Relazione sull'implementazione dell'algoritmo Work

Alberto Franco, Mirko Polato, Lorenzo Tessari

2 dicembre 2011

1 Introduzione

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

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 efficienza del linguaggio e la libertà che questo offre in termini di gestione delle risorse, specie per quanto riguarda la memoria.

1.1 Struttura del progetto

Il progetto è suddiviso in due cartelle `src` e `docs`. Questo documento è stato generato dal file LaTeXcontenuto nella cartella `docs`. L'altra cartella contiene invece 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 con compilatore Mingw GCC a 32 bit e Microsoft Visual C++ sia a 32 che a 64 bit.

2 Sviluppo dell'applicazione

In questa sezione descriveremo brevemente quali sono state le scelte di design operate nell'implementare l'algoritmo. Rispetto ad una realizzazione volta solo ad ottenere efficienza si è preferito usare un disegno (e conseguentemente una programmazione) ad oggetti il più corretto possibile in modo da renderne più semplice la comprensione per gli utenti finali.

Si è scelto 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 non rende necessarie modifiche nella gestione interna del codice utente già scritto.

A scopo di test dell'algoritmo si è deciso di usare un generatore di richieste casuali (la classe `RequestGenerator`) che genera in modo uniforme le richieste nell'intervallo di default $[-50, 50]$. L'utente può però decidere di modificare al momento dell'avvio dell'applicazione la dimensione dello spazio in cui queste richieste verranno generate.

Nelle sottosezioni 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, effettua una scelta:

- 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 viene 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 viene subito richiesto, è molto probabile che questo resti in cache (o quanto meno il puntatore all'oggetto).

Si fa notare che la classe `ConfigurationFactory` è stata implementata seguendo il design pattern *Singleton* per avere una sola istanza della classe; il costruttore di `Configuration`, inoltre, 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 correntemente in analisi e $w_{t-1}(A_{t-1})$ è il costo pagato fino alla richiesta t-esima 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 Ulteriori Ottimizzazioni

Una idea iniziale era di migliorare l'efficienza dell'algoritmo usando una finestra di dimensione w controllando un numero costante di elementi ad ogni richiesta. In pratica la variazione dell'algoritmo (chiamato ω-WFA) calcola la funzione lavoro all'iterazione t partendo da $w_{t-\omega}$. Questa ottimizzazione è già stata proposta in [**?**]. In tale articolo gli autori esibiscono un risultato empirico del fatto che ω-WFA preservi la competitività di WORK, ma tale dimostrazione risulta poco accettabile a livello matematico e viene infatti smentita. Una prova della non-competitività di ω-WFA è data in [**?**]. Qui ne presentiamo una versione ridotta.

Teorema 1. *ω-WFA non è α-competitivo.*

Dimostrazione. Per il caso in cui $\omega = 1$ la dimostrazione è banale dato che $1-$WFA è l'algoritmo Greedy. Ci limitiamo al caso $\omega \geq 2$. Sulla base di queste assunzioni costruiamo uno scenario che ci permette di confutare l'affermazione che ω-WFA sia α-competitivo. Il nostro scenario è composto da un'"isola" composta da due locazioni x_1 e x_2 e da un "entroterra" in cui vi sono tutte le altre locazioni. Se scegliamo una sequenza di richieste in modo siano localizzate alternativamente in x_1 e x_2, l'idea è che ω-WFA muoverà solo i server sull'isola lasciando fermi quelli nell'entroterra. Per dimostrarlo individuiamo due casi:

1. $1 \leq i \leq \omega$. In questo caso la funzione lavoro si comporta in modo canonico perciò in questa sezione l'algoritmo coincide con Work

2. $i > \omega$. In questa sezione l'algoritmo non tiene conto di tutte le richieste ma considera $r_{t-\omega+1}$ come la prima richiesta. In questo caso l'algoritmo sceglie sempre una configurazione che muove solo i server che sono nell'isola perché il costo pagato per queste è minore rispetto a quella in cui un server dell'entroterra viene mosso sull'isola.

Se chiamiamo δ la distanza tra x_1 e x_2 e Δ la distanza tra x_1 e la prima locazione dell'entroterra, allora per una sequenza di richieste di lunghezza n tale che $n > \Delta/\delta > \omega$ l'algoritmo diventa non-competitivo. □

Una supposizione ragionevole è che questa ottimizzazione non funzioni per spazi metrici qualsiasi ma che con le dovute restrizioni la competitività venga preservata. In primo luogo considereremo uno spazio metrico finito, la cui distanza massima è Δ; consideriamo inoltre una discretizzazione di questo spazio tale che la distanza minima tra le richieste sia δ. In generale queste assunzioni non sono limitanti dal momento che le implementazioni effettive sul calcolatore hanno intrinsecamente questi vincoli.

La prima scelta che operiamo è la dimensione della finestra: scegliamo $\omega > \Delta/\delta$ e $\omega > k$, dove k è il numero di server a disposizione[1].

[1] In generale un valore di ω ragionevole è $\omega = \Delta/\delta + 1$

Il nostro algoritmo andrà a calcolare la funzione lavoro esattamente nello stesso modo in cui viene calcolata nell'algoritmo WORK con la differenza che alla richiesta t-esima avremo:

$$work(t, A_t) = \begin{cases} w_t(A_t) & \text{se } t < \omega \\ w_\omega(A_t) & \text{se } t \geq \omega \end{cases}$$

Andiamo ad effettuare una analisi sull'esempio fornito da [**?**] e supponiamo che la nostra configurazione sia la stessa. Abbiamo un'"isola" composta dalle posizioni ammissibili x_1, x_2 in cui arrivano le richieste con un server nella posizione x_1 (intercambiabilmente in x_2) e un "entroterra" in cui vi sono tutti gli altri server a distanza Δ. Supponiamo di aver scelto $\omega = \Delta/\delta + 1$, allora alle prime $\omega - 1$ richieste ω-WFA pagherà un costo pari a $\delta\omega$ ed alla w-esima sposterà un server dall'entroterra pagando un costo totale $(\omega - 1)\delta + \Delta$ mentre OPT off-line paga un costo Δ partendo dalla stessa configurazione. Perciò comparando i costi otteniamo che

$$\frac{(\omega - 1)\delta + \Delta}{\Delta} \leq (2k - 1) \Rightarrow 2 \leq (2k - 1)$$

Dunque in questo scenario l'algoritmo paga un costo inferiore a $(2k - 1)C_{OPT}$.

4 Conclusioni

Passare dalla pura teoria alla pratica e trovarsi di fronte a dover dare un'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 gli elementi necessari, abbiamo implementato il tutto.

La scelta del C++ come linguaggio 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.