Primo homework

Gabriele Atria Michele Laurenti

12 dicembre 2015

1 Algoritmo ideale

Trovare una password a forza bruta, conoscendone solo l'hash, equivale a cercare una stringa di lunghezza non determinata su un alfabeto. Chiamiamo Σ l'alfabeto su cui è costruita la stringa, e $\xi = |\Sigma|$ il numero di caratteri nell'alfabeto. La password che cerchiamo è una parola di $\Sigma^\star$ (l'insieme di tutte le parole costruite sull'alfabeto Σ).

L'idea algoritmica è la seguente: si enumerano tutte le stringhe in ordine quasi lessicografico (ordinandole prima per lunghezza, poi lessicograficamente), e si controlla se ciascuna di queste è la password cercata. Se n è la lunghezza della password, questo approccio permette di trovarla controllando non più di $\sum_{i=1}^{n} \xi^i \simeq \xi^n$ stringhe.

L'implementazione ideale è scegliere di creare un numero "infinito" di thread, assegnando a ciascuno una parola, e schedulare per primi i thread a cui è stata data una parola che viene prima nel nostro ordinamento. Il programma termina nel momento in cui un thread trova la stringa corrispondente alla password.

Possiamo rappresentare l'esecuzione dell'algoritmo ideale con un DAG (diretto verso il basso), in cui ogni nodo rappresenta un thread, e un arco orientato (a, b) rappresenta la relazione "a termina prima dell'inizio di b".

Con un solo processore a disposizione l'algoritmo visita i nodi da quello corrispondente alla parola più piccola a quello della password. Possiamo quindi pensare il DAG con un solo processore come un cammino (figura 1). Assumendo che controllare se una stringa corrisponde alla password cercata abbia costo costante[1], il *work* (tempo di esecuzione sequenziale) è $O(\xi^n)$.

Con p processori, invece, il DAG di esecuzione di questo algoritmo possiamo pensarlo come un albero con p figli, e ciascuno di questi figli è radice di un cammino lungo $\frac{\xi^n}{p}$ (figura 2).

[1] In realtà l'algoritmo di hash avrà probabilmente un costo in funzione di l, la lunghezza della stringa di cui viene calcolato l'hash.

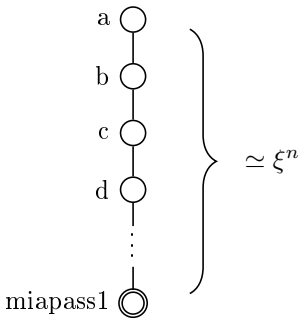

Figura 1: Il DAG di esecuzione dell'algoritmo sequenziale è un cammino lungo circa ξ^n.

Quindi il tempo di esecuzione con p processori è $O\left(\frac{\xi^n}{p}\right)$. Quindi lo speedup è lineare con p.

Lo *span*, ossia il tempo di esecuzione con un numero infinito di processori, è $O(1)$: viene creato un numero infinito di thread, ciascuno su un processore, e ciascun thread controlla una sola parola.

1.1 Ottimizzazione dell'algoritmo ideale

La situazione illustrata è ideale e non prende in considerazione l'impossibilità pratica di creare infiniti thread. L'implementazione che proponiamo dell'algoritmo cerca però di avvicinarsi alla situazione ideale, in cui la sincronizzazione non ha costi:

1. Ciascun thread non lavora su una singola parola alla volta, ma su un insieme di parole di cardinalità fissata, per ridurre il numero di thread totali. È quindi necessario trovare una partizione di $\Sigma^\star$ le cui classi abbiano tutte la stessa cardinalità, e che sia possibile ordinare linearmente in modo che la classe A preceda la classe B se e solo se tutte le parole di A precedono in ordine quasi lessicografico tutte le parole di B. Se le due proprietà sono verificate gli elementi dell'insieme quoziente saranno insiemi di parole verificabili nello stesso tempo e ordinabili "rispettando" l'ordine quasi lessicografico delle parole.

