UNIVERSITÀ DI FIRENZE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di Laurea Magistrale in Ingegneria Informatica

Pattern Recognition, Comparing Sequential and Parallel version

Docente:

Marco Bertini

Candidato:

Daniele Morganti

ANNO ACCADEMICO 2024/2025

Indice

1 Introduzione

1.1 Contesto e obbiettivi

Questo report presenta lo sviluppo di un algoritmo di pattern recognition per serie temporali, con l'obiettivo di identificare l'indice della migliore corrispondenza tra una lunga serie temporale e una query più piccola. Il criterio di similarità utilizzato è la somma delle differenze assolute (SAD), una metrica ampiamente applicata per quantificare la distanza tra due sequenze di valori numerici. Il progetto consiste in due fasi principali: l'implementazione di una versione sequenziale dell'algoritmo e la sua ottimizzazione tramite parallelizzazione.

L'algoritmo sequenziale calcola la similarità tra la query e ogni sottosequenza della serie temporale, restituendo l'indice che minimizza la distanza calcolata. Tuttavia, a causa del costo computazionale di tale approccio su grandi serie temporali, è stata introdotta una versione parallela dell'algoritmo, progettata per ridurre i tempi di esecuzione distribuendo i calcoli su più thread o processori.

Questo report si concentrerà prima sull'analisi dell'algoritmo nella sua versione sequenziale, quindi descriverà in dettaglio il processo di parallelizzazione, evidenziando i miglioramenti delle prestazioni. Infine, verranno discussi i risultati sperimentali, confrontando le due implementazioni, con particolare attenzione all'efficienza e alla scalabilità del metodo parallelo rispetto a quello sequenziale.

1.2 Approccio Generale

Il progetto è stato sviluppato utilizzando il linguaggio C++, scelto per la sua efficienza e il controllo a basso livello sulle risorse del sistema. Per la parallelizzazione del codice, è stato utilizzato OpenMP (Open Multi-Processing), un'API che consente di sfruttare i moderni processori multi-core e multiprocessore per l'esecuzione parallela.

OpenMP è stato integrato nell'algoritmo per distribuire i carichi di lavoro su più thread. In particolare, le operazioni di confronto tra la query e le varie sotto-sequenze della serie temporale sono state suddivise tra i thread disponibili, riducendo significativamente i tempi di esecuzione. Con OpenMP, è stato possibile parallelizzare facilmente i cicli for che eseguono questi confronti, attraverso l'uso di direttive come #pragma omp parallel for.

Il vantaggio principale di OpenMP è la semplicità con cui consente di trasformare un codice sequenziale in parallelo, mantenendo la leggibilità e gestendo automaticamente la creazione, sincronizzazione e terminazione dei thread. Inoltre, grazie al supporto di C++ e OpenMP, è possibile sfruttare al meglio l'architettura hardware del sistema, bilanciando il carico su tutti i core del processore e migliorando le prestazioni complessive dell'algoritmo.

In sintesi, l'implementazione in C++ e la parallelizzazione con OpenMP permettono di ottenere un'ottimizzazione significativa, sia in termini di tempo di esecuzione che di efficienza computazionale, sfruttando la potenza del calcolo parallelo e la capacità di C++ di gestire risorse a basso livello.

2 Implementazione

2.1 Versione sequenziale

In questo capitolo viene descritta l'implementazione seriale dell'algoritmo di ricerca di una sottoserie all'interno di una serie più grande utilizzando la metrica SAD (Sum of Absolute Differences). L'algoritmo cerca il punto nella serie principale (detta anche longSeries) dove la somiglianza con la sottoserie (detta anche shortSeries) è massima, misurata in termini di SAD, ossia la somma delle differenze assolute tra gli elementi corrispondenti delle due serie.

L'algoritmo implementato è semplice e intuitivo: scorre la longSeries a partire dall'indice zero fino alla posizione longSizeshortSize (dove longSize è la dimensione della serie più lunga e shortSize è la dimensione della serie più corta), confrontando la sottoserie di lunghezza shortSize a partire da ogni posizione con la shortSeries e calcolando il valore SAD tra le due.

L'indice che minimizza la metrica SAD rappresenta il punto di inizio del miglior match trovato, ovvero la posizione dove la shortSeries si avvicina di più alla longSeries.

