ASSIGNMENT 3

-

MD5 Decrypter

Forti Daniel 20157 (daforti@unibz.it) | Larcher Matteo 21183 (malarcher@unibz.it)

L’obbiettivo di questo esercizio era di sviluppare un sistema che, distribuendo il compito su più macchine (raspberry pi) usando i metodi da noi applicati nei precedenti assignment, è in grado di decifrare una serie di numeri casuali di cui conosciamo solamente l’hash.

Tattica:

la tattica generale utilizzata dal nostro gruppo consiste nella generazione degli hash di tutti i numeri partendo da 0 fino ad arrivare alla problemsize, confrontandoli prograssivamente con il hash corrente da trovare e salvandoli subito dopo. Una volta ricevuto un nuovo hash da trovare effettuiamo una ricerca sui dati salvati in precedenza per vedere se lo abbiamo già, e in caso contrario si continua con il ciclo citato precedentemente. Per una questione che verrà trattata tra poco, gli hash vengono mantenuti come stringhe e non come array di byte.

Tattica di salvataggio dei progressi:

l’approccio che utilizzava una HashMap si è rivelato troppo dispendioso di memoria e quindi abbiamo cambiato strategia. Utilizziamo come struttura dati un albero binario di ricerca, che utilizza come chiavi per i nodi la somma delle cifre del hash da trovare. In ogni nodo salviamo una ArrayList di interi, contenente tutti i numeri i cui hash hanno somma dei caratteri uguale alla chiave di quel nodo. La scelta di tenere gli hash come stringhe è stata fatta perché la quantità di somme uniche dei caratteri è di gran lunga maggiore rispetto a quella della quantità di somme uniche dei byte: per i primi 5000000 di hash ci sono 2500 circa somme uniche dei byte, mentre 500000 somme uniche di stringhe. Riuscire a dividere il dominio in 500000 nodi è più dispendioso di memoria ma rende la parte di ricerca molto più veloce, ed essendo che noi salviamo solo array di numeri e non gli hash interi, non abbiamo grandi problemi di memoria. Un altro contro è che durante la ricerca tra gli elementi dell’array contenuti in un nodo, dobbiamo effettuare il hashing di ogni elemento dell’array di nuovo, ma essendo che il dominio è così piccolo (grazie alla scelta del criterio di divisione dei nodi) che questo tempo diventa poco influente.

Tattica di divisione dei compiti:

le classi che utilizziamo sono Master e Slave. Entrambe svolgono il lavoro di base di fare l’hash dei numeri e cercare l’hash, ma il master in più si occupa di gestire gli Slave. La divisione dei numeri di cui fare il hash si basa sul fatto che Master e Slave partano con un offset che dipende dal numero dello Slave e avanzano con un incremento uguale al numero totale di Slave più il Master: se il Master parte dallo 0, il primo Slave parte dal 1, il secondo dal 2 e tutti avanzano di 3 in 3. Una volta che viene aggiunto un nuovo Slave, il Master ferma l’esecuzione propria e tutti gli Slave, calcola il nuovo punto di partenza per tutti prendendo il numero più alto tra le variabili current degli Slave e la propria, e da quella tutti si posizioneranno con un offset uguale a quello che avevano in partenza. Continuando con l’esempio di prima, se il current più alto è 4 e appartiene al primo Slave, il Master arriverà a current 6, mentre il secondo Slave a 5, dopodiché il nuovo Slave arrivato verrà posizionato a current 7, e da lì in poi tutti avanzeranno di 4 in 4. Con questo sistema di divisione ci accertiamo che tutti lavorino su hash della stessa dimensione così da non avere uno Slave/Master in idle visto che ha già terminato i propri numeri in quanto più veloci da elaborare rispetto agli altri.

Comunicazione tra Master e Slave:

la comunicazione avviene tramite RMI, in quanto si è rivelato il metodo più comodo per gestire il tutto. Onde evitare problemi di accesso e/o modifica concorrente ad oggetti, e anche per evitare di bloccare un thread su una chiamata remota, le nostre classi hanno un metodo lifecycle che esegue un compito diverso in base a dei flag. Le chiamate remote impostano solo quindi delle variabili che servono ad eseguire un compito e impostano a true il flag corrispondente all’azione (nello specifico abbiamo solo flag di aggiornamento e dell’arrivo di un nuovo problema da risolvere). Così ogni thread esegue le proprie azioni più dispendiose di tempo, ed esseno eseguite da un solo thread, non dobbiamo preoccuparci di problemi di concorrenza. Per assicurarci che non vengano effettuate comunque delle chiamate a metodi in situazioni sconvenienti, abbiamo implementato un semplice sistema di comunicazione tra Master e Slave che consiste nei flag update e waiting, così che il Master possa conoscere lo status degli Slave per poter eseguire o meno certe chiamate.

Questo è quanto per ciò che riguarda le strategie ad alto livello, nel codice sono presenti commenti che indicano nel dettaglio il funzionamento delle varie parti del codice