PC-2022/23 Bloom Filter

Jacopo Manetti
E-mail address
jacopo.manetti1@stud.unifi.it

Abstract

In questo progetto abbiamo confrontato le prestazioni dell'algoritmo bloom filter in versione parallela con quella sequenziale, tutti i risultati sono stati ottenuti su un computer dotato di un processore Intel i7 11800h a 8 core (16 logici). Utilizzando python e la libreria Joblib, abbiamo sviluppato entrambe le implementazioni e abbiamo utilizzato tecniche di profilazione per valutare le prestazioni. I risultati hanno dimostrato che l'utilizzo della versione parallela dell'algoritmo ha permesso di ottenere uno speed-up medio di circa 5 rispetto alla versione sequenziale, dimostrando l'efficienza dell'implementazione parallela su questo specifico hardware. Inoltre, questo progetto ha fornito una panoramica delle tecniche di programmazione parallela e dei loro vantaggi nell'ottimizzare i tempi di esecuzione di un algoritmo. Dato che uno degli utilizzi del bloom filter è l'individuazione di url maligni i test utilizzano liste di url, tuttavia il codice scritto è corretto per qualsiasi altra applicazione.

1. L'algoritmo

Il Bloom Filter è un algoritmo probabilistico che consente di determinare se un elemento è già stato inserito in un insieme o meno. Utilizza una matrice di bit di dimensione fissa, che inizialmente è impostata su zero. Quando un elemento viene inserito nell'insieme, vengono calcolati diversi hash dell'elemento e i bit corrispondenti nella matrice vengono impostati su 1. Quando si verifica se un elemento è presente nell'insieme, vengono calcolati di nuovo i suoi hash e se tutti i bit corrispondenti sono impostati su 1, l'elemento viene considerato presente. Tuttavia, è possibile che l'elemento non sia presente ma i suoi hash corrispondano a bit impostati su 1 a causa di collisioni, il che fa sì che il Bloom Filter commetta un falso positivo. Il tasso di falsi positivi può essere controllato modificando la dimensione della matrice e il numero di hash utilizzati.

1.1. Il codice sequenziale

La classe BloomFilter rappresenta il filtro di bloom e definisce i seguenti metodi:

- **__init__**: inizializza una struttura di dati bit_array di dimensione "size" e imposta il numero di hash da utilizzare per l'operazione di hashing su "hash_count".
- **initialize**: riceve una lista di URL e utilizza la funzione di hash "hash_calculate" per ottenere l'indice in bit_array per ogni URL. Questo indice viene impostato a 1 per indicare che l'URL è stato visto.
- **check**: riceve un URL e utilizza la funzione di hash "hash_calculate" per ottenere l'indice in bit_array. Se tutti i valori in bit_array corrispondenti a questi indici sono 1, l'URL viene considerato come già visto e viene restituito True. In caso contrario, viene restituito False.

Il codice utilizza la libreria hashlib per creare hash sha256 e utilizza la funzione **hash_calculate** per calcolare l'indice in bit_array per ogni URL, questa funzione riceve in input oltre alla stringa a cui applicare l'hash anche 2 valori interi, il primo n serve per applicare il modulo al valore hash restituito in modo che il valore rientri nella lunghezza dell'array di bit, il secondo y serve a cambiare indice della funzione hash in modo che a ogni iterazione su j la funzione hash applicata restituisca un valore diverso, in questo modo per ogni stringa si possono applicare più funzioni hash diverse.

Il codice testa il funzionamento del filtro creando una lista di URL maligni e una lista di URL da testare. Il tempo di inserimento degli URL maligni nel filtro di bloom viene misurato e viene eseguito un controllo sulla lista di URL da testare per determinare se sono URL maligni o no. Il tempo di esecuzione per la verifica degli URL viene misurato e mostrato all'utente.