Di seguito il codice della funzione che esegue la ricerca in modo sequenziale:

```
int findBestMatch_sequential(const std::vector<double>& longSeries, const
    std::vector<double>& shortSeries) {
    int longSize = longSeries.size();
    int shortSize = shortSeries.size();
    double minSAD = std::numeric_limits<double>::max();
```

```
    int bestMatchIdx = -1;

    for (int i = 0; i <= longSize - shortSize; ++i) {
        double sad = calculateSAD(longSeries, shortSeries, i);
        if (sad < minSAD) {
            {
                minSAD = sad;
                bestMatchIdx = i;
            }
        }
    }
    return bestMatchIdx;
}
```

Analisi del Codice La funzione prende come input due vettori di tipo std::vector¡double¿, uno per la serie più lunga (longSeries) e uno per la serie più corta (shortSeries). Viene inizializzata la variabile minSAD al valore massimo rappresentabile da un double, per garantire che qualsiasi valore di SAD calcolato durante l'esecuzione possa sostituire questa iniziale soglia. Viene inizializzato l'indice bestMatchIdx a -1, come valore sentinella per indicare che nessun match è stato trovato inizialmente.

Il ciclo for scorre tutti i possibili punti di partenza per confrontare la shortSeries all'interno della longSeries, fermandosi a longSize - shortSize. In questo modo, l'algoritmo evita errori di overflow accedendo a elementi fuori dai limiti del vettore. Per ogni posizione i, la funzione calculateSAD calcola il valore di SAD tra la shortSeries e la sottoserie della longSeries che inizia in i.

La funzione calculateSAD è una funzione ausiliaria che confronta le due serie elemento per elemento, calcolando la somma delle differenze assolute tra i valori corrispondenti (detta SAD). Se il valore SAD calcolato per l'indice i è inferiore all'attuale minSAD, il valore di minSAD viene aggiornato e l'indice bestMatchIdx viene impostato a i. In questo modo, il ciclo conserva solo l'indice del match con la somiglianza maggiore fino a quel momento. Restituzione del Risultato:

Al termine del ciclo, la funzione restituisce bestMatchIdx, ovvero l'indice di partenza all'interno della longSeries in cui è stato trovato il miglior match con la shortSeries.

Complessità Computazionale L'algoritmo ha una complessità temporale di O(nm), dove n è la lunghezza della longSeries e m è la lunghezza della shortSeries. Ogni iterazione del ciclo

principale esegue un confronto con complessità O(m), e il ciclo viene ripetuto n volte.

2.2 Versione parallela

In questo capitolo viene descritta la versione parallela dell'algoritmo di ricerca di una sottoserie all'interno di una serie più grande, implementata con OpenMP per sfruttare il parallelismo. L'obiettivo della parallelizzazione è migliorare le prestazioni rispetto alla versione sequenziale, distribuendo i calcoli su più core della CPU. La logica dell'algoritmo resta la stessa, ma i calcoli della metrica SAD per ogni sottoserie della longSeries vengono eseguiti in parallelo.

L'algoritmo parallelo, rispetto alla versione seriale, introduce l'uso di OpenMP per distribuire il carico di lavoro tra più thread. OpenMP permette di creare e gestire thread in maniera semplice tramite direttive, come *#pragma omp parallel for*, inserite direttamente nel codice. Di seguito, viene riportato il codice dell'algoritmo parallelo:

```
int findBestMatch_parallelized(const std::vector<double>& longSeries, const
    std::vector<double>& shortSeries, int numThreads) {
    int longSize = longSeries.size();
    int shortSize = shortSeries.size();
    double minSAD = std::numeric_limits<double>::max();
    int bestMatchIdx = -1;

    omp_set_num_threads(numThreads);

    #pragma omp parallel for
    for (int i = 0; i <= longSize - shortSize; ++i) {
        double sad = calculateSAD(longSeries, shortSeries, i);
        if (sad < minSAD) {
            #pragma omp critical
            {
                minSAD = sad;
                bestMatchIdx = i;
            }
        }
    }
    return bestMatchIdx;
}
```

La funzione findBestMatch riceve in input anche il parametro numThreads, che indica il numero di thread da utilizzare. Questo permette di configurare il parallelismo per adattarsi al numero di core disponibili. La variabile minSAD è inizializzata con il valore massimo rap-

presentabile da un double, e bestMatchIdx è inizializzato a -1, come nella versione seriale, per mantenere la logica invariata.

