Elaborato Intermedio PC: decriptazione password

Bernardo Tiezzi
bernardo.tiezzi@stud.unifi.it

Abstract

La seguente relazione descrive l'elaborato scelto come progetto intermedio per l'esame di Parallel Computing, appartenente al corso di laurea magistrale in Ingegneria Informatica presso l'Università degli Studi di Firenze. Il progetto, implementato in linguaggio di programmazione Java 8, consiste nella realizzazione di un decriptatore di hash codificate impiegando l'algoritmo di tipo DES (Data Encryption Standard). Lo studio condotto analizza la variazione, in termini di costo computazionale, tra l'implementazione sequenziale e parallela del processo di decriptazione, utilizzando le librerie di Java per consentire un'esecuzione thread-safe del codice.

1. Introduzione

Il progetto consiste nell'implementazione di un decodificatore di password criptate attraverso l'algoritmo di cifratura **DES** al fine di simulare fasi di criptaggio e decriptaggio, di fondamentale importanza per la sicurezza informatica. Il sistema **DES** esegue una codifica di una stringa `x` da 64 bit in una stringa *y* sempre da 64 bit, utilizzando una chiave di cifratura a 56 bit (64 in realtà, ma gli ultimi 8 sono bit di parità). Per semplicità sono state utilizzate soltanto password di 8 caratteri di tipo [a-zA-Z0-9\.]. Il codice sorgente realizzato prevede di decodificare una stringa di caratteri alfanumerici, appartenente ad un dataset di password salvato su un file `.txt`. La password scelta viene confrontata con l'intero dataset; il processo si conclude quando si riscontra l'uguaglianza tra la codifica di una stringa del dataset e la password ricercata. Il costo computazionale del codice sorgente implementato dipende principalmente dalla ricerca eseguita sul dataset. La realizzazione di una versione sequenziale ed una versione parallela del ciclo di iterazioni è necessaria al fine di fornire un'adeguata analisi delle performance per la fase di decodifica. Entrambe le versioni sono scritte utilizzando il liguaggio di programmazione `Java 8`: in aggiunta per la parallelizzazione del processo di ricerca sono definite strutture dati per l'accesso concorrente e la comunicazione tra threads. I risultati ottenuti sono riportati sotto forma di grafici che rappresentano lo speedup (rapporto tra tempo di esecuzione sequenziale e parallela del processo) ottenuto in base al numero di threads utilizzati.

2. Codice Sorgente: Implementazione

Per la progettazione dell'elaborato è stato utilizzato l'ambiente di sviluppo `Netbeans`, versione 8.2. L'algoritmo di cifratura **DES** è stato definito all'interno della classe `DesEncrypter`. Quest'ultima presenta come attributi due istanze della classe `Chiper`, `echiper` e `dchiper`, necessari ai fini della fase di criptaggio e decriptaggio. Al costruttore viene passata una `SecretKey` definita utilizzando il metodo API `generatekey()` appartenente alla classe `KeyGenerator`; la chiave viene utilizzata per inizializzare correttamente i due chiper, definendo inoltre il tipo di algoritmo criptografico utilizzato (*DES, AES, TripleDES, ...*). Il metodo `String encrypt(String str)` trasforma la stringa ricevuta in ingresso in un array di byte usando il charset scelto (UTF-8 nel nostro

caso); in seguito viene eseguito il criptaggio attraverso il metodo `.doFinal(byte [] b)` di `echiper`. La stringa codificata viene generata e restituita dopo aver codificato il byte in base64. La decriptazione viene eseguita dal metodo `String decrypt(String str)` rimuovendo la codifica in base64 dall'array di byte associato alla stringa `str`. Analogalmente al metodo `encrypt()` viene decriptato l'array e riportato il valore della stringa originale. La versione sequenziale implementa l'algoritmo di **ricerca esaustiva** (brute force) e viene ripetuta più volte in modo da poter fornire una valutazione più accurata del costo computazionale effettivo. La parellelizzazione della ricerca richiede di definire come parametro aggiuntivo il numero di threads da processare. Le tempistiche vengono calcolate sfruttando il metodo API `System.nanoTime()` che riporta in formato Timestamp il tempo corrente, espresso in ns.

