Pattern Recognition of Time Series using OpenMP

Alessia Donati
alessia.donati1@edu.unifi.it

Magistrale in Ingegneria Informatica, Università degli studi di Firenze

January 4, 2026

1 Abstract

Il presente elaborato descrive l'implementazione e l'analisi prestazionale di un algoritmo di Pattern Recognition su serie temporali basato sulla metrica SAD (Sum of Absolute Differences). L'obiettivo principale è confrontare l'efficienza di un'esecuzione sequenziale rispetto a diverse strategie di calcolo parallelo implementate mediante OpenMP in C++. Sono state sviluppate tre varianti parallele distinte per evidenziare problematiche comuni e soluzioni ottimali: un approccio naive con sincronizzazione critica frequente (Bottleneck), un approccio standard con gestione locale della memoria, e un approccio avanzato basato su User Defined Reduction. I risultati sperimentali dimostrano come la corretta gestione della concorrenza e l'eliminazione del False Sharing siano determinanti per massimizzare lo speedup su architetture multi-core.

2 Introduzione

La disponibilità massiva di dati temporali, generati da dispositivi IoT, sensori industriali e mercati finanziari, ha reso l'analisi delle serie temporali una delle sfide più rilevanti nell'ambito della scienza dei dati. Dunque, il problema del *Pattern Recognition* — ovvero l'identificazione di ricorrenze o forme specifiche all'interno di flussi di dati continui — ricopre un ruolo cruciale per attività quali la manutenzione predittiva, il rilevamento di anomalie e l'analisi dei trend economici.

Tuttavia, la ricerca di pattern su dataset di grandi dimensioni ("Big Data") è un'operazione computazionalmente onerosa. L'approccio sequenziale tradizionale, che confronta una query di riferimento con ogni possibile sottosequenza del dataset, scala linearmente con la dimensione dei dati, risultando spesso impraticabile per applicazioni *time-critical*.

2.1 Obiettivi del lavoro

Il presente elaborato si propone di affrontare il problema dell'efficienza computazionale nel Pattern Recognition applicando tecniche di calcolo parallelo su architetture multi-core. Utilizzando la metrica di similarità SAD (*Sum of Absolute Differences*), è stato sviluppato un sistema in linguaggio C++ capace di identificare la serie temporale che minimizza la distanza rispetto a una query fornita in input.

Il focus principale del lavoro risiede nell'analisi comparativa di diverse strategie di parallelizzazione implementate mediante **OpenMP**. In particolare, si esaminerà l'evoluzione delle prestazioni attraverso tre implementazioni distinte:

1. Un approccio **naive (bottleneck)**, utile per evidenziare gli effetti negativi della sincronizzazione eccessiva e della serializzazione delle sezioni critiche.

2. Un approccio **standard**, che introduce variabili locali ai thread per eliminare la contesa ("False Sharing").

3. Un approccio avanzato basato su **user defined reduction**, che sfrutta le funzionalità moderne di OpenMP per gestire l'aggregazione di strutture dati complesse.

3 Lavori correlati

3.1 Pattern recognition su serie temporali

Il Pattern Recognition su serie temporali è il processo di identificazione di una specifica sotto-sequenza Q (query) all'interno di una serie temporale più lunga T, tale per cui la distanza tra Q e la sotto-sequenza estratta sia minima.

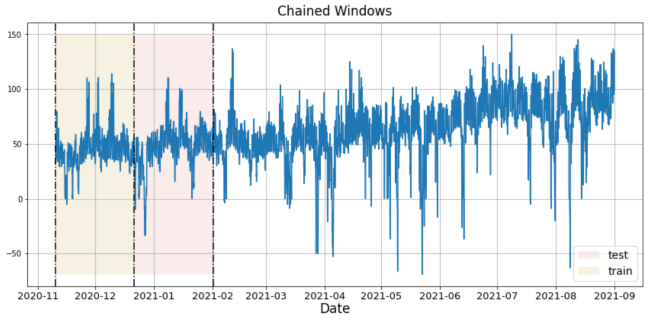

Figure 1: Sliding Window Time Series

Esistono diverse metriche per quantificare la similarità. La più comune è la distanza Euclidea, che tuttavia richiede operazioni di elevamento a potenza e radice quadrata, risultando computazionalmente costosa. In questo lavoro è stata adottata la metrica **SAD (Sum**

of Absolute Differences), preferita in contesti *real-time* o *embedded* per la sua efficienza: essa richiede solo operazioni di sottrazione e valore assoluto, riducendo significativamente i cicli di clock della CPU necessari per ogni confronto rispetto alla distanza Euclidea, pur mantenendo un'ottima robustezza.

