Elaborato intermedio Parallel Computing - Password decryption in c++ ed Open MP

Alessandro Arezzo
E-mail address
Alessandro.arezzo@stud.unifi.it

Abstract

L'elaborato presentato è stato realizzato come progetto intermedio dell'insegnamento di Parallel Computing previsto dal corso di laurea magistrale in ingegneria informatica dell' Università degli studi di Firenze. Il lavoro consiste nell'implementazione di un decriptatore di hash codificate con algoritmo DES. L'obiettivo dell'elaborato è quello di confrontare i costi computazionali necessari per l'esecuzione di una sua implementazione sequenziale in linguaggio C++ rispetto ad una sua variante parallela sviluppata mediante l'ausilio del framework Open MP.

1. Introduzione

L'elaborato consiste nell'implementazione di un decodificatore di password che utilizzi l'algoritmo di cifratura **DES (Data Encryption Standard)** ai fini della criptazione e della conseguente decriptazione. Tale standard utilizza una crittografia a chiave simmetrica e dei caratteri detti salt, i quali per lo svolgimento del presente elaborato si sono supposti essere la stringa predefinita "aa" per facilitare la procedura. In aggiunta a ciò, ulteriore semplificazione è data dal fatto che si sono considerate solamente le password appartenenti all'insieme di caratteri [a-zA-Z0-9./].

L'implementazione presentata prevede che, ai fini della decriptazione di una data stringa codificata, questa venga confrontata con la codifica DES di ogni parola contenuta in un file .txt di input. Nel qual caso si rilevi quindi che la codifica di una certa parola memorizzata in tale file è identica alla stringa da decriptare, tale termine viene identificato come la password cercata e la decriptazione si considera quindi essere andata a buon fine.
Il progetto, scritto in linguaggio C++, prevede che tale procedura sia implementata in una versione sequenziale ed in una versione parallela che utilizza il framework Open MP.
I tempi di esecuzione necessari ai due metodi per decriptare determinate parole, scelte sia come termini localizzati in particolari posizioni del dataset sia selezionati casualmente al suo interno, sono stati poi confrontati per mezzo della generazione di relativi grafici che ne mostrano lo speedup.

2. Implementazione

Il codice del presente elaborato prevede la definizione di una classe principale denominata Decrypter, all'interno della quale vi è l'implementazione dei metodi che consentono di eseguire la decriptazione di un vettore di stringhe codificate ricevuto come parametro. Per la classe in questione sono definiti due attributi necessari ai fini della codifica di una parola mediante l'applicazione dell'algoritmo DES. Il primo di questi è il salt mentre il secondo consiste in un vettore di stringhe all'interno del quale sono memorizzate le parole contenute nel file .txt di input, il quale a sua volta contiene tutte le possibili password da analizzare per la ricerca in fase di decodifica. In relazione a ciò occorre specificare come la lettura del suddetto file ed il successivo

inserimento dei termini memorizzativi nel vettore in questione avvenga all'interno del costruttore della classe.
Quest'ultima implementa poi due metodi, i quali permettono di decriptare delle hash ricevute come parametro rispettivamente in maniera sequenziale, utilizzando l'algoritmo brute force, e parallela, per mezzo dell'ausilio del framework Open MP.
Entrambi i metodi, oltre alle parole da decriptare, ricevono un ulteriore parametro relativo al numero di test da effettuare per ciascuna parola. Ogni termine da decodificare, infatti, viene decriptato per un numero di volte pari a tale valore ricevuto ed il tempo computazionale richiesto ai fini della relativa decriptazione viene calcolato come la media dei tempi rilevati da ciascuna computazione. In merito a ciò, inoltre, risulta opportuno specificare che per prelevare il tempo corrente all'inizio ed al termine dell'esecuzione dei due metodi è stato utilizzato il metodo *now()* definito nella classe *steady_clock* della libreria *chrono*.