1.2. Il codice parallelo

Per parallelizzare il codice sono state individuate 2 parti adatte, la prima è nel metodo **initialize**, non è possibile parallelizzare direttamente questo metodo perchè

scrivendo direttamente sull'array di bit i valori ottenuti da "hash_calculate" in parallelo avremmo un problema con la zona condivisa, tuttavia è possibile dividere il compiti usando un array d'appoggio, viene quindi eseguito in parallelo il calcolo della funzione hash su tutti gli URL e salvati nell'array result, per ognuno degli hash che devono essere calcolati, il codice esegue un ciclo che utilizza la funzione **delayed** per eseguire il calcolo dell'hash per ogni URL nella lista. La funzione **delayed** consente di eseguire il calcolo dell'hash in modo asincrono, ovvero il codice continua ad eseguire i cicli successivi senza attendere il completamento del calcolo dell'hash. e infine l'array di bit viene impostato a 1 usando il contenuto di result, inoltre l'intero calcolo parallelo si trova all'interno del context manager API della classe Parallel per evitare di dover generare e distruggere i thread ogni volta che una nuova funzione di hash viene applicata. La seconda parte che può essere parallelizzata è il check degli elementi, per farlo viene creato un nuovo metodo **ParallelCheck** che semplicemente applica il metodo check a tutti gli elementi della lista di URL in parallelo per ogni funzione hash.

2. Analisi della correttezza

Per controllare il corretto funzionamento dell'algoritmo in entrambe le versioni sono state create 2 liste di url, url_maligni con cui inizializzare il filtro e url_test con contiene degli url maligni della prima lista (nelle prime 7 posizioni) e altri url generici, viene quindi creato un filtro inizializzato con la lista degli url maligni e viene poi stampato il check usando la seconda lista, il risultato ottenuto segnalerà quindi come maligni (true) solo i primi 7 elementi.

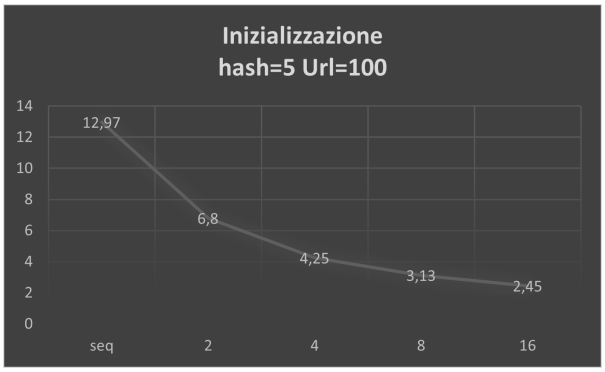
3. Analisi delle prestazioni

Di seguito vengono riportati i risultati degli speed-up ottenuti dall'esecuzione del codice in versione sequenziale e parallela al variare del numero dei thread, data la natura del problema i parametri di confronto rilevanti sono il variare del numero delle funzioni hash e del numero di url, per confrontare quest'ultimo cambiamento è stata definita una funzione per generare casualmente una lista di stringhe, dato che viene usato sempre lo stesso seed le parole generate sono sempre le stesse. Dato che al fine del controllo delle prestazioni la dimensione del filtro non influisce viene usato un filtro di dimensione 1000. Per ognuna delle esecuzioni viene misurato il tempo impiegato per l'inizializzazione e quello per controllare gli url, infine viene riportato il tempo totale di esecuzione.

3.1. Inizializzazione

Qui vengono mostrati i risultati di diverse esecuzioni dell'inizializzazione

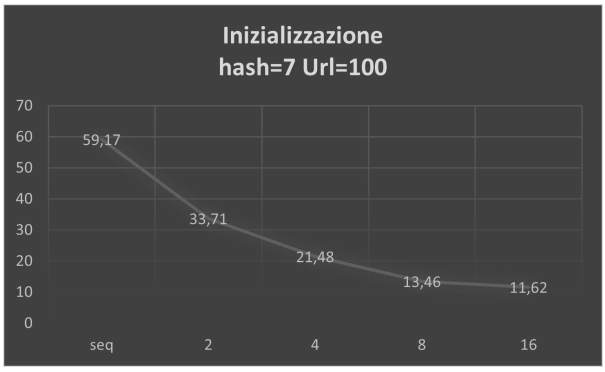
3.1.1 Esecuzione con 5 funzioni hash e 100 Url



Le prestazioni descrivono l'inizializzazione di un hash con un fattore di carico pari a 5 e una dimensione iniziale di 100. Il tempo impiegato per l'inizializzazione viene misurato in secondi per un diverso numero di thread in esecuzione contemporaneamente: sequenziale, 2, 4, 8 e 16 thread.