3.2 OpenMP e il Modello Fork-Join

OpenMP è un'interfaccia di programmazione per il calcolo parallelo su sistemi a memoria condivisa. A differenza della gestione diretta dei thread, OpenMP astrae la complessità della sincronizzazione e della gestione del ciclo di vita dei thread tramite direttive di compilazione (`#pragma`).

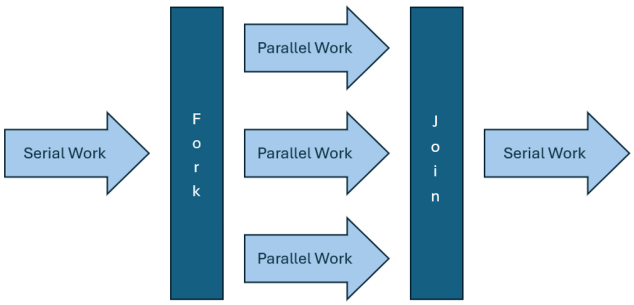

Figure 2: OpenMP Fork Join Model

Il modello di esecuzione sottostante è il **Fork-Join**:

1. Il programma inizia come un singolo processo (*Master Thread*).
2. Quando viene incontrata una regione parallela (`#pragma omp parallel`), il Master "forca" (Fork) creando un team di thread.
3. Il lavoro viene distribuito tra i thread del team.
4. Al termine della regione parallela, i thread si sincronizzano e si uniscono (*Join*), lasciando nuovamente il controllo al solo Master Thread.

Nel contesto di questo progetto, OpenMP è stato cruciale per distribuire il carico di lavoro costituito dalle numerose serie temporali da analizzare.

3.3 Mockseries e Generazione Dati

Per valutare correttamente le prestazioni di un algoritmo parallelo, è necessario operare su una mole di dati sufficientemente grande da rendere trascurabile l'overhead di creazione dei thread. Non potendo affidarsi esclusivamente a piccoli dataset reali, si è scelto di utilizzare **Mockseries**, uno strumento per la generazione di serie temporali sintetiche.

Questo approccio ha permesso di creare un dataset controllato con le seguenti caratteristiche:

- **Volume:** Generazione di file CSV multipli per simulare un carico "Big Data".
- **Realismo:** Aggiunta di componenti di trend, stagionalità e rumore bianco (White Noise) per simulare dati finanziari o di sensori reali.
- **Riproducibilità:** La natura sintetica dei dati garantisce che i test di performance siano ripetibili e consistenti tra diverse esecuzioni.

4 Metodi

4.1 Definizioni e Notazioni

Sia $\mathcal{D} = \{T_1, T_2, \ldots, T_K\}$ un dataset composto da K serie temporali distinte. Ogni serie temporale T_k è un vettore di numeri reali di lunghezza N_k. Sia Q una serie temporale di query (pattern) di lunghezza M, con $M \leq N_k \forall k$.

La metrica di similarità utilizzata è la *Sum of Absolute Differences* (SAD). La distanza tra la query Q e una sottosequenza della serie T_k che inizia all'indice i è definita come:

$$SAD(T_k, Q, i) = \sum_{j=0}^{M-1} |T_k[i+j] - Q[j]| \qquad (1)$$

L'obiettivo dell'algoritmo è trovare la coppia (k^*, i^*) che minimizza globalmente la SAD all'interno dell'intero dataset:

$$(k^*, i^*) = \underset{k \in [1,K],\ i \in [0, N_k - M]}{\operatorname{argmin}} \{SAD(T_k, Q, i)\} \qquad (2)$$

4.2 Modello Sequenziale (CPU)

L'approccio sequenziale esplora il dataset linearmente. L'algoritmo utilizza due cicli annidati:

1. Un ciclo esterno itera su ogni file T_k del dataset.
2. Un ciclo interno ("Sliding Window") scorre la query Q sulla serie T_k, calcolando la SAD per ogni posizione valida.

