Elaborato Programmazione Parallela

Anno Accademico 2022/23
Alberto Biliotti
Matricola: 7109894
alberto.biliotti@stud.unifi.it

Abstract

Per svolgere la seconda parte del progetto del corso di Parallel Computing è stato deciso di sviluppare una versione parallela del filtro di Bloom sfruttando la libreria Joblib con il linguaggio Python aggiornato alla versione 3.11.

Descrizione dell'ambiente di lavoro utilizzato

Per sviluppare questo progetto è stato utilizzato come ambiente di lavoro interno (IDE) PyCharm aggiornato alla versione 2022.1.3 con sistema operativo Windows 11 e con una CPU Intel di undicesima generazione i7-11800H con frequenza di 2.30GHz, 8 core fisici e 16 virtuali e con 16 GB di memoria RAM disponibile.

1. Caratteristiche implementative sequenziali

Come accennato in precedenza, nel progetto è stato implementato un filtro di Bloom, ossia una struttura dati che permette in generale di stabilire se un dato elemento appartenga o meno ad un dato insieme. Nel caso in cui si abbia responso negativo, saremo certi che tale elemento non appartiene all'insieme, se invece il responso è positivo non possiamo essere certi che l'elemento appartenga effettivamente all'insieme, in quanto il filtro di Bloom può accettare anche elementi non appartenenti all'insieme. In altre parole il filtro di Bloom può essere soggetto a falsi positivi ma non a falsi negativi.
Il filtro non è altro che un vettore di bit in cui, al momento della creazione, tutti i bit vengono posti a 0. Devono essere fornite inoltre un certo numero di funzioni hash che permettano di trasformare un elemento in un valore numerico compreso tra zero e la lunghezza del nostro filtro. Per inizializzare la nostra struttura, per ogni valore dell'insieme di cui si vuole testare l'appartenenza o meno di un elemento, viene calcolato il valore restituito dalle funzione citate prima. Successivamente ad ogni posizione del filtro individuata da tali valori viene assegnato valore uno. Quando vogliamo andare a controllare se un elemento appartiene o meno all'insieme andremo ad applicare le funzioni di hash all'elemento. Se tutte le posizioni del filtro così individuate contengono valore uno accetterò l'elemento come appartenente all'insieme, altrimenti sarò certo che l'elemento non appartiene all'insieme

1.1. Formato degli elementi

Nella mia implementazione ho deciso di usare come elementi le stringhe, ed in particolare nei miei esempi ho usato stringhe contenenti indirizzi e-mail, in quanto una delle possibili applicazioni del filtro di Bloom è la rilevazione della posta indesiderata. Per fare ciò ho creato una lista di indirizzi sicuri che userò per inizializzare il filtro, oltre a qualche altro indirizzo che andrò a testare in seguito.

1.2. Funzioni Hash

Ho scelto di implementare ben sette funzioni hash per gli indirizzi, di cui una va semplicemente a sommare i valori ASCII della stringa e ad applicargli il modulo con la lunghezza del filtro, la seconda ripete tale la funzione vista prima per numero di volte pari alla lunghezza della stringa alla quarta, la terza applica la prima funzione alla stringa a cui ho invertito lettere minuscole e maiuscole, la quarta sfrutta la funzione di criptazione "hash" per generare una nuova stringa che andrò a criptare con la prima funzione, la quinta sfrutta lo stesso meccanismo solo con la funzione di criptazione per le stringhe "sha256" della libreria "hashlib", la sesta invece, similmente alla seconda, applica la quinta funzione per numero di volte pari alla lunghezza della stringa in ingresso alla quarta ed infine la settima, che in realtà può essere sfruttata più volte in quanto sfrutta la funzione hash della libreria "mmh3" che restituisce un intero data una stringa ed un seme di generazione, che posso variare per poter ottenere più funzioni di hash senza dover implementare nuovi metodi.