La funzione *omp_set_num_threads(numThreads)* configura il numero di thread che OpenMP utilizzerà. In questo modo, l'esecuzione dell'algoritmo può essere adattata dinamicamente in base al numero di thread specificato dal codice utente.

La direttiva *#pragma omp parallel for* indica a OpenMP di parallelizzare il ciclo for. In pratica, le iterazioni del ciclo vengono distribuite tra i thread disponibili, riducendo il tempo di esecuzione totale. Ogni thread esegue una parte delle iterazioni in parallelo con gli altri thread. Ad ogni iterazione, il thread calcola la metrica SAD tra la shortSeries e una sottosequenza di longSeries, iniziando dalla posizione i. Sezione Critica:

Nel caso in cui la metrica SAD calcolata sia inferiore al valore di minSAD, il thread esegue un blocco *#pragma omp critical*. Questa direttiva garantisce che solo un thread alla volta possa accedere e modificare le variabili minSAD e bestMatchIdx, prevenendo potenziali conflitti tra i thread. Il blocco critical è necessario per evitare condizioni di corsa (race conditions) che potrebbero verificarsi quando più thread tentano di aggiornare contemporaneamente il miglior risultato trovato. La sezione critica garantisce che solo un thread alla volta possa confrontare e aggiornare minSAD e bestMatchIdx, preservando la correttezza dei risultati.

Come nella versione seriale, la funzione restituisce l'indice bestMatchIdx della posizione in cui è stato trovato il miglior match, calcolato in para

3 Risultati

In questo capitolo vengono presentati e analizzati i risultati sperimentali ottenuti eseguendo sia la versione sequenziale che la versione parallela dell'algoritmo di ricerca di una sottoserie all'interno di una serie più lunga, utilizzando la metrica SAD (Sum of Absolute Differences) come criterio di somiglianza. L'obiettivo è valutare le prestazioni delle due versioni in termini di tempo di esecuzione, efficienza e scalabilità.

Tutti gli esperimenti di seguito sono stati eseguiti su un processore Intel® Core™ i7-12700K con 12 core

Obiettivi dell'Analisi L'analisi dei risultati sperimentali si propone di rispondere a diverse domande:

- Quanto è significativa la riduzione del tempo di esecuzione ottenuta grazie alla parallelizzazione rispetto alla versione seriale?
- Come varia il tempo di esecuzione in funzione del numero di thread utilizzati nella versione parallela?
- Come varia lo speedup in funzione della lunghezza della query?
- Qual è l'efficienza della parallelizzazione?

Metodologia dei Test I test sono stati eseguiti utilizzando come serie temporale il valore del Bitcoin durante l'intero anno 2019 minuto per minuto, variando la lunghezza della query (generata randomicamente) per valutare come le due versioni si comportano al crescere del carico computazionale. Inoltre, per la versione parallela, sono stati eseguiti test variando il numero di thread da 1 (simulando l'esecuzione seriale) fino al numero massimo di core disponibili, per esaminare l'impatto della parallelizzazione sul tempo di esecuzione.

Metriche di Valutazione Le principali metriche utilizzate per il confronto tra le due versioni sono le seguenti:

Tempo di esecuzione: tempo necessario per completare l'algoritmo, misurato in millisecondi. Questa metrica permette di osservare la riduzione di tempo ottenuta dalla parallelizzazione. Speedup: definito come il rapporto tra il tempo di esecuzione della versione seriale e il tempo di esecuzione della versione parallela con n thread. Lo speedup ideale è direttamente proporzionale al numero di thread, ma in pratica è influenzato dall'overhead di sincronizzazione. Efficienza: rappresenta la frazione di speedup ottenuta rispetto al numero di thread e viene calcolata come $efficienza = \frac{speedup}{numero\ di\ thread}$. Questa misura consente di valutare quanto bene la versione parallela scala con l'aumentare del numero di thread.

3.1 Primo Esperimento

Nel primo esperimento effettuato il codice è stato eseguito più volte incrementando ogni volta la lunghezza della query in ingresso, partendo da una lunghezza di 100 campioni, raddoppiando fino ad arrivare a 25600 campioni. In questo esperimento il numero di thread è stato mantenuto costante a 20, pari al numero di thread logici che il processore può eseguire contemporaneamente.