2.1. Versione sequenziale

Il metodo atto a decriptare un insieme di stringhe sequenzialmente prevede l'applicazione dell'**algoritmo brute force**. Il codice in questione confronta cioè ciascun termine da decriptare con la codifica DES di ogni parola del dataset. Se viene rilevata la corrispondenza tra la codifica da decriptare e quella di una parola del dataset allora il termine in questione viene identificato come la decodifica cercata. Per generare l'hash delle password contenute nel dataset viene utilizzata la funzione *cript()* della libreria *std::unistd.h*, la quale riceve come argomenti la stringa da codificare ed i termini di salt richiesti dall'algoritmo DES.
Per ciascuna parola la decriptazione viene eseguita per un numero di volte pari al numero di test da effettuare ricevuto in input e per ciascuna esecuzione viene calcolato il tempo computazionale richiesto come il tempo che intercorre tra l'inizio del for che scorre ogni parola del dataset e la sua conclusione. Al termine dell'esecuzione di tutti i test, il tempo medio viene aggiunto ad un vettore di oggetti di tipo long, il quale, al termine della computazione di tutte le decodifiche richieste, viene restituito al chiamante.

2.2. Versione parallela

Il metodo che consente di parallelizzare l'approccio sequenziale sopra descritto utilizza il framework Open MP. Tale versione implementata, si basa sulla suddivisione dei dati da computare in chunks. Le parole del dataset da analizzare, di cui cioè si vuole confrontare la relativa codifica con l'hash da decodificare, vengono cioè suddivise in gruppi aventi stessa dimensione. Ciascun gruppo viene quindi assegnato alla computazione di un determinato thread tra quelli generati, in cui il numero di threads in questione viene specificato al metodo come parametro.

La suddivisione in chunks avviene per mezzo dell'utilizzo del numero di ciascun thread (prelevato utilizzando la funzione *omp_get_thread_num()* del file *omp.h*) e del calcolo del numero di parole che ogni thread deve computare (calcolato come il rapporto tra il numero totale delle parole del dataset ed il numero dei threads). L'offset delle parole del dataset da assegnare a ciascun thread viene quindi calcolato come il prodotto tra il numero del thread corrente e la dimensione dei chunks.
Il codice viene quindi parallelizzato per mezzo della direttiva *#pragma omp parallel* con la specifica della clausola che definisce il numero di threads da generare. Nel corpo di tale costrutto viene implementato l'algoritmo di ricerca brute force così come fatto in sequenziale, ma con la particolarità che ogni thread generato computa solamente le parole del dataset appartenenti al chunk assegnatogli secondo la suddetta modalità appena descritta.

Affinchè sia possibile far comunicare i threads, permettendo così a ciascuno di questi di verificare se qualche altro thread ha trovato la decodifica cercata, si utilizza poi un flag (variabile booleana) volatile e condiviso tra tutti i threads. Tale flag viene inizializzato inizialmente sul valore di false per poi essere settato su true solamente nel momento in cui un thread trova la stringa cercata.

All'interno del for di ricerca di ogni thread quindi:

- Se il flag è ancora settato su false e se l'indice non ha superato il limite dettato dalla dimensione del dataset, la parola corrente viene codificata e confrontata con la stringa da decodificare. Se le due hash sono uguali il flag viene settato su true ed il thread esce dal for (break).
- Altrimenti il thread esce dal for (break) poichè la stringa è già stata decriptata da un altro thread oppure sono terminate le parole da computare.

Dalla descrizione appena fornita del flusso di esecuzione, si nota come tale implementazione presentata consenta di risparmiare un notevole costo computazionale rispetto ad una variante che vada ad utilizzare la direttiva *#pragma omp parallel for* sia questa in modalità statica oppure dinamica. L'approccio presentato difatti ha la particolarità di consentire ad un thread di uscire dal for (con istruzione break) nel qual caso rilevi che la codifica è già stata trovata da un altro thread, il che consente riguardo alle altre varianti, di risparmiare l'inutile esecuzione di un certo numero di iterazioni del ciclo for, il quale numero cresce al diminuire della posizione in cui è memorizzata la parola cercata all'interno del rispettivo chunk.

Particolare rilevanza è data dalla fase di codifica di ciascuna stringa del dataset, fase necessaria ad ogni iterazione per il confronto con l'hash da decodificare. La funzione *crypt()*, utilizzata per l'approccio sequenziale, la quale si può considerare una **funzione aliena** a tutti gli effetti, si rileva inutilizzabile per tale approccio parallelo in quanto non consente di definire lo spazio di memoria in cui allocare il risultato della codifica. Tale problematica infatti risulta fatale durante la computazione dal momento che può accadere che due o più threads vadano ad allocare la codifica nella stessa locazione di memoria con la conseguente perdita della prima stringa allocata. Per debellare il problema in questione, per la criptazione si è utilizzato la funzione *DES_fcrypt()* della libreria *openssl/des.h*, la quale riceve non solo la stringa da codificare ed i caratteri di salt, ma anche l'indirizzo della locazione di memoria in cui allocare il risultato. E' stato così sufficiente allocare uno spazio di memoria per ciascun thread all'interno della direttiva pragma omp parallel e passare il puntatore a tale locazione alla suddetta funzione di criptazione.