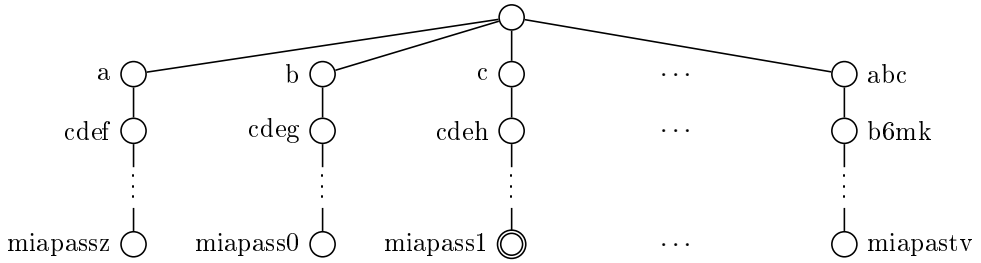

Figura 2: Il DAG di esecuzione dell'algoritmo parallelo è un albero che ha come figli della radice p cammini lunghi ciascuno $\frac{\xi^n}{p}$.

La partizione che proponiamo inserisce in una classe tutte le parole lunghe k che hanno in comune i primi j caratteri, dove $j < k$. La differenza $k - j$ deve essere un'invariante per tutte le classi di questa partizione.

2. Non creiamo un numero infinito di thread, che è solo una utile astrazione mentale, ma ne creiamo un numero m che ripetono il lavoro finché non trovano la password, e cerchiamo per quali valori di m si ottiene lo speedup maggiore. In questa implementazione deve esistere un metodo che atomicamente ritorna una diversa stringa su Σ in ordine quasi lessicografico, o meglio una classe di stringhe come da punto 1. Ciascun thread chiama questo metodo e controlla l'insieme di stringhe che riceve.

2 Implementazione

2.1 Implementazione ingenua

La prima implementazione che proponiamo ricalca grossomodo l'algoritmo ideale. Rappresentiamo ogni parola con una `String`, e attraverso due generatori di stringhe (`AllStringGenerator`, globale e sincronizzato, e `StringGenerator`, locale a ciascun thread) forniamo ai thread creati un "inizio" di parola e lasciamo loro il compito di concatenare questo inizio con tutte le possibili combinazioni di n lettere.

La classe `AllStringGenerator` genera tutte le stringhe su un dato alfabeto. La classe `StringGenerator` genera tutte le stringhe di una data lunghezza.

Le parole di lunghezza minore o uguale di j (soglia che deve essere scelta opportunamente) vengono controllate in sequenziale all'inizio del programma.

Ciascun thread lavora su un insieme di parole e si sincronizza all'inizio del lavoro con la classe `Master` per ottenere l'insieme di password non ancora trovate. Abbiamo scelto questa implementazione per evitare di dover mettere a disposizione dei thread un insieme di password sincronizzato (con lock di scrittura e di lettura), che avrebbe comportato l'acquisizione e il rilascio di un lock a ogni parola controllata. Il draw-back di questa scelta è che i thread non termineranno il lavoro nel momento in cui l'ultima password viene trovata, ma se l'insieme di password su cui i thread lavorano è di dimensione ridotta l'attesa diviene trascurabile.

La struttura delle classi è rappresentata in figura 3.

Abbiamo scelto di parametrizzare, oltre al numero di thread, anche la dimensione dei blocchi di lavoro. Cerchiamo di illustrare il motivo di questa scelta.

La dimensione dei blocchi di lavoro determina il numero di operazioni sincrone (e quindi sequenziali) che devono essere fatte (in particolare, `getPassowrds` e `getNextWord`). Se n è la lunghezza della password e j la dimensione del blocco di lavoro, la percentuale di operazioni sincrone dipenderà da $\frac{1}{36^j}$.

Come detto poco sopra la dimensione dei blocchi di lavoro comporta una "coda" di lavoro inutile dopo che è stata trovata l'ultima password. L'impatto di questo lavoro inutile sull'esecuzione del programma dipende dal rapporto fra j e n (e non dal numero di processori).

Abbiamo quindi due fattori che limitano lo speedup del programma: per valori di j vicini ad n la "coda" dell'esecuzione avrà una durata paragonabile alla durata del resto del programma; se invece il rapporto fra j ed n è trascurabile, lo speedup (secondo Amdahl) è limitato dalla percentuale di operazioni sincrone, e dipende quindi da 36^j.