2.1. Dataset

Per eseguire qualunque metodo di decriptazione è fondamentale possedere un dataset di stringhe, utilizzabili come password, all'interno del progetto. Questo deve essere fornito in input al programma all'inizio di ogni ciclo d'esecuzione. Il dataset utilizzato è disponibile al link seguente `https://www.kaggle.com/wjburns/common-password-list-rockyoutxt` [1] . Il dataset è composto da $14,341,564$ di passwords uniche, tra quelle maggiormente utilizzate, con dimensioni variabili e possibili caratteri speciali. Il file originale `rockyou.txt` non risulta conforme alle richieste specificate nel comando del progetto; per ovviare al problema è stato opportunamente implementato un script in linguaggio Python che riceve in ingresso il file scaricato ed inserisce, in un file di uscita, le stringhe alfanumeriche di 8 caratteri. Come fase a priori alla stesura del codice sorgente è stato quindi generato il nuovo dataset `dataset_password.txt`, contenente $2,781,526$ stringhe.

2.2. Versione Sequenziale

Il metodo corretto per decriptare sequenzialmente un insieme di passwords prevede l'applicazione del metodo di **ricerca esaustiva**. In fase di **inizializzazione** il file `dataset_password.txt` viene letto ed i dati contenuti vengono inseriti in un array di stringhe. La password campione per l'esecuzione corrente del programma è selezionata tra le stringhe contenute all'interno dell'array. Sono istanziate inoltre la chiave di cifratura e la classe `DesEncrypter`. La password selezionata viene quindi criptata con algoritmo di cifratura **DES**. La **ricerca esaustiva** viene implementata eseguendo un ciclo `for()` sulla lunghezza complessiva del dataset; ad ogni iterazione viene effettuato un confronto tra le codifiche della password obiettivo e la parola corrente dell'array; nel caso risulti positiva l'uguaglianza, il ciclo viene interrotto e il termine confrontato del dataset viene mandato in stampa al programma. Sia all'inizio del ciclo `for()`, sia al termine, sono salvati i valori `TimeStamp` su due variabili di tipo `double`, di cui viene fatta la differenza in modo da ottenere il tempo di esecuzione della fase di ricerca. La ricerca viene eventualmente ripetuta più volte per una stringa in particolare oppure ogni volta su una stringa differente, in modo da ottenere rispettivamente il tempo medio per decriptare una determinata password e per decriptare una password generica.

2.3. Versione Parallela

Per eseguire la parallelizzazione del processo di ricerca viene estesa la classe `Thread` di Java, definendo la classe derivata `DesEncrypterThread`. L'implementazione corretta prevede di suddividere, in blocchi di uguale dimensione, il dataset delle passwords iniziale: ogni blocco viene assegnato ad un thread specifico. La fase di **inizializzazione** del processo è simile al caso sequenziale, definendo l'array di stringhe, il decodificatore e la password campione. In aggiunta, per consentire ai threads allocati di comunicare tra loro,

viene definita una variabile `numPasswords` di tipo `AtomicInteger`; la caratteristica di tali strutture dati è l'atomicità di ogni operazione eseguita su di esse. I threads, grazie a `numPasswords`, sono in grado di conoscere in realtime lo stato della ricerca. Per l'esecuzione viene utilizzata la classe `Executor`, la quale sostituisce l'istanziazione classica della classe Thread consentendo la processazione di pool di threads. La definizione di un `executor` permette di riutilizzare ogni thread appartenente al pool; infatti la terminazione avviene solo attraverso lo specifico comando `.shutdown()`. Nel caso descritto viene utilizzato il metodo di factory `newFixedThreadPool(int n)`, che definisce un pool di threads di dimensione fissa, possibilmente riutilizzabili. La classe `DesEncrypterThread` viene istanziata all'interno di un ciclo `for()`, utilizzando il metodo della classe executor `submit(new Thread())`, per un numero di volte pari alla dimensione del pool; viene avviato il processo di ricerca parallela. Al costruttore `DesEncrypterThread()` vengono passati come parametri il nome del thread istanziato (stringa numerica), il dataset inizializzato, la password criptata da ricercare, la variabile atomica `numPasswords` e la dimensione del pool realizzato. All'interno del metodo `run()` è istanziata la classe del decodificatore DES e definiti gli indici di `inizio` e `fine` del ciclo `for()`. Analogalmente al caso sequenziale le codifiche della password obiettivo e dell'elemento corrente all'n-esima iterazione vengono confrontate. Durante l'esecuzione, l'accesso alla variabile `numPasswords` avviene a seconda dello stato del processo:

- nel caso in cui l'uguaglianza tra le codifiche sia verificata, `numPasswords` viene modificato usando il metodo API `.decrementAndGet()`, operazione thread-safe che riduce di un'unità il valore della variabile.
- quando ogni password richiesta è stata decodificata, il ciclo viene interrotto e l'esecuzione del thread termina. Per garantire l'ottimalità dei tempi di ricerca viene effettuato un confronto, ad ogni iterazione, sul valore di `numPasswords`; se quest'ultimo risulta pari a 0, viene invocato il comando `break` che forza l'uscita dal `for()`.