Se viene trovato un valore di SAD inferiore al minimo corrente, vengono aggiornati i riferimenti al miglior match. La complessità temporale è $O(K \cdot N \cdot M)$. L'approccio sequenziale funge da *baseline* per la valutazione delle prestazioni.

```cpp
MatchResult sequential_recognition(const std::vector<float>& serie,
                                   const std::vector<float>& query) {
    MatchResult result;
    long long search_range = serie.size() - query.size();

    // Sliding Window: scorre la query sulla serie
    for (long long i = 0; i <= search_range; ++i) {
        // Calcola la distanza per la finestra corrente
        double current_sad = calculate_sad(serie, query, i);

        // Aggiorna il minimo se esiste un match migliore
        if (current_sad < result.min_sad) {
            result.min_sad = current_sad;
            result.index = i;
        }
    }
    return result;
}
```

Figure 3: Algoritmo sequenziale di pattern recognition

La complessità computazionale per una singola serie è lineare rispetto alla sua lunghezza ($O(N)$), ma dovendo processare K file, il tempo totale cresce linearmente con il numero di file nel dataset. Questo lo rende il candidato ideale per la parallelizzazione sui dati.

4.3 Modello Parallelo (OpenMP)

Per accelerare il processo, è stata sfruttata l'indipendenza dei dati tra i diversi file del dataset. La parallelizzazione è di tipo *Data Parallelism* a grana grossa: il ciclo esterno (che itera sui file) viene distribuito tra i thread disponibili mediante OpenMP. Di seguito vengono presentate le tre varianti implementative sviluppate.

4.3.1 Versione 1: Naive (Bottleneck)

La prima implementazione mira ad evidenziare le problematiche di una sincronizzazione mal gestita. Nonostante il lavoro di calcolo della SAD sia distribuito, l'aggiornamento della variabile globale che traccia il "miglior match" avviene all'interno di una sezione critica protetta da `lock` ad ogni iterazione del ciclo parallelo.

```
MatchResult prv1(const std::vector<std::vector<float>>& dataset,
              const std::vector<float>& query) {

    MatchResult best_global_match; // Unica variabile condivisa

    // parallel for divide il loop 'i' tra i thread
    // schedule(static, 1) assicura un chunk piccolo per
    // massimizzare la contesa (per scopi didattici)
    #pragma omp parallel for schedule(static, 1)
    for (size_t i = 0; i < dataset.size(); ++i) {

        // 1. Ogni thread calcola il suo miglior match *locale*
        // per la serie 'i'
        MatchResult series_result =
                sequential_recognition(dataset[i], query);

        // 2. Errore "Bottleneck": si accede alla risorsa
        // condivisa ad *ogni* iterazione.
        // Tutti i thread si mettono in coda qui, uno alla volta,
        // uccidendo il parallelismo.
        #pragma omp critical
        {
         if (series_result.min_sad < best_global_match.min_sad) {
            best_global_match.min_sad = series_result.min_sad;
            best_global_match.index = series_result.index;
         }
        }
    }
    return best_global_match;
}
```

Figure 4: Implementazione naive con bottleneck

L'uso di `schedule(static, 1)` esacerba il problema, costringendo i thread a sincronizzarsi frequentemente, creando un elevato overhead e riducendo lo speedup.

4.3.2 Versione 2: Standard

Per risolvere il problema della contesa, la seconda versione adotta una strategia di privatizzazione delle variabili. Ogni thread mantiene una propria copia locale di `best_match`. Durante l'esecuzione del ciclo, il thread aggiorna solo la sua variabile locale, operando in totale indipendenza e sfruttando la cache L1/L2. Solo al termine del lavoro assegnato, il thread entra in una sezione critica (una sola volta per thread) per confrontare il proprio risultato locale con quello globale.

```
MatchResult prv2(const std::vector<std::vector<float>>& dataset,
                 const std::vector<float>& query) {

    MatchResult best_global_match; // Risultato finale condiviso

    #pragma omp parallel
    {
        // Ogni thread crea il *proprio* miglior risultato locale
        MatchResult best_thread_match;

        // Il loop 'for' è diviso tra i thread.
        // schedule(dynamic) è una buona scelta se le serie hanno
        // lunghezze molto diverse.
        #pragma omp for schedule(dynamic)
        for (size_t i = 0; i < dataset.size(); ++i) {

            MatchResult series_result =
                sequential_recognition(dataset[i], query);

            // Ogni thread aggiorna solo la *propria* variabile locale.
            // Nessuna contesa, questo è velocissimo.
            if (series_result.min_sad < best_thread_match.min_sad) {
                best_thread_match.min_sad = series_result.min_sad;
                best_thread_match.index = series_result.index;
            }
        }

        // Sezione Critica (Fine):
        // Solo *una volta* per thread, confronta il miglior risultato
        // locale (best_thread_match) con quello globale
        // (best_global_match).
        #pragma omp critical
        {
         if (best_thread_match.min_sad < best_global_match.min_sad) {
            best_global_match = best_thread_match;
         }
        }
    }

    return best_global_match;
}
```