3. Dataset

Di centrale rilevanza risultano essere le parole presenti nel file .txt fornito in input al programma. Questo difatti costituisce il dataset di riferimento contenente tutti e soli i termini che è possibile decriptare. Ai fini della realizzazione del presente elaborato, è stato utilizzato il file reperibile al link https://www.kaggle.com/wjburns/common-password-list-rockyoutxt[1].
Tale dataset contiene una lista di 14 341 564 di passwords comuni, le quali sono di lunghezza variabile e contenenti anche dei caratteri speciali. Dato quindi che i termini in questione non risultano conformi alle specifiche dettate dal problema, il quale richiede che si considerino solamente parole di 8 caratteri appartenenti al set [a-zA-Z0-9./], tale file è stato filtrato per mezzo di un apposito script realizzato in linguaggio Python ed atto ad estrarne le sole password che soddisfano i requisiti. Il file risultante dal suddetto filtraggio contiene un numero di parole pari a 2 781 526 ed è stato utilizzato come dataset del progetto presentato.

4. Esperimenti e risultati

Affinchè fosse possibile confrontare i tempi computazionali richiesti per decriptare determinate passwords utilizzando l'approccio sequenziale piuttosto che la sua variante parallela si sono eseguite diverse tipologie di tests.
In ciascun test, per ogni termine selezionato per la decriptazione, gli esperimenti hanno consistito nel calcolo dello **speedup** (dato dal rapporto tra tempo richiesto per la decodifica in sequenziale e tempo necessario per quella in parallelo) al variare del numero di threads generati nella versione parallela. In particolare, ciascun test è stato ese-

guito calcolando lo speedup con i seguenti numeri di threads lanciati: 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500, 1000, 1500. I risultati ottenuti da tale calcolo degli speedup si sono riassunti attraverso dei grafici atti a descriverli e realizzati per mezzo di un apposito script Python.

In primo luogo, un' **iniziale fase di testing** ha consistito nella valutazione dello speedup per parole poste in particolari posizioni del dataset. Tali termini, che sono stati selezionati sulla base dello loro posizione nel file fornito in input, sono i seguenti:

- *password:* prima parola del dataset
- *jrock521:* parola posta in posizione $(N/2)+1$ con *N:* numero di parole del dataset
- *10022513:* ultima parola del dataset

In aggiunta a ciò, è stata eseguita un' **ulteriore parte di esperimenti** atta a rilevare lo speedup che si ottiene in media, decriptando cioè parole scelte casualmente all'interno del dataset.
Tali tests sono stati eseguiti su un terminale avente la seguente CPU: Intel Core i7 4770HQ 2,2 GHz con 4 core fisici.

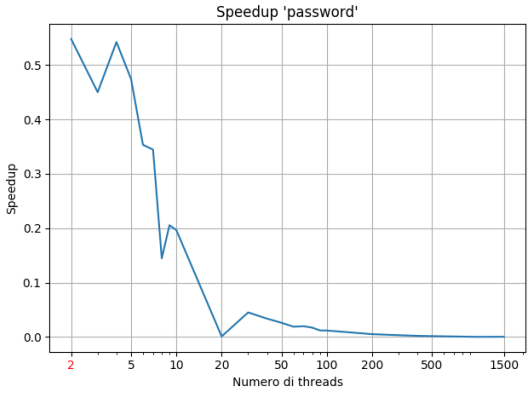

Figure 1. Speedup della parola 'password'

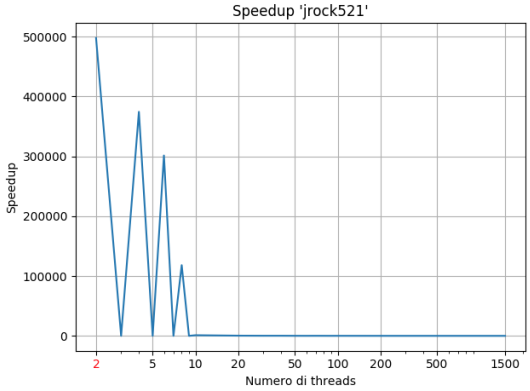

Figure 2. Speedup della parola 'jrock521'