Dal grafico in figura 3.1 è possibile osservare come l'andamento del tempo di esecuzione in entrambi i casi sia lineare con la lunghezza della query (coerente con l'analisi computazionale effettuata nel capitolo precedente) e di come lo speedup rimanga piuttosto costante. Possiamo quindi dedurre da questo primo esperimento che la lunghezza della query non influisca in modo rilevante sullo speedup della parallelizzazione.

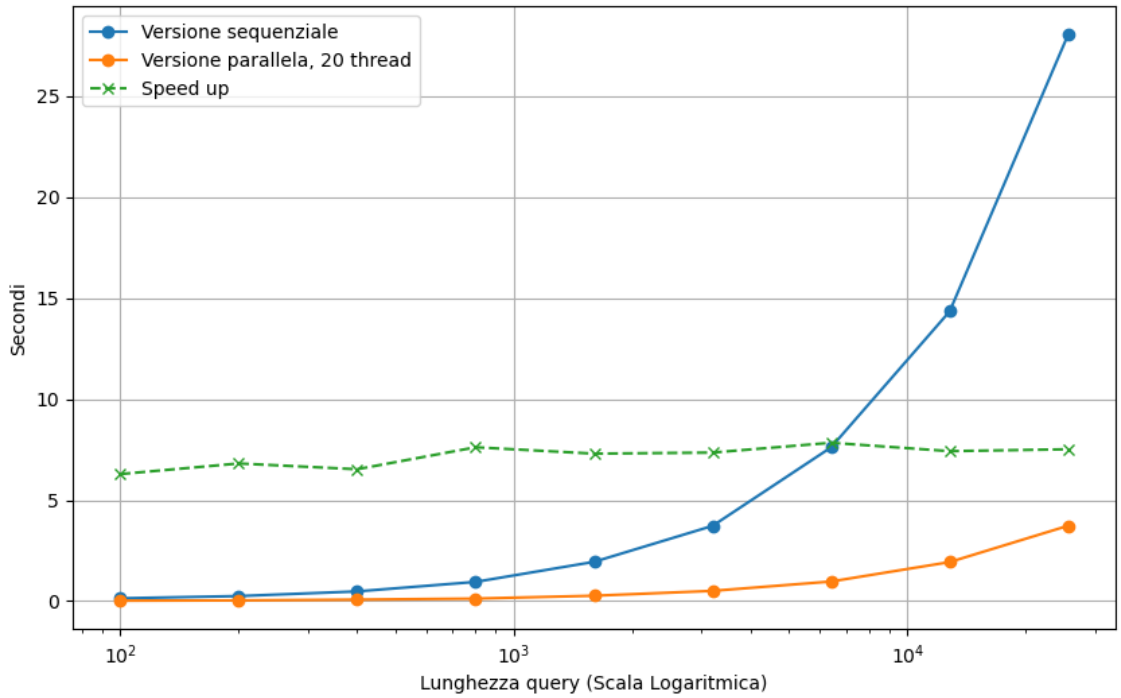

Figura 3.1: Tempo di esecuzione in funzione della lunghezza della query

Lunghezza Query	tempo sequenziale	tempo parallelo	speedup
100	0.137443	0.021862	6.28684
200	0.250124	0.036716	6.8124
400	0.480059	0.073559	6.52618
800	0.952056	0.125044	7.61377
1600	1.9446	0.266433	7.29865
3200	3.7257	0.506788	7.3516
6400	7.61209	0.971283	7.83714
12800	14.3403	1.93228	7.42143
25600	28.0418	3.73127	7.51536

3.2 Secondo Esperimento

In questo secondo esperimento è stata testata la versione parallela dell'algoritmo aumentando progressivamente il numero di thread, andando ad osservare come si comporta lo speedup mantenendo il carico computazionale complessivo invariato.

