesto di implementare questo algoritmo in modo efficiente, in questo documento presentiamo l'implementazione data. Il pezzo di software che è stato prodotto è stato scritto in C++ per la grande effificienza del linguaggio e la libertà che questo offre in termini di gestione delle risorse, come la memoria, ad esempio.

1.1 Struttura del progetto

Il progetto è strutturato in questo modo: vi sono due cartelle all'interno della principale `src` e `docs`. Questo documento è stato generato dal file LaTeXcontenuto nella cartella `docs`. L'altra cartella contiene tutti i sorgenti (`*.h, *.cpp`) dell'applicazione. Durante lo sviluppo è stato utilizzato il sistema di gestione del progetto *CMake* il cui file di configurazione è all'interno della directory radice, nella stessa cartella è presente un makefile per compilare l'applicazione.

Il software è stato testato su Linux a 64 bit con il compilatore GCC e in ambiente Windows sempre a 64 bit su compilatore Microsoft Visual C++.

2 Sviluppo della applicazione

In questa sezione descriveremo brevemente quali sono state le scelte di design operate nell'implementare l'algoritmo. Rispetto ad una implementazione volta solo ad ottenere efficienza si è preferito usare disegno (e conseguentemente una programmazione) ad oggetti il più corretto possibile in modo da dare a chi usa la componente software una maggiore comprensione del modo in cui questa funziona.

In primo luogo la classe che gestice i punti in cui arrivano le richieste è una classe che ha metodi tutti virtuali puri (una interfaccia in altri linguaggi di programmazione, quali Java). In questo modo è possibile scrivere altre implementazioni della classe `Point` su altri spazi metrici e il tutto resta coerente e corretto. A questo scopo è stato anche parametrizzato il tipo "distanza" in

modo che si possa sostituire con un altra e la componente software continua a funzionare[1].

La seconda scelta di design operata è stato di permettere agli utenti della classe `Configuration` di scorrere gli elementi della configurazione tramite un iteratore invece che dare accesso diretto alla memoria sottostante. Questo permette una gestione più elegante dell'iterazione di una configurazione e successive modifiche nella gestione interna non modificherebbero il codice utente già scritto.

Nelle sotto sezioni successive verranno presentate le ottimizzazioni operate per rendere l'esecuzione più veloce.

2.1 Paginazione della memoria

La prima ottimizzazione che a nostro avviso era necessario operare risiede nell'ottimizzare l'uso della memoria al fine di evitare continue allocazioni e deallocazioni. Notoriamente l'uso efficiente della memoria è fonte di grande ottimizzazione all'interno delle applicazioni. Il metodo più semplice che ci è sembrato opportuno usare per sviluppare l'algoritmo è il *caching*. Al posto di allocare e deallocare gli oggetti che usiamo spesso (configurazioni nel nostro caso) abbiamo deciso di usare e riciclare gli oggetti che già abbiamo. A questo proposito la classe `ConfigurationFactory` si occupa di gestire le copie di oggetti di tipo `Configuration` che abbiamo creato. Inizialmente configuration factory inizializza una pagina di memoria con un numero di oggetti da noi deciso e, ogni qualvolta gli venga richiesto un oggetto nuovo questa effettua la seguente operazione:

- Se vi sono ancora oggetti disponibili sulla pagina di memoria correntemente allocata allora ritorna uno di quelli oggetti.
- Se gli oggetti sono tutti in uso allora alloca una nuova pagina di memoria e ritorna un oggetto.

Questo avviene quando il metodo `create()` viene invocato. Una volta che l'oggetto è stato utilizzato esso deve venire riciclato tramite una chiamata a `recycle(Configuration*)` che reinserisce l'oggetto nello stack degli oggetti disponibili. É stato usato uno stack per favorire la località dei riferimenti, se un oggetto viene deallocato e poi un altro subito richiesto è probabile che questo resti in cache (o quanto meno il puntatore all'oggetto).

Una nota da fare è che la classe `ConfigurationFactory` è stata implementata seguendo il design pattern *Singleton* per avere una sola istanza della classe inoltre, il costruttore di `Configuration` è privato dunque si possono creare istanze di tale tipo solo attraverso la classe appena descritta.

2.2 Ottimizzazione della ricorsione

Per ottimizzare il calcolo del server che deve rispondere ad ogni richiesta è stata operata la modifica all'algoritmo originale descritta nella sezione "Dettagli implementativi" della dispensa. Ad ogni richiesta viene associato un limite superiore calcolato come $L_{sup} = w_{t-1}(A_{t-1}) + d(x_t, r_t)$ dove x_t è il server correntmente in analisi e $w_{t-1}(A_{t-1})$ è il costo pagato fino alla richiesta t-esima

[1] Questo è stato fatto con un `typedef`, il tipo che può essere sostituito è `range_t`.

dall'algoritmo WORK. Questo costo viene aggiornato dopo aver processato ogni richiesta.

Ad ogni iterazione della funzione lavoro viene verificata la disuguaglianza:

$$L_{sup} - \sum_{s=i+1}^{t} d(y_s, r_s) \geq D(X_i, X_0)$$

se questa non è verificata allora proseguiamo a verificare il prossimo elemento senza scendere nelle chiamate ricorsive legate al calcolo di $w_i(A_i)$. Questo permette di eliminare un buon numero di chiamate ricorsive a `work`.

3 Conclusioni

Passare dalla pura teoria a trovarsi di fronte a dover dare una implementazione effettiva di un algoritmo così complesso e con così forte ricadute teoriche non è un compito semplice. Inizialmente abbiamo dovuto ragionare su quali fossero le componenti necessarie e, dopo aver isolato quello di cui abbisognavamo abbiamo implementato il tutto.

La scelta del C++ come linguggio di implementazione è stata quasi ovvia. Una idea alternativa poteva essere Java ma questo non ci avrebbe permesso di gestire la memoria così come è stato fatto.