La descrizione fornita per il flusso di esecuzione, nel caso multi-threading, evidenzia il notevole risparmio sui tempi, ottenuto grazie a questa implementazione. Infatti, sfruttando le variabili atomiche, quando la decodifica della password è stata recuperata il ciclo viene interrotto su ogni thread appartenente al pool. Il costo risulta perciò inferiore rispetto ad un'implementazione in cui si attende che ogni thread abbia terminato la ricerca sull'intero blocco di appartenenza. Il miglioramento effettivo dipende dalla posizione della password decodificata all'interno del `chunk` associato ad un thread; più alta è la priorità della `word`, maggiore il numero di operazioni inutili risparmiate. Per attendere che ogni thread abbia concluso la processazione dei dati, vengono invocati due metodi appartenenti due metodi appartenenti alla classe `executor`: `shutdown()` e `awaitForCompletation()`. Le misurazioni dei tempi vengono effettuate prima del ciclo `for()` di inizializzazione e subito dopo la terminazione del pool. La differenza tra i tempi di inizio e fine viene inserita in una `Hashmap` avente come chiave il numero di threads inizializzati.

3. Esperimenti ed analisi dei Risultati

Per ottenere dei risultati che consentissero di effettuare un'analisi adeguata della differenza di prestazioni dell'algoritmo di decifrazione, utilizzando l'approccio sequenziale e l'approccio parallelo, è stato necessario eseguire più tipologie di test. Il confronto tra le varianti di codice è presentato attraverso lo studio dello **speedup** (il rapporto tra i tempi i esecuzione nel caso sequenziale e nella variante parallela), calcolato in base alla dimensione del pool di threads utilizzata. Negli esperimenti condotti è stato calcolato lo speedup per il seguente numero di threads allocati: 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60, 70, 80, 90,

100, 128, 256, 512, 1024. I primi esperimenti effettuati avevano come obiettivo l'analisi della decriptazione di passwords poste in posizioni specifiche all'interno del dataset iniziale. La scelta è ricaduta sulle stringhe di seguito elencate:

- ***password***: stringa posta in prima posizione all'interno del dataset.
- ***jrock521***: stringa posta in posizione $(N/2) + 1$ all'interno del dataset.
- ***10022513***: stringa posta in ultima posizione all'interno del dataset.

Si può notare a priori che lo speedup riportato dagli esperimenti condotti su ogni password risulterà differente. Per offrire una stima dello speedup generico, ottenuto dal programma, è stato effettuato un ulteriore test, riportando la media dei risultati ottenuti dalla ricerca di una password casuale contenuta nel file `dataset_password.txt`. Ogni test è stato effettuato su un dispositivo avente la seguente CPU: Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz, con 2 cores fisici.