1.3. Implementazione sequenziale

Per prima cosa è stato deciso di sfruttare la programmazione a oggetti per il nostro progetto, in particolare è stato deciso di definire il filtro di Bloom come una su-

perclasse, da cui erediteranno due sottoclassi contenenti l'implementazione sequenziale e parallela del filtro stesso. Tale superclasse conterrà, oltre al costruttore che, presi in input la dimensione desiderata del filtro ed il numero di funzioni da applicare, andrà a creare il filtro inizializzando tutti i valori a zero, anche un metodo per ricavare la probabilità di ottenere un falso positivo sfruttando la formula $p_{err} = (1 - e^{-\frac{r*m}{n}})^r$ dove r è il numero di funzioni hash, n la lunghezza del filtro e m il numero di elementi distinti con cui è stato inizializzato il filtro insieme anche al metodo per controllare se una stringa appartiene all'insieme. Inoltre è stato deciso di dividere il progetto su tre file differenti, il primo contenente la superclasse insieme alle funzioni hash definite in precedenza ed alla lista di indirizzi considerati sicuri, il secondo contenente la sottoclasse con i metodi di inizializzazione sequenziali del filtro, insieme ai vari test mentre il terzo conterrà la sottoclasse con i metodi paralleli. Nel primo file è stata inclusa anche una funzione "applicahash" che, in base all'indice passato in ingresso, applicherà una funzione di hash diversa alla stringa in esame (se l'indice è pari o superiore al sette, applicherà la funzione di hash della libreria "mmh3" con un seme di generazione pari all'indice passato in ingresso) in modo tale da poter iterare su tale metodo per un numero di volte pari al numero di funzioni hash desiderate nel filtro (questo metodo servirà anche per non dover distruggere tutti i thread dopo l'applicazione di una funzione nella versione parallela) L'implementazione sequenziale del metodo per inizializzare il filtro dato una lista di indirizzi consiste quindi nel calcolare prima i risultati dell'applicazione di ciascuna funzione hash a tutta la lista inserendo volta volta i valori calcolati in una lista sfruttando la list comprehension per poi, raggiunto il numero di funzioni hash desiderate, porre ad uno le posizioni del filtro corrispondenti ai valori calcolati. Per quanto riguarda invece l'implementazione del controllo dell'appartenenza di una stringa, implementato in maniera sequenziale per entrambe le versioni, è stato sufficiente controllare tramite un ciclo for se le posizioni risultanti dall'applicazione delle funzioni hash desiderate alla stringa in ingresso corrispondano ad uno e, in caso negativo, restituire falso, mentre al contrario se in nessuna posizione trovo uno zero restituisco vero.

2. Sviluppo del programma parallelo

2.1. Trovare gli hotspot dell'algoritmo

Per poter parallelizzare il codice visto in precedenza è necessario individuare le parti dell'algoritmo che richiedono più tempo e risorse per poter essere eseguite in modo tale da poter parallelizzare tali parti in maniera che il nostro programma parallelo ottenga effettivamente uno speed up rispetto alla versione sequenziale. Per quanto riguarda il Filtro di Bloom abbiamo due parti esose dal punto di vista computazionale che sono il controllo dell'appartenenza di una stringa all'insieme ma soprattutto l'inizializzazione del filtro con una lista (possibilmente lunga) di stringhe che costituiranno l'insieme di cui andremo a decidere l'appartenenza o meno delle stringhe. Entrambe queste operazioni sono imbarazzantemente parallele, infatti, per quanto riguarda il controllo dell'appartenenza di una stringa all'insieme, ogni funzione di hash è indipendente dall'altra (come vedremo però, il fatto che sia parallelizzabile non garantisce il fatto che ciò sia effettivamente conveniente), mentre per quanto riguarda l'inizializzazione posso applicare una stessa funzione in parallelo a più elementi della lista senza che sia necessario alcun tipo di sincronizzazione tra i vari processi in quanto ogni stringa dell'insieme è indipendente dall'altra e l'ordine di inizializzazione non conta, in quanto devo semplicemente porre ad uno una posizione del filtro che non è altro che un vettore di bit.