I risultati mostrano che all'aumentare del numero di thread, il tempo di inizializzazione diminuisce notevolmente, passando dai 12.97 secondi della modalità sequenziale ai 2.45 secondi con 16 thread. Lo speed up calcolato è di circa 5.29, il che indica una buona scalabilità con l'aumentare del numero di thread.

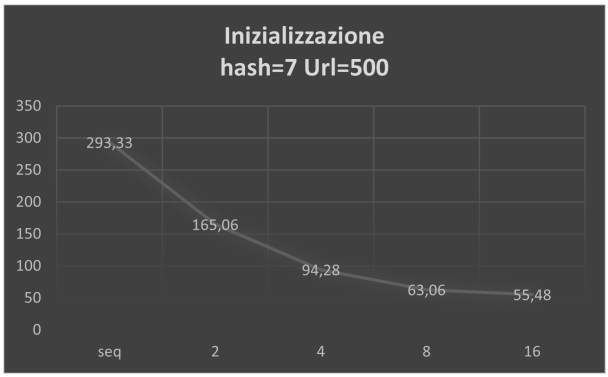
3.1.2 Esecuzione con 7 funzioni hash e 100 Url



Qui vengono mostrati i dati che si riferiscono all'inizializzazione con un hash di valore 7 e un numero di url di 100.

Si può vedere che all'aumentare del numero di thread utilizzati, il tempo necessario per l'inizializzazione diminuisce. Il speedup massimo è di 5,09 volte rispetto all'esecuzione in sequenza.

3.1.3 Esecuzione con 7 funzioni hash e 500 Url

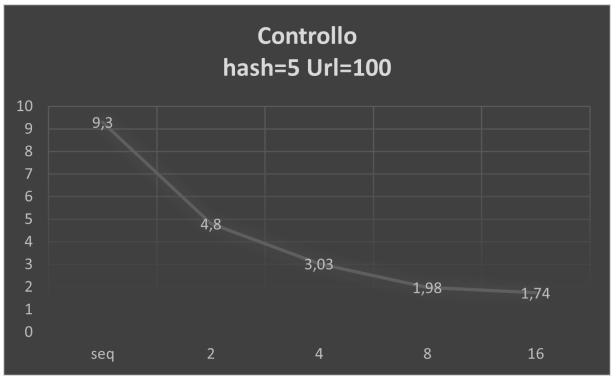


Questa tabella riporta le prestazioni di un'operazione di inizializzazione su una determinata funzione hash, con un fattore di hash pari a 7 e una dimensione degli url di 500. Si osserva che anche in questo caso abbiamo un miglioramento delle prestazioni all'aumentare del numero di thread, con un speedup di 5,28 quando si passa dalla versione sequenziale a quella con 16 thread.

3.2. Confronto

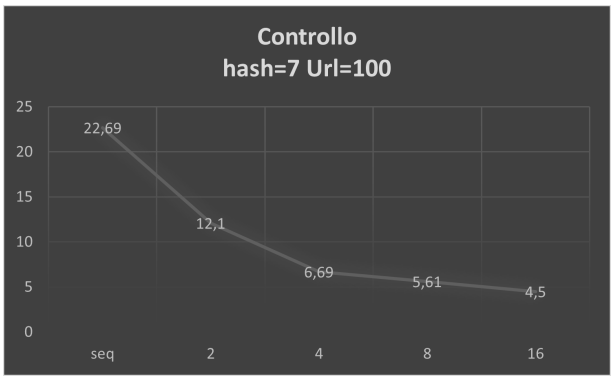
Qui vengono mostrati i risultati di diverse esecuzioni della funzione di controllo

3.2.1 Esecuzione con 5 funzioni hash e 100 Url



Questa tabella riporta i tempi di esecuzione della funzione "parallel check" per diverse configurazioni. In particolare, la funzione è stata eseguita su un hash di dimensione 5 e url 100 utilizzando diversi numeri di thread. La colonna "seq" indica il tempo di esecuzione sequenziale, mentre le altre colonne indicano i tempi di esecuzione con 2, 4, 8 e 16 thread. In questo caso, il miglioramento di prestazioni è di circa 5,3 volte.