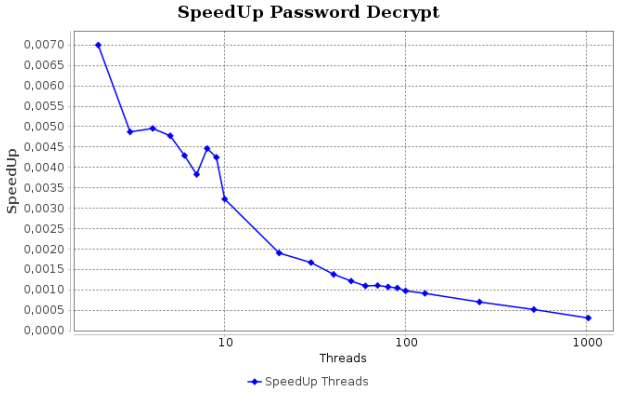

Figure 1. Grafico first Password

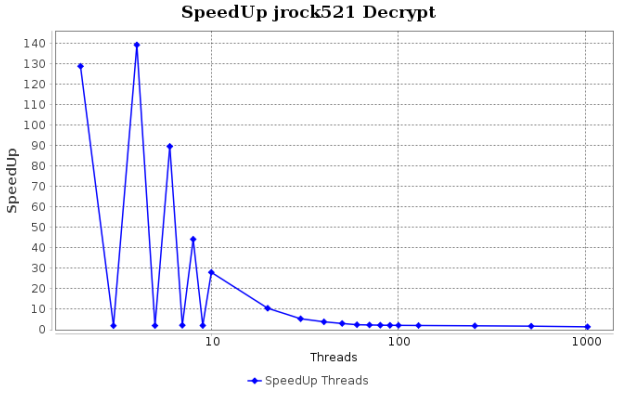

Figure 2. Grafico middle Password

3.1. Esperimenti su password specifiche

Come già enunciato in precedenza, le prime tipologie di test analizzano lo speedup ottenuto dalla decriptazione di hash relative a passwords individuate in specifiche posizioni del dataset. Per ogni stringa considerata l'algoritmo viene testato 25 volte, sia nel caso sequenziale che nella versione multithreading (per ogni dimensione scelta del pool); viene successivamente ricavata una media dei tempi di esecuzione. Il primo termine considerato è la parola **`password`**, che occupa la prima posizione del dataset. Come si può notare dal grafico in figura 1, che riporta lo speedup ottenuto al variare del numero di threads utilizzati, la parallelizzazione si rivela completamente inefficiente. Infatti il miglior risultato si ottiene con 2 threads allocati; tuttavia lo speedup è fortemente inferiore al valore 1. Questo perchè la ricerca sequenziale, dopo aver salvato su un apposito array il dataset iniziale, individua immediatamente la coincidenza tra le codifiche della password ricercata con la stringa iniziale dell'array. Questo risultato non può essere migliorato dalla parallelizzazione del processo; al contrario si assiste ad un peggioramento dei tempi di calcolo dovuto all'*overhead* richiesto dall'inizializzazione e la gestione dei threads eseguiti. Analizzando invece il grafico in figura 2, viene mostrato l'andamento dello speedup rilevato durante la ricerca della password **`jrock521`**, posta in posizione $(N/2) + 1$ all'interno del dataset. É immediato notare che le prestazioni ottenute durante l'esecuzione sequenziale della decodifica siano sovraperformate dall'implementazione multithreading. I valori più elevati dello speedup sono ottenuti con l'esecuzione di 2 e 4 threads; infatti il costo computazionale è ridotto di almeno due ordini di grandezza (lo speedup è almeno 100 volte superiore). Ovviamente i valori ottenuti sono troppo alti per essere considerati adatti a fornire una de-

scrizione generale della decodifica. Quello individuato è un caso particolare: la spiegazione di un tale evento è dovuta alla necessità per l'algoritmo di decodifica sequenziale di scorrere metà del dataset, che contiene quasi 3 milioni di stringhe, per individuare la password obiettivo. Al contrario, quando vengono istanziati 2 threads, il secondo thread individua come prima stringa del blocco assegnato la password *jrock521*. Il ragionamento è valido in generale quando la dimensione del pool di threads eseguiti è pari. La dimostrazione di quanto affermato si evince direttamente dal grafico di figura 2, che presenta un gap estremamente ampio quando il numero di threads varia da pari a dispari; pool di threads dispari rilevano uno speedup lineare. Ovviamente, quando il numero di threads cresce, si verifica una riduzione delle prestazioni causata dai ritardi dovuti all'overhead richiesto per l'inizializzazione dei processi paralleli. Infine l'ultima parola considerata nella ricerca di password specifiche è **`10022513`**, situata in ultima posizione all'interno del dataset. Essendo di fatto l'ultima parola allocata all'interno del file `dataset_password.txt`, risulta essere l'ultima password su cui l'algoritmo sequenziale esegue il confronto; dunque la decodifica di 10022513 richiede il tempo di esecuzione maggiore. Il grafico dello speedup è mostrato in figura 3; le prestazioni più elevate si ottengono quando il numero di threads allocati è piccolo. Quando vengono istanziati due threads lo speedup ottenuto è di poco inferiore a 2; dopo una lieve crescita si assiste ad una fase di "assestamento" attorno al valore di soglia 2. Si è infatti raggiunto il limite di crescita lineare, dovuto alla presenza di due soli core fisichi. In seguito lo speedup si riduce a causa del ritardo dovuto alla gestione di elevati processi paralleli.