2.2. Implementazione parallela

Per parallelizzare il programma è stato quindi sufficiente creare una sottoclasse della superclasse astratta vista in precedenza e ridefinire sia il costruttore in modo tale da poter inserire il numero massimo di core utilizzabili come parametro in ingresso sia il metodo per l'inizializzazione del filtro con la lista di indirizzi sicuri. Tale metodo ridefinito, similmente alla versione sequenziale, considera una funzione di hash alla volta tramite un ciclo for che itera sull'indice del metodo che racchiude tutte le funzioni hash per un numero di volte pari al numero di funzioni hash desiderate. Tale funzione sarà però eseguita contemporaneamente su più elementi della lista in parallelo. Per fare ciò è stato sufficiente racchiudere, insieme al decoratore delayed per poter raccogliere i risultati, il metodo "applicahash" applicato ad ogni stringa della lista di indirizzi sicuri. I risultati, una volta terminata l'applicazione della funzione in parallelo, saranno inseriti in una lista contenente tutti i risultati di ogni funzione che, similmente all'implementazione sequenziale, verranno usati per porre ad uno le posizioni del filtro corrispondenti Inoltre, per evitare di dover generare e distruggere i thread ogni volta che devo applicare una nuova funzione di hash, è stato deciso di sfruttare il context manager API della classe Parallel "with ... as ..:" invece di usare la classe Parallel direttamente dentro il ciclo for. Per quanto riguarda invece il controllo dell'appartenenza di una stringa all'insieme, è stato scelto di mantenere l'implementazione sequenziale in quanto, pur essendo un problema imbarazzantemente parallelo, l'overhead che avrei nella generazione dei thread per poter applicare ogni funzione hash in parallelo su una singola stringa fa in modo che l'implementazione parallela risulti estremamente più lenta di quella sequenziale per una singola stringa, e siccome devo controllare una

sola stringa alla volta e non un insieme come avviene per l'inizializzazione del filtro, è stato scelto di mantenere la versione sequenziale.

3. Analisi delle prestazioni e speed up

Per confrontare le prestazioni dell'implementazione parallela rispetto a quella sequenziale è stata fornita ad entrambe le implementazioni una lista, definita nel file "BloomFilter.py" importato da entrambe i file contenenti le sottoclassi, come visto in precedenza, ed è stato registrato il tempo di esecuzione dell'inizializzazione di entrambi i filtri (parallelo e sequenziale) con tale lista. L'implementazione parallela è stata eseguita inoltre con un numero diverso di thread massimi (2,4,8,16) per osservare il comportamento del codice parallelo al variare del numero di thread massimi utilizzabili. Inoltre non ci siamo preoccupati della dimensione del filtro in quanto non influisce in alcun modo sul tempo di esecuzione ma solo sul rischio di falsi positivi durante il controllo di una stringa, perciò la dimensione è stata impostata a 800 posizioni. Infine, per registrare correttamente il tempo di esecuzione di entrambe le implementazioni, sono stati eseguiti più volte entrambi i programmi ed è stato scelto come tempo da registrare la media di tutti i tempi di esecuzione.

3.1. Filtro con 5 funzioni hash e 100 indirizzi

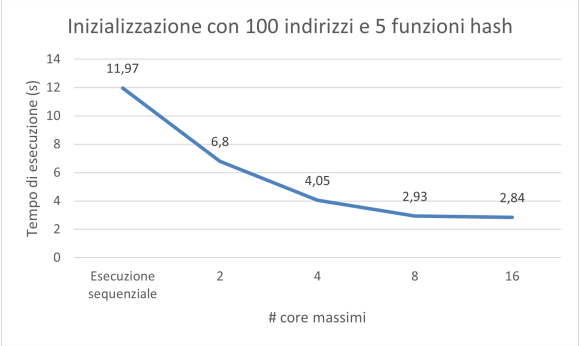

Figure 1. Tempi di esecuzione per l'Inizializzazione di un filtro con 100 indirizzi sicuri e 5 funzioni hash al variare del numero di core massimi