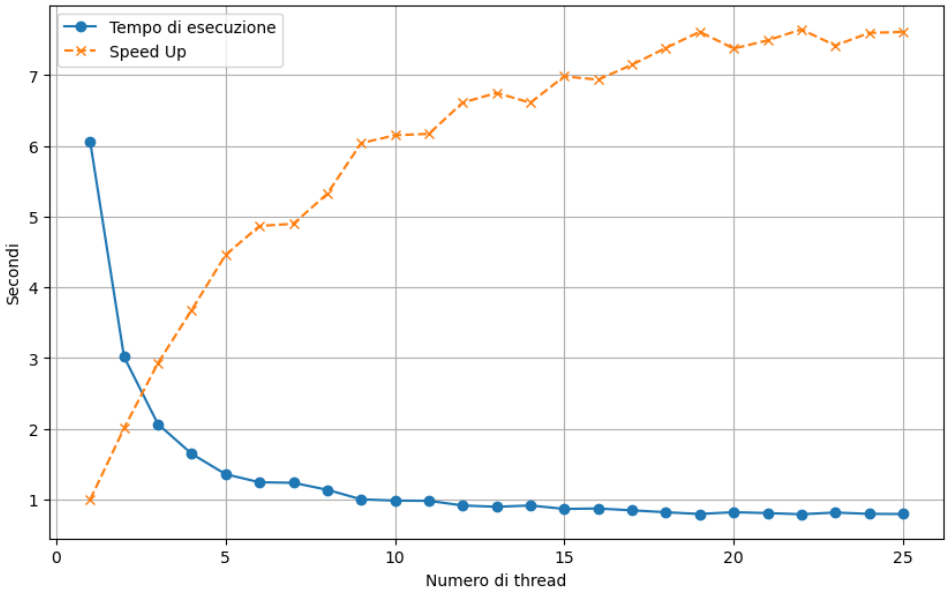

Figura 3.2: Tempo di esecuzione e speedup in funzione del numero di thread a parità di lunghezza query

lunghezza query	tempo di esecuzione	numero di thread
100	6.06079	1
100	3.01752	2
100	2.06883	3
100	1.6454	4
100	1.35845	5
100	1.24474	6
100	1.23707	7
100	1.13838	8
100	1.00343	9
100	0.985311	10
100	0.982045	11
100	0.916448	12
100	0.897839	13
100	0.91665	14
100	0.867676	15
100	0.87368	16
100	0.847737	17
100	0.820546	18
100	0.795928	19
100	0.821551	20
100	0.808904	21
100	0.792418	22
100	0.816738	23
100	0.797637	24
100	0.796084	25

Come si può vedere in figura 3.2 abbiamo un significativo incremento delle prestazioni all'aumentare dei thread. Come era possibile prevedere, lo speedup sale fino intorno ai 20 thread per poi assestarsi intorno a 7.5. Questo è ragionevole, considerando che la CPU con la quale gli esperimenti sono stati condotti possiede proprio 20 thread logici.

4 Conclusioni

L'obiettivo principale di questo elaborato era valutare l'efficacia della parallelizzazione nel ridurre il tempo di esecuzione e migliorare le prestazioni di un semplice algoritmo di pattern recognition in versione seriale.

in conclusione possiamo dire di aver osservato una riduzione significativa del tempo di esecuzione: La versione parallela ha dimostrato di essere significativamente più veloce della versione seriale, indipendentemente dalla dimensione dell'input. All'aumentare del numero di thread, il tempo di esecuzione si è ridotto, confermando l'efficacia della parallelizzazione. Questo vantaggio è stato particolarmente evidente su una serie temporale molto lunga, dove i costi computazionali della metrica SAD hanno potuto essere distribuiti tra i thread.

Il confronto dello speedup ha mostrato un incremento quasi lineare fino a un certo numero di thread, oltre il quale l'overhead di sincronizzazione ha iniziato a limitare ulteriormente i guadagni in termini di velocità. Sebbene lo speedup ideale sarebbe proporzionale al numero di thread, l'utilizzo della direttiva *#pragma omp critica* per garantire la correttezza dei dati ha introdotto una sincronizzazione necessaria, ma costosa, che ha ridotto il miglioramento prestazionale.

In conclusione, l'implementazione parallela dell'algoritmo ha dimostrato di essere un approccio efficace per ridurre il tempo di esecuzione, pur con alcune limitazioni di scalabilità dovute alla sincronizzazione tra thread. I risultati sperimentali mostrano che l'uso di OpenMP rappresenta una soluzione valida per migliorare l'efficienza computazionale dell'algoritmo preso

in considerazione, offrendo spunti per ulteriori ottimizzazioni e sviluppi futuri per applicazioni su larga scala.