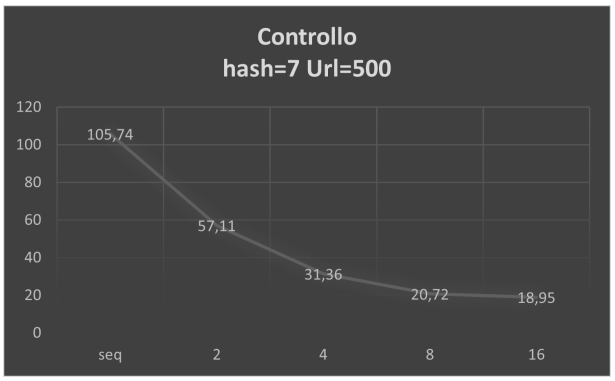
3.2.2 Esecuzione con 7 funzioni hash e 100 Url



In particolare, la tabella si riferisce al caso in cui hash=7 e url=100.

Per eseguire la funzione in sequenza (con un solo thread), sono necessari circa 22,69 secondi. Con l'aumento del numero di thread, i tempi di esecuzione diminuiscono, fino a raggiungere circa 4,5 secondi con 16 thread. Il speed up massimo ottenuto è di circa 5,04 volte.

3.2.3 Esecuzione con 7 funzioni hash e 500 Url

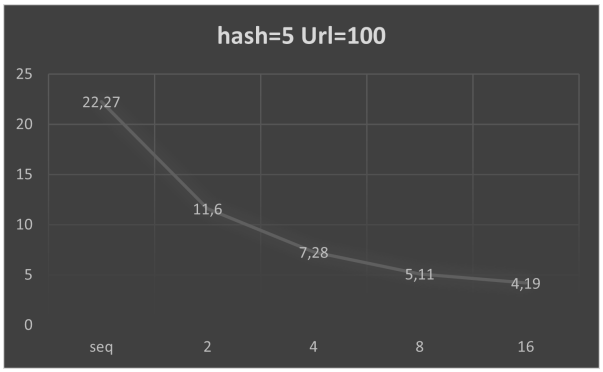


Questa tabella riporta i risultati dell'esecuzione della funzione "parallel_check" con hash=7 e url=500, utilizzando diversi numeri di thread. Il tempo di esecuzione viene misurato in secondi. Come per le altre tabelle, si riporta il tempo di esecuzione per il caso sequenziale (seq) e per i casi in cui vengono utilizzati 2, 4, 8 o 16 thread. Viene inoltre calcolato lo speedup, ovvero il rapporto tra il tempo di esecuzione sequenziale e il tempo di esecuzione con il numero di thread considerato. I risultati mostrano che l'utilizzo di un numero maggiore di thread porta ad uno speedup positivo, con un valore massimo di circa 5,58 con 16 thread.

3.3. Confronto totale

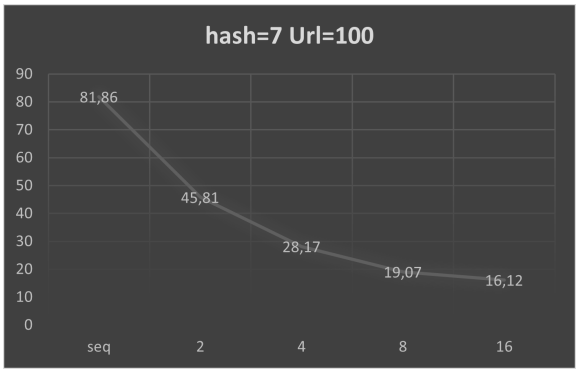
Qui vengono mostrati i risultati dell'esecuzione totale, visto che il miglioramento è paragonabile in entrambe le funzioni anche il risultato finale è simile mantentendo un speed up di circa 5.