# thread massimi	Tempo esecuzione (s)
Implementazione sequenziale	11,97
2	6,8
4	4,05
8	2,93
16	2.84

Table 1. Tempi di esecuzione per l'Inizializzazione di un filtro con 100 indirizzi sicuri e 5 funzioni hash

Abbiamo deciso di iniziare a testare la nostra implementazione parallela inizializzando il filtro con 100 indirizzi sicuri distinti tra loro e 5 funzioni di hash, per controllare come si comporta il programma con poche funzioni e pochi elementi da inizializzare.
Come possiamo notare dal grafico, riportato nella figura 1, all'aumentare del numero massimo di core a disposizione il tempo di esecuzione diminuisce, anche se in maniera sempre minore fino ad avere una differenza quasi impercettibile tra l'esecuzione con 8 e 16 core. Ciò dimostra che non sempre aumentare il numero di core a disposizione renda il programma più efficiente. I risultati però dimostrano che, anche con poche funzioni hash e una mole ridotta di indirizzi da inizializzare, la nostra implementazione parallela risulti essere nettamente più vantaggiosa rispetto a quella sequenziale, tanto che il tempo impiegato dall'esecuzione parallela con 16 core a disposizione è circa un quarto di quello sequenziale.

3.2. Filtro con 7 funzioni hash e 100 indirizzi

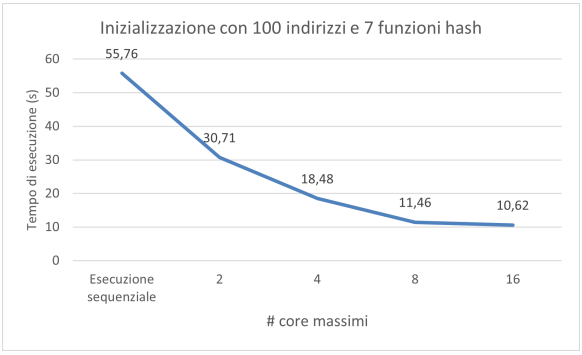

Figure 2. Tempi di esecuzione per l'Inizializzazione di un filtro con 100 indirizzi sicuri e 7 funzioni hash al variare del numero di core massimi

# thread massimi	Tempo esecuzione (s)
Implementazione sequenziale	55,76
2	30,71
4	18,48
8	11,46
16	10,62

Table 2. Tempi di esecuzione per l'Inizializzazione di un filtro con 100 indirizzi sicuri e 7 funzioni hash

Come secondo test abbiamo deciso di aumentare il numero di funzioni hash da considerare a 7, includendo così tutte le diverse funzioni implementate nel codice che sono state descritte in precedenza.
Possiamo osservare come i tempi di esecuzione siano nettamente maggiori rispetto alla precedente esecuzione. Questo

dipende dal fatto che la sesta funzione di hash, che ripete la funzione di hash che si basa sulla funzione di crittografia "sha256" per un numero di volte pari alla lunghezza della stringa alla quarta, richiede molto più tempo per essere completata rispetto alle altre funzioni. Nonostante ciò si nota come il grafico non si discosti troppo da quello mostrato nella sezione precedente, arrivando addirittura ad avere un tempo di esecuzione parallela con 16 core pari ad un quinto rispetto a quello del programma sequenziale, segno che l'efficacia della mia implementazione parallela non è condizionato in maniera negativa dal numero di funzioni hash da utilizzare per il filtro.

3.3. Filtro con 5 funzioni hash e 500 indirizzi

Come ultimo test è stato deciso di testare le implementazioni inizializzando un filtro di Bloom con una lista di ben 500 indirizzi sicuri e 7 funzioni di hash.