4.1. Risultati esperimenti su passwords specifiche

Come già accennato, la prima parte di esperimenti è dedicata all'analisi delle performance dell'approccio parallelo presentato se applicato a password localizzate in posizioni specifiche del dataset. Per ciascuno dei termini selezionati e per ogni tipologia di prova (in sequenziale ed in parallelo per ogni differente numero di threads lanciati) sono stati eseguiti 50 tests e si è considerato il tempo di esecuzione come il tempo computazionale medio rilevato dai singoli esperimenti.
Il primo termine selezionato è la parola **password**, posizionata alla prima locazione del dataset utilizzato. Come si nota dal grafico in Figura 1 che mostra lo speedup dato dal rapporto tra tempo sequenziale e tempo parallelo al variare del numero di threads, l'approccio parallelo si rivela essere totalmente inefficiente. Lo speedup migliore si ottiene difatti con 2 threads, il cui relativo valore risulta comunque notevolemente inferiore ad 1. Nell'applicare quindi l'approccio parallelo per la decriptazione di termini posti all'inizio del file di passwords fornito in input si evidenzia un notevole peggioramento delle performance rispetto alla variante sequenziale. Ciò è ovviamente dovuto al fatto che quest'ultimo metodo trova in un tempo irrisorio la decodifica di termini posti in prossimità dell'inizio del dataset, tempo questo che non può in alcun modo migliorare in parallelo ma che anzi peggiora a causa dell'**overhead** richiesto dalla creazione e dalla gestione dei threads lanciati.

Per quanto concerne adesso lo speedup del termine **jrock521**, posizionato in posizione $(N/2)+1$, dall'osservazione del relativo grafico in Figura 2 si nota come le performance dell'approccio sequenziale siano notevolmente sovraperformante

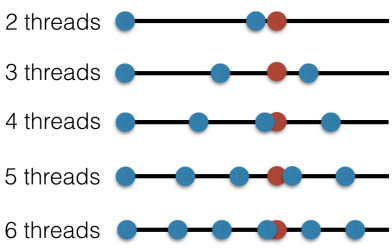

Figure 3. Schema che mostra la posizione della password 'jrock521' nel rispettivo chunk al variare del numero di threads generati

per mezzo dell'applicazione del metodo parallelo. Il migliore speedup, anche in questo caso, si denota essere con il lancio di 2 soli threads per la versione parallela, in cui il valore tocca addirittura un dato di poco inferiore a 500 000.

Tale speedup, di valore apparentemente troppo elevato, è giustificato dal fatto che per trovare una parola posta in prossimità della metà del dataset, il metodo sequenziale deve scorrere più di un milione di vocaboli prima di trovarne la decodifica. Come mostrato dall schema in Figura 3, se però il termine considerato è posto all'inizio di un chunk, come la password *'jrock521'* per casi in cui il numero di threads è un valore pari, il thread incaricato di computare il chunk in questione troverà immediatamente la decodifica, con la conseguenza che il rapporto tra tempo richiesto dalla variante sequenziale e tempo di esecuzione in parallelo restituirà un valore estremamente grande. Si nota infatti che valori così elevati, per la password in questione, si ottengono solamente se il metodo parallelo è applicato generando un numero pari di threads, mentre risulta generalmente di tipo lineare e superlineare per un numero di threads dispari. Ovviamente, all'aumentare del numero di threads generati, diminuisce il valore dello speedup a causa sia del tempo di overhead richiesto dai molteplici threads lanciati sia, nel caso di numero di threads pari, per il fatto che l'inizio di un chunk si allontana progressivamente dalla parola considerata.