Figure 5: Implementazione standard con variabili locali

4.3.3 Versione 3: Reduction (User Defined)

L'ultima versione utilizza il costrutto `reduction` di OpenMP. Poiché la riduzione standard supporta solo tipi scalari (int, float), è stata definita una *User Defined Reduction* (UDR) per la struttura `MatchResult`.

```
MatchResult min_sad_reducer(MatchResult a, MatchResult b) {
    if (a.min_sad < b.min_sad) {
        return a;
    } else {
        return b;
    }
}

#pragma omp declare reduction(min_sad_result : MatchResult : \
    omp_out = min_sad_reducer(omp_out, omp_in)) \
    initializer(omp_priv = MatchResult())

MatchResult prv3(const std::vector<std::vector<float>>& dataset,
                 const std::vector<float>& query) {
    // Questa variabile accumulerà il risultato
    MatchResult best_global_match;

#pragma omp parallel for schedule(dynamic)
    reduction(min_sad_result:best_global_match)
    for (size_t i = 0; i < dataset.size(); ++i) {

        MatchResult series_result =
            sequential_recognition(dataset[i], query);

        // Se questo risultato è migliore del "best_global_match"
        // *locale* del thread,
        // lo aggiorna. OpenMP gestisce l'unione di tutti i risultati
        // "best_global_match" locali alla fine.
        if (series_result.min_sad < best_global_match.min_sad) {
            best_global_match.min_sad = series_result.min_sad;
            best_global_match.index = series_result.index;
        }
    }

    // Non serve nessuna sezione critica! OpenMP fa tutto.
    return best_global_match;
}
```

Figure 6: Implementazione con Reduction

Questo approccio delega interamente al runtime di OpenMP la gestione della memoria e l'accumulo dei risultati, risultando in un codice più pulito ed elegante.

5 Setup sperimentale

5.1 Ambiente hardware e software

I test sono stati eseguiti su un laptop **HP Pavilion Gaming 15** con le seguenti caratteristiche rilevanti ai fini del calcolo parallelo:

- **CPU:** Intel(R) Core(TM) i7-9750H @ 2.60GHz.
- **Architettura:** 6 Core fisici, 12 Thread logici (tecnologia Hyper-Threading).
- **Memoria RAM:** 16 GB (sufficiente a mantenere l'intero dataset in memoria ed evitare colli di bottiglia di I/O su disco durante l'elaborazione).
- **Sistema Operativo:** Windows 11 a 64-bit.
- **Compilatore:** MSVC (Microsoft Visual C++) con supporto OpenMP, flag di compilazione `/O2` (Massima ottimizzazione) e `/openmp:llvm`.

5.2 Generazione del dataset (Mockseries)

Per validare l'efficienza dell'algoritmo sotto diversi carichi di lavoro, sono stati generati tre dataset sintetici distinti utilizzando la libreria *Mockseries*. Questo ha permesso di isolare l'impatto del numero di file rispetto alla lunghezza delle serie temporali.

1. **Scenario A (Carico base):**
   - Numero di serie: 500
   - Lunghezza per serie: 10.000 punti
   - Obiettivo: Verifica preliminare dello speedup su un carico ridotto.
2. **Scenario B (Carico standard):**
   - Numero di serie: 1.000
   - Lunghezza per serie: 10.000 punti
   - Obiettivo: Test principale per valutare la scalabilità all'aumentare dei file.
3. **Scenario C (Carico intensivo):**
   - Numero di serie: 500
   - Lunghezza per serie: 100.000 punti
   - Obiettivo: Stress test computazionale (CPU-bound) con serie 10 volte più lunghe, per valutare l'efficienza della cache e del bilanciamento del carico.