È difficile quantificare precisamente il peso della coda e delle operazioni sincrone, ma quello che abbiamo cercato di illustrare è che la dimensione dei blocchi di lavoro deve essere una qualche funzione

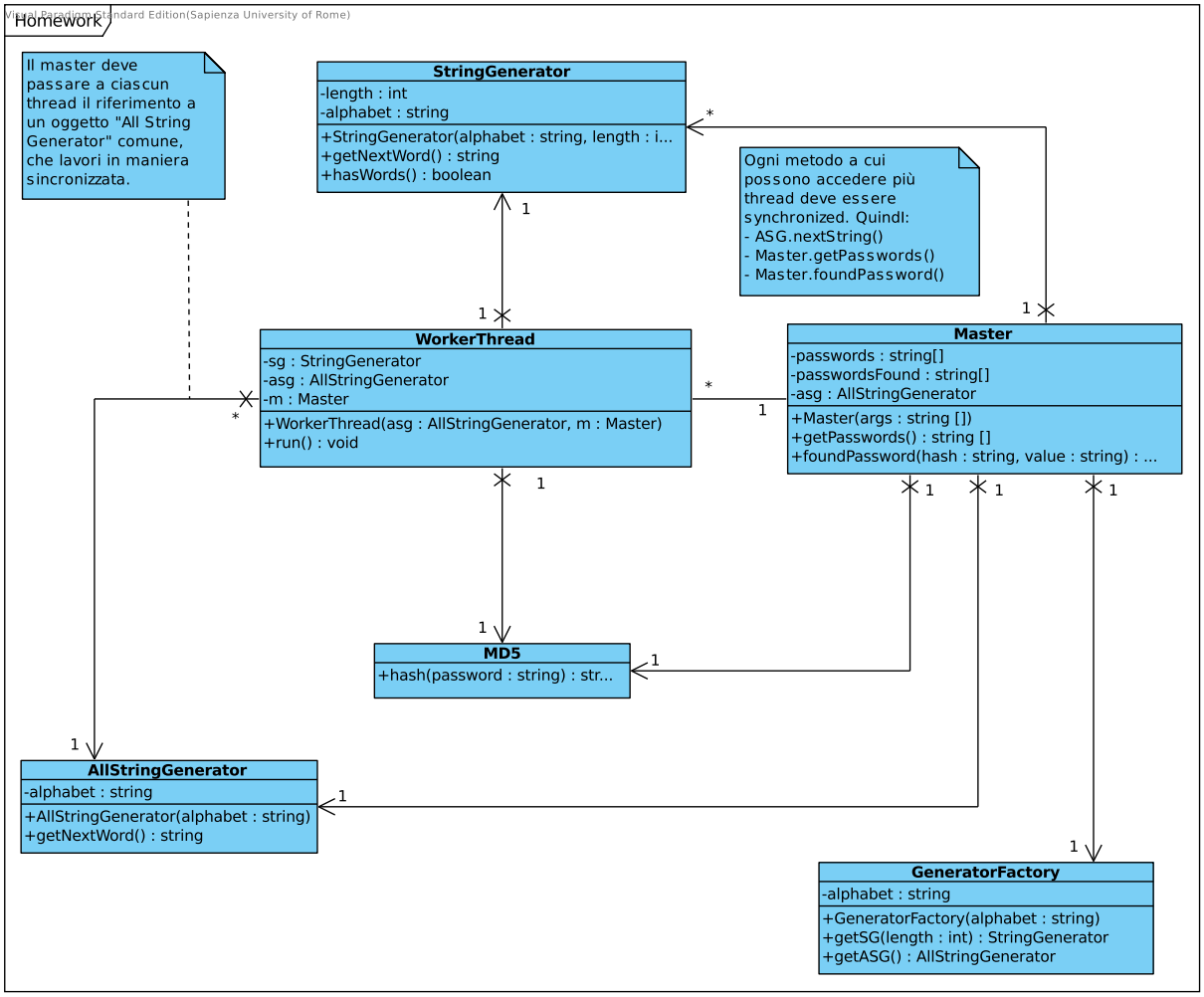

Figura 3: Struttura delle classi del progetto.

del numero di processori e della (supposta) lunghezza della password. Nei test che abbiamo condotto questo non è visibile, poiché il numero di processori e la lunghezza delle password sono molto bassi.

2.2 Piattaforma dei test

I computer su cui abbiamo testato l'algoritmo sono:

- un Raspberry Pi 2 Model B, con CPU "900 MHz quad-core ARM Cortex-A7" secondo wikipedia [2];
- un computer fisso con processore "Intel Core i5 (650) 3.2 GhZ dual-core", con hyperthreading (e possibilità di disattivarlo);
- un iMac 27″ "11,1" con processore "Intel Core i7 (860) 2.8 GHz quad-core", con hyperthreading.