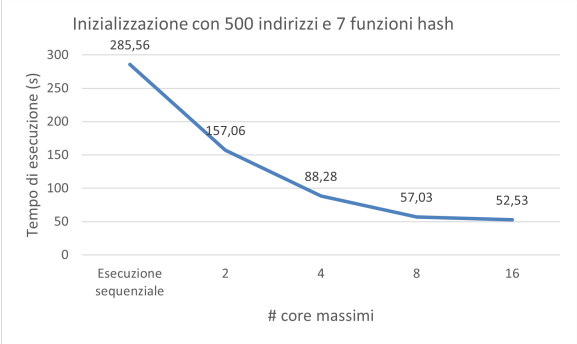

Figure 3. Tempi di esecuzione per l'inizializzazione di un filtro con 500 indirizzi sicuri e 7 funzioni hash al variare del numero di core massimi

# thread massimi	Tempo esecuzione (s)
Implementazione sequenziale	285,56
2	157,06
4	88,28
8	57,03
16	52,53

Table 3. Tempi di esecuzione per l'inizializzazione di un filtro con 500 indirizzi sicuri e 7 funzioni hash

Notiamo come, anche all'aumentare dell'ampiezza dell'insieme di inizializzazione del filtro, il grafico tenda ad assumere lo stesso andamento di quello visto in precedenza con 100 indirizzi, con l'esecuzione con 16 core che impiega sempre un quinto del tempo di quello necessario all'esecuzione sequenziale per completare l'inizializzazione. Ciò conferma ulteriormente la validità della mia implementazione parallela che mantiene sempre uno speed-up quasi costante anche al variare delle condizioni iniziali, come avremo modo di confermare anche nella sezione successiva

3.4. Confronto dello speed up

Riportiamo inoltre il grafico che mostra il valore dello speed up al variare del numero di thread massimi disponibili per le varie combinazione di numero di funzioni di hash e di ampiezza della lista delle stringhe su cui inizializzare il filtro, ottenuto dividendo per il tempo di esecuzione sequenziale i tempi di esecuzione parallela.

Notiamo come i grafici dei vari speed up (in particolare

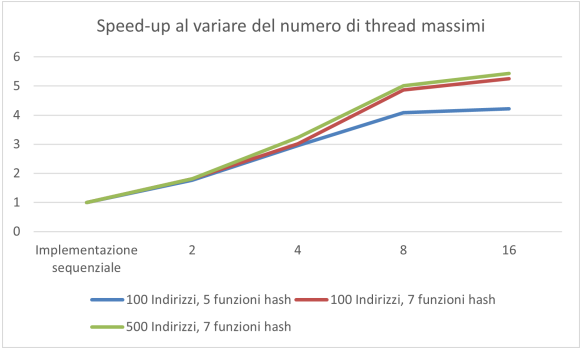

Figure 4. Speed up delle varie combinazioni di numero di funzioni hash e lunghezza della lista di stringhe di inizializzazione al variare del numero di core massimi

delle esecuzioni con 7 funzioni hash) risultino essere quasi sovrapposti tra loro, il che conferma definitivamente il fatto che la mia implementazione risulti soddisfacente anche per un numero di funzioni hash e di lunghezze della lista di stringhe da inizializzare variabile, mantenendo sempre uno speed up di circa 5.

3.5. Confronto prestazioni all'aumentare del numero di indirizzi da inizializzare

Per dimostratore ulteriormente quanto affermato in precedenza, sono di seguito riportati i grafici che mostrano il variare dei tempi di esecuzione e dello speed up tra l'inizializzazione sequenziale e parallela, con 16 thread a disposizione, del filtro al variare del numero di indirizzi "sicuri" (100, 300, 500 e 1000).

Riportiamo inoltre il grafico dello speedup relativo ottenuto divedendo i tempi ottenuti in precedenza.