3.3.1 Esecuzione con 5 funzioni hash e 100 Url



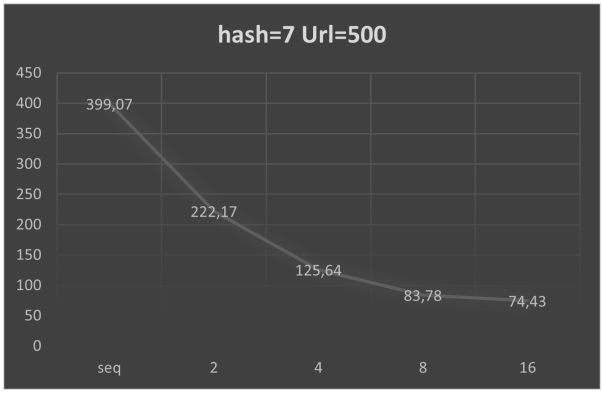
L'esecuzione completa con 5 funzioni hash e 100 url ha uno speed up di 5,32

3.3.2 Esecuzione con 7 funzioni hash e 100 Url



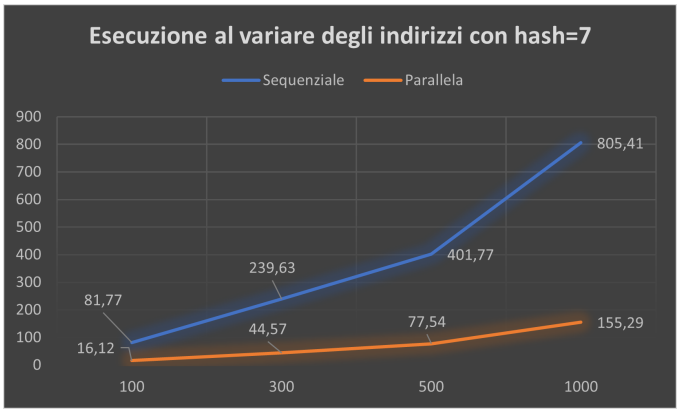
L'esecuzione completa con 7 funzioni hash e 100 url ha uno speed up di 5,08

3.3.3 Esecuzione con 7 funzioni hash e 500 Url



L'esecuzione completa con 7 funzioni hash e 500 url ha uno speed up di 5,36

3.3.4 Esecuzione al variare degli indirizzi con 7 funzioni hash



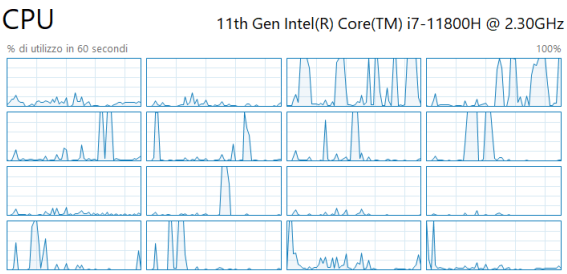
Qui vengono confrontati i diversi tempi di esecuzione al variare del numero di url (100,300,500,1000) con 7 funzioni hash, anche qui si puo notare che il valore degli speed up è di circa 5.

4. Conclusione

In conclusione, la comparazione tra l'implementazione parallela e quella sequenziale del Bloom Filter ha dimostrato che l'utilizzo della versione parallela ha permesso di ottenere un significativo speed-up, il quale si è attestato ad una media di circa 5 volte rispetto alla versione sequenziale. Grazie alla creazione di diverse funzioni hash e alla parallellizzazione delle operazioni svolte, l'efficienza dell'implementazione parallela è stata dimostrata in un caso specifico su un hardware specifico. Inoltre, questo progetto ha offerto un'approfondita panoramica sulle tecniche di programmazione parallela e sui loro vantaggi per quanto riguarda l'ottimizzazione dei tempi di esecuzione degli algoritmi. Il filtro di Bloom risulta uno strumento utile e flessibile per la risoluzione di molti problemi in vari campi applicativi, tra cui la sicurezza informatica.

Di seguito le prestazioni della cpu durante l'esecuzione sequenziale e parallela.

4.1. Esecuzione sequenziale CPU



4.2. Esecuzione parallela CPU