Nei test condotti abbiamo fissato un tetto alla lunghezza della password e generato password casuali fino a questa lunghezza, assieme all'ultima password in ordine quasi lessicografico di quella lunghezza. Con lunghezza 4, il programma deve cercare n password di lunghezza minore o uguale a 4 e l'hash della password "9999". Evitiamo quindi casi limite in cui le n password generate sono tutte "vicine" e con un basso indice quasi lessicografico, costringendo il programma a provare (almeno) tutte le parole da "a" a "9999".

La lunghezza delle password è stata scelta da computer a computer, per arrivare a test della durata di qualche ora.

Nel calcolo dello speedup abbiamo considerato come tempo sequenziale il tempo di esecuzione dell'algoritmo con un solo thread.

2.2.1 Risultati attesi

Ci aspettiamo di vedere:

- speedup lineare;
- speedup massimo quando il numero di thread è uguale al numero dei processori;
- buon load balancing.

I primi due risultati saranno evidenti dai tempi di esecuzione.

Il terzo riteniamo di poterlo dedurre da un eventuale speedup massimo e lineare con pochi thread: significherebbe che il lavoro è ben distribuito anche fra questi pochi thread, che riescono a sfruttare al meglio i processori a disposizione. Inoltre l'algoritmo assegna blocchi di lavoro di uguale dimensione a ogni thread, e poiché il numero di questi blocchi (in una situazione reale) è molto maggiore del numero di processori (36^{n-j} blocchi contro p processori) ci aspettiamo che ogni thread lavori su circa $\frac{36^{n-j}}{p}$ blocchi di lavoro.

2.2.2 Risultati ottenuti

I risultati sul Raspberry Pi sono differenti con le due JVM utilizzate.

- Con la JVM OpenJDK i tempi sono piuttosto lunghi, ma lo speedup è praticamente lineare.

  Oltre i 4 thread l'algoritmo ha crashato circa un quinto delle volte per un "unhandled signal 11". Un rapida ricerca su internet mostra che il segnale indica un segmentation fault. Stupiti, seguendo il consiglio di [1], cambiamo JVM.

- Con la JVM Oracle i tempi sono sensibilmente più corti, ma lo speedup si arresta intorno a 3. I grafici di utilizzo della CPU mostrano che non viene utilizzato più del 75% dei 4 processori, il che spiega la limitazione dello speedup. Ipotizziamo che lo scarso utilizzo dei processori sia dovuto al garbage collector della JVM Oracle. Informazioni trovate su internet indicano che l'azione del garbage collector è una parte forzatamente sequenziale del codice, poiché ferma l'esecuzione di ogni thread. La prima implementazione dell'algoritmo usa un gran numero di oggetti immutabili (fra `String` e `BigInteger`), facendo lavorare molto il garbage collector.

Con l'i5 abbiamo avuto la possibilità di testare l'algoritmo con e senza hyperthreading. Con hyperthreading abilitato e aumentando il numero di thread lo speedup migliora leggermente, e supera il numero di core fisici, ma resta molto inferire al numero di core logici. Ipotizziamo che l'hyperthreading porti a risultati migliori, anche se non ne conosciamo il funzionamento.

Con l'iMac i risultati sono tragici: usando 3 thread si raggiunge già lo speedup massimo, molto inferiore allo speedup lineare. La causa potrebbe essere sempre il troppo lavoro del garbage collector.

È evidente che la prima implementazione è fortemente ottimizzabile: lo speedup non è assolutamente lineare come ci saremmo aspettati. La seconda implementazione, con le ottimizzazioni descritte nel seguito, ci aspettiamo abbia un comportamento più vicino a quello dell'algoritmo ideale.

2.3 Ottimizzazione del'implementazione

Le ottimizzazioni che abbiamo implementato sono le seguenti:

- rappresentiamo le parole con `byte[]`, invece di `String`. I vantaggi sono diversi: l'implementazione di md5 prende in input e dà in output `byte[]`, il che ci risparmia almeno due conversioni di tipo; non creiamo $O(36^n)$ nuove stringhe (che sono tipi immutabili), ma possiamo riutilizzare lo stesso `byte[]`.
- vale un discorso simile sostituendo `int` a `BigInteger`: passiamo da un tipo immutabile a un tipo mutabile, per la gioia del garbage collector.
- la struttura dati che accoglie le password è un `TreeSet<byte[]>`, con associato un `Comparator<byte[],byte[]>`, che ci permette di effettuare operazioni di ricerca e rimozione in $\log(n)$ (dove n è il numero di password). Per n sufficientemente grandi è un vantaggio rispetto a una ricerca lineare su una struttura non ordinata: per ogni parola bisognerebbe controllare tutti gli n hash, dato che la probabilità di trovare una password è molto bassa.
- l'insieme di password, mantenuto dal `Master`, non viene copiato da `getPasswords`: il metodo ritorna (senza sincronizzazione) il riferimento all'insieme del master. Nel momento in cui viene trovata una password, il master si occupa (con sincronizzazione) di duplicare l'insieme di password, rimuovere la password trovata, e di cambiare il proprio riferimento.