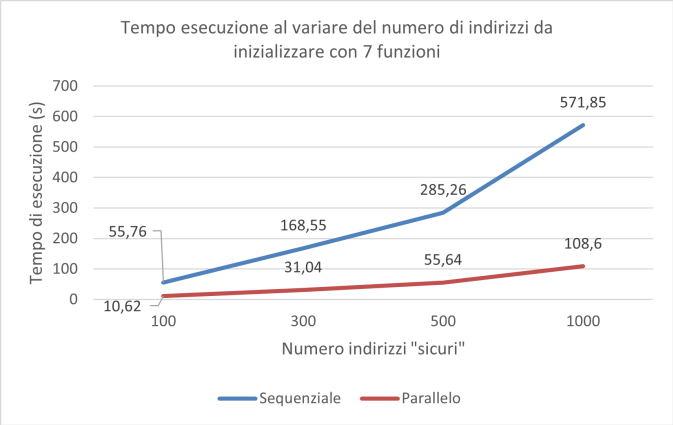

Figure 5. Tempi di inizializzazione del filtro al variare del numero di indirizzi sicuri con 7 funzioni hash

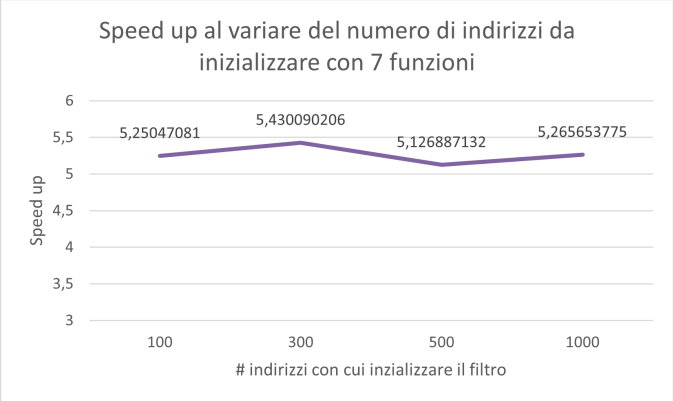

Figure 6. Speed Up tra esecuzione sequenziale e parallela a 16 thread al variare del numero di stringhe da inizializzare

Possiamo notare come, anche decuplicando il numero di indirizzi sicuri con cui inizializzare il filtro, lo speed up non sembri variare in maniera significativa, il che conferma in maniera definitiva il fatto che l'implementazione parallela da me proposta è valida per qualsiasi numero abbastanza grande di indirizzi da inizializzare.

4. Conclusione

Per quanto riportato nelle sezioni precedenti, possiamo affermare che la mia implementazione parallela per quanto riguarda l'inizializzazione del filtro di Bloom abbia un evidente convenienza in termini di tempo di esecuzione rispetto alla versione sequenziale, andando a sfruttare a pieno le potenzialità del processore, come evidenziato anche dalle immagini di seguito riportate.

Figure 7. Utilizzo core logici CPU per l'inizializzazione sequenziale

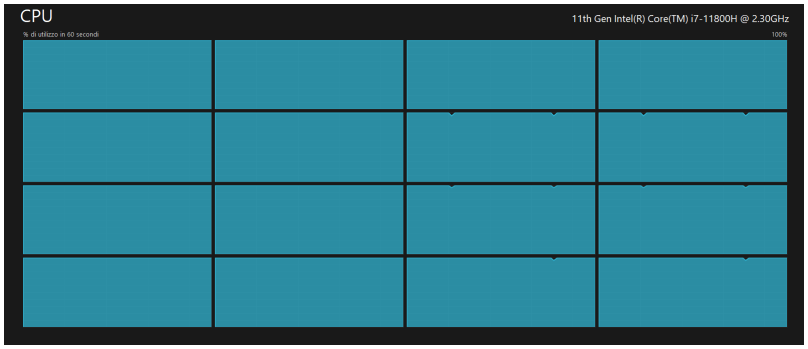

Figure 8. Utilizzo core logici CPU per l'inizializzazione parallela

Per quanto riguarda invece il controllo dell'appartenenza all'insieme della singola stringa, tale operazione, pur essendo facilmente parallelizzabile, è stata mantenuta esclusivamente sequenziale in quanto si presume di dover analizzare un singolo elemento alla volta e non una lista di elementi come invece avviene per l'inizializzazione del filtro, motivo per il quale i benefici della parallelizzazione non solo erano del tutto assenti ma addirittura l'implementazione sequenziale è risultata essere nettamente più rapida rispetto a quella parallela, a causa del overhead dovuto alla generazione dei thread o dei processi.