Infine, la terza ed ultima password considerata in posizione rilevante che si è deciso di analizzare è data dalla parola **10022513**, posta sul fondo del

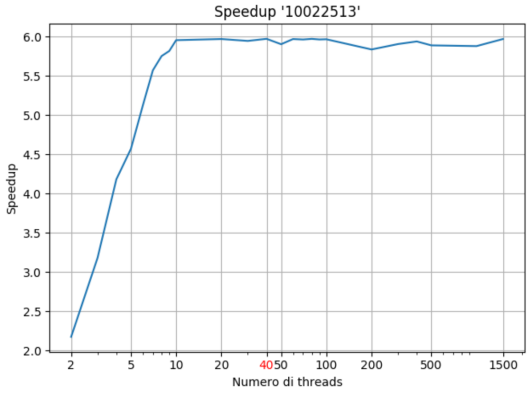

Figure 4. Speedup della parola '10022513'

dataset.
Essendo appunto il termine ultimo del file .txt, si ha che questa è la password che tra tutte quelle contenute al suo interno richiede il maggior tempo computazionale per la decriptazione per mezzo del metodo sequenziale. Dall'analisi dei risultati raffigurati in Figura 4 si nota come già solamente con 2 threads l'appproccio parallelo dimezzi il tempo di esecuzione richiesto in sequenziale e come la crescita risulti pressochè lineare fino a 10 threads lanciati. In prossimità di tale punto difatti, lo speedup arriva addirittura a toccare un valore molto vicino a 6, il che implica uno **speedup superlinare** in quanto avente valore superiore a 4, numero di core fisici della macchina utilizzata per gli esperimenti. In prossimità dei 10 threads generati, si nota che lo speedup raggiunge un punto di saturazione, dal momento che per i test effettuati con un più elevato numero di threads lo speedup ricavato rimane comunque molto vicino al suddetto valore.

4.2. Risultati esperimenti casuali

Affinchè fosse possibile comprendere le prestazioni ottenute in media dal metodo parallelo rispetto a quello sequenziale, si sono eseguiti dei tests atti a decriptare 200 parole posizionate in locazioni scelte casualmente nel dataset. Per ciascuna delle passwords scelte, per motivi correlati all'elevato tempo computazionale richiesto, si è eseguito un solo esperimento, senza cioè calcolare il tempo medio ottenuto da più esperimenti

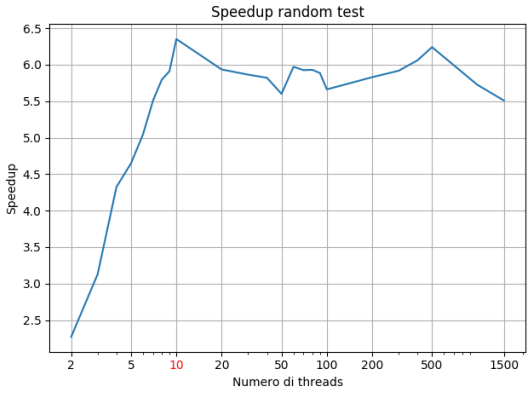

Figure 5. Speedup di 200 parole scelte casualmente nel dataset

per ciascuna hash.

Dall'analisi dei risultati prodotti, il cui grafico correlato è raffigurato in Figura 5, si può notare come le performance medie rilevate risultino estremamente simili a quelle relative alla decodifica dell'ultima parola del dataset '10022513'. Come per quest'ultima difatti, lo speedup aumenta pressochè linearmente fino a 10 threads, numero per il quale si denota uno **speedup superlineare**, per poi rimanere sempre superiore al valore di 5.5 per un maggiore numero di threads lanciati.

5. Conclusioni

In conclusione, come si evince dall'analisi dei risultati elargita nel precedente capitolo, la valutazione delle prestazioni dell'approccio parallelo proposto per il presente elaborato dipende dalle diverse casistiche considerate. Per parole posizionate all'inizio del dataset, il metodo parallelo si rivela cioè totalmente inefficiente, come è ovvio che sia dal momento che il parallelismo induce notevoli vantaggi quando la computazione avviene su un elevato numero di dati. Le performance del metodo, migliorano difatti per parole posizionate nella seconda metà del dataset. Lo speedup raggiunge valori superlineari intorno ai 10 threads per l'ultima parola, ed addirittura valori ancora estremamente superiori per vocaboli posizionati all'inizio dei chunks.

Interessante risulta poi essere la denotazione di una stretta correlazione tra i risultati che si ottengono in media decodificando parole scelte arbitrariamente e quelli ricavati dalla decriptazione dell'ultima parola del dataset, il che permette di comprendere come l'approccio parallelo induca in generale notevoli vantaggi in termini di tempo computazionale richiesto, consentendo cioè di ottenere uno speedup anche superlineare in media lanciando un numero di threads superiore rispetto a quelli del terminale sul quale si sta eseguendo il processamento.

References

[1] Common password list - rockyou.txt. https://www.kaggle.com/wjburns/common-password-list-rockyoutxt.