L'implementazione del metodo `foundPassword` non lascia l'insieme di password in uno stato intermedio e non corretto: per questo siamo portati a concludere che il metodo `getPasswords` non ritornerà mai un insieme di password in uno stato inconsistente.

2.3.1 Risultati ottenuti

I tempi di esecuzione sono diminuiti di un fattore compreso fra 4 e 8.

Con il Raspberry Pi lo speedup ora è poco sotto il lineare, anche con la JVM Oracle. Non si sono verificati altri segmentation fault.

Con l'i5, senza hyperthreading, i tempi sono più corti rispetto alla prima implementazione, ma lo speedup è praticamente identico. Con hyperthreading attivo lo speedup massimo che otteniamo è maggiore rispetto alla prima implementazione: si aggira intorno a 2.9, contro il 2.2 che avevamo in precedenza. Il risultato ci sembra buono, considerato che i core fisici sono solo 2, anche se non è lineare nel numero di core logici.

Con l'iMac, gli effetti dell'hyperthreading sembrano essere positivi con la seconda implementazione: lo speedup è vicino al lineare (nel numero di core fisici), e supera il numero di core fisici arrivando a circa 5. L'andamento non è comunque regolare.

3 Conclusioni

Dai risultati nascono alcune considerazioni:

- con parole troppo corte lo speedup non è visibile. Ipotizziamo sia dovuto al rapporto fra parti sequenziali dell'algoritmo (principalmente inizializzazione e sincronizzazione) e parti parallele, che per password corte è più alto.
- lo speedup non è *esattamente* lineare nel numero di processori. In accordo con la legge di Gustafson le parti sequenziali dell'algoritmo "abbassano" la pendenza della retta dello speedup, rendendolo quindi meno che lineare. Riteniamo che, per il ragionamento descritto nella sezione sull'implementazione dell'algoritmo, aumentando nel modo opportuno il numero di processori, la lunghezza della password e la dimensione dei blocchi di lavoro lo speedup si avvicinerà ulteriormente allo speedup lineare.
- i grafici mostrano che la nostra previsione non era infondata: abbiamo ottenuto speedup massimo quando il numero di thread è uguale al numero di core (fisici senza hyperthreading, logici con hyperthreading).

Le considerazioni sono molto parziali: l'hyperthreading complica l'interpretazione dei risultati, e il numero di core fisici dei computer su cui abbiamo eseguito i test è basso.

È facile pensare un'implementazione distribuita dell'algoritmo, dopo le dovute considerazioni. L'algoritmo non ha problemi di fiducia: nel momento in cui assegna un inseme di parole a un thread, assume che questo le controlli tutte. Se eseguito su più computer l'algoritmo deve tenere traccia degli insiemi di parole che ha distribuito per assegnare a un nuovo *worker* insiemi che restano "orfani" in caso di fallimenti.

Ci sono poi alcune ottimizzazioni che non abbiamo considerato, preferendo un'implementazione semplice che ci permettesse di concentrarci sull'analisi dell'algoritmo. Una su tutte è quella di interrompere i thread nel momento in cui tutte le password sono state trovate, eliminando il problema della "coda" di esecuzione.

3.1 Esecuzione dell'algoritmo

Abbiamo creato un `Makefile` per la compilazione il programma. Un modo possibile per eseguirlo è `java -cp ./bin homework_seconda_implementazione.Main [nome_file].txt`.

Visti i risultati ottenuti, il `Main` indicato sopra esegue l'algoritmo con un numero di thread pari al numero di processori a disposizione, e con la dimensione dei blocchi di lavoro fissata a 2, un numero magico che sembra comportarsi bene nei test condotti.

Riferimenti

[1] *[Java7] internal error.* 2013. URL: `http://archlinuxarm.org/forum/viewtopic.php?p=29876&sid=4f7640dedc4c7652bec6532798b87453#p29876`.

[2] *Raspberry Pi - Wikipedia.* URL: `https://en.wikipedia.org/wiki/Raspberry_Pi#Specifications`.