3.2. Risultati esperimenti casuali

La ricerca di password specifiche offre comunque una visione particolare e limitata del problema. Per conoscere effettivamente le prestazioni che il multithreading consente di ottenere è logico pensare ad una fase di test più generica. Sono

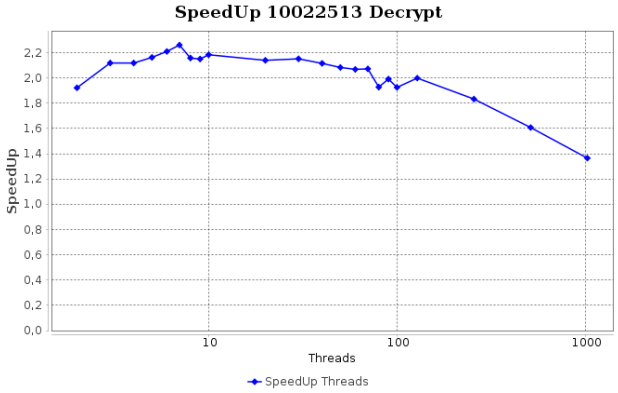

Figure 3. Grafico last Password

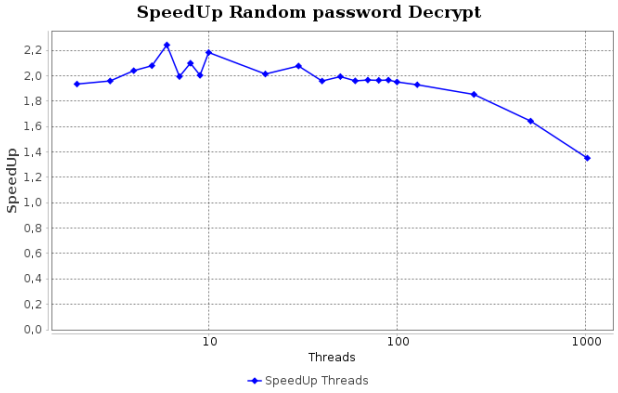

Figure 4. Grafico random Password

stati dunque eseguiti dei test atti a decriptare un numero massimo di 200 parole **casuali** del dataset, ottenute utilizzando la funzione pseudo-randomica `Math.random()`. Ovviamente per ogni password scelta è stato eseguito un solo test, in modo da ridurre i costi computazionali e facilitarne l'esecuzione. Il grafico riportato in figura 4 presenta un comportamento molto simile alla decodifica della stringa '10022513'. Anche in questo caso lo speedup presenta una crescita lineare assestandosi attorno al valore corrispondente al numero di core fisici. La riduzione di velocità è comunque lenta al crescere del numero di threads.

3.3. Considerazioni finali

Grazie all'analisi condotta è possibile ottenere una visione più o meno generale degli effetti che il multithreading comporta quando si sostituisce alla processazione sequenziale. Quando si vuole ricercare un termine posto all'inizio di un dataset,

di dimensione arbitraria, è sconsigliato sfruttare la programmazione in parallelo; essa infatti conduce a ritardi gravosi causati dall'overhead. La situazione cambia quando viene ricercata una stringa appartenente alla seconda metà del dataset di riferimento. La crescita dello speedup in questo caso è lineare ed aumenta ulteriormente quando la stringa ricercata si trova presso le ultime posizioni del dataset. L'aspetto più interessante è l'analogia riscontrata nello speedup ottenuto ricercando l'ultima parola del dataset ed una parola scelta a caso. Questa somiglianza può indicare che in generale, qualunque password si desideri decifrare, la probabilità che le prestazioni fornite dal multithreading processing siano simili a quelle che otterremo ricercando l'ultima password del dataset, è elevata. Un limite che è stato riscontrato durante l'implementazione del codice riguarda la codifica in Base64, utilizzata nella crittografia DES; altro limite importante riguarda le caratteristiche fisiche del terminale su cui vengono eseguiti gli esperimenti. Sarebbe interessante eseguire dei test, consentendo all'algoritmo di ricerca esaustiva, sia nella sua versione sequenziale che parallela, di rimuovere la codifica Base64 ed eseguire il confronto tra bytes che spesso risulta più performante del confronto tra stringhe.

References

[1] W. J.Burns. Common password list. https://www.kaggle.com/wjburns/common-password - list-rockyoutxt.