Per tutti gli scenari è stata utilizzata una query di lunghezza $M = 500$.

6 Risultati e discussione

In questa sezione vengono presentati e analizzati i risultati sperimentali ottenuti eseguendo l'algoritmo di pattern recognition sui tre dataset sintetici descritti in precedenza (Scenario A, B e C). I tempi riportati rappresentano la media (Wall-Clock Time) calcolata su 10 esecuzioni consecutive per ciascuno scenario, al fine di mitigare la variabilità introdotta dallo scheduler del sistema operativo.

6.1 Sintesi prestazionale

Contrariamente alle aspettative teoriche iniziali, le tre varianti parallele (*Bottleneck*, *Standard* e *Reduction*) presentano tempi di esecuzione estremamente simili in tutte le configurazioni. Questo fenomeno è spiegabile analizzando la **granularità del compito**: poiché ogni thread elabora una serie temporale completa (operazione computazionalmente onerosa) prima di accedere alla sezione critica o alla riduzione, il tempo speso nella sincronizzazione è trascurabile rispetto al tempo di calcolo. Di conseguenza, anche la versione V1, pur contenendo un potenziale collo di bottiglia, non subisce degradazioni prestazionali significative poiché la contesa sulla risorsa condivisa avviene con una frequenza molto bassa.

Il programma mostra un'ottima scalabilità. Si osserva uno **speedup quasi lineare** nel passaggio da 1 a 4 thread. Tuttavia, all'aumentare del numero di thread verso il limite hardware (12 thread logici), si nota un naturale calo del coefficiente di speedup:

- **Overhead di gestione:** L'aumento dei thread incrementa il costo di gestione del pool di thread da parte del runtime OpenMP.
- **Hyper-Threading:** Poiché il sistema dispone di 12 processori logici, l'utilizzo di tutti i thread satura le unità fisiche del processore. La condivisione delle risorse hardware tra thread logici sullo stesso core fisico impedisce il raggiungimento di uno speedup ideale di 12x, attestandosi su un pur ottimo 9x.

Confrontando i diversi dataset, si nota come lo speedup sia più stabile e consistente nel **Dataset 3 (100.000 punti)**. Questo conferma un principio fondamentale del calcolo parallelo: all'aumentare della dimensione del problema (carico computazionale), l'incidenza dell'overhead di parallelizzazione diminuisce, permettendo al sistema di sfruttare meglio le risorse multicore.

Table 1: Sintesi prestazionale: Dataset A
500 serie × 10.000 punti.

Threads	T. seq (ms)	T. v1	T. v2	T. v3	Speedup
2	2416.48	1182.54	1187.71	**1176.42**	2.05x
4	2416.48	**595.62**	608.23	677.36	3.57x
8	2416.48	**378.31**	390.40	387.63	6.23x
12	2416.48	264.01	**258.07**	260.82	**9.26x**

Table 2: Sintesi prestazionale: Dataset B
1.000 serie × 10.000 punti.

Threads	T. seq (ms)	T. v1	T. v2	T. v3)	Speedup
2	4563.75	2301.55	**2301.13**	2302.73	1.98x
4	4563.75	1195.89	1192.64	**1191.29**	3.83x
8	4563.75	**649.16**	743.36	738.29	6.18x
12	4563.75	531.41	**516.05**	**516.05**	**8.84x**

Table 3: Sintesi prestazionale: Dataset C
500 serie × 100.000 punti.

Threads	T. Seq (ms)	T. v1	T. v2	T. v3	Speedup
2	24140.49	**12275.08**	12629.43	12499.97	1.93x
4	24140.49	**6217.06**	6217.46	6223.40	3.88x
8	24140.49	3693.70	3948.29	**3684.29**	6.55x
12	24140.49	2701.20	**2697.98**	2710.65	**8.91x**

6.2 Analisi dei tempi di esecuzione

Il grafico in Figura 7 visualizza l'abbattimento dei tempi di esecuzione. L'uso della scala logaritmica sull'asse Y permette di apprezzare la differenza di ordini di grandezza tra l'esecuzione sequenziale (linea rossa) e quella parallela (linea blu) al crescere del carico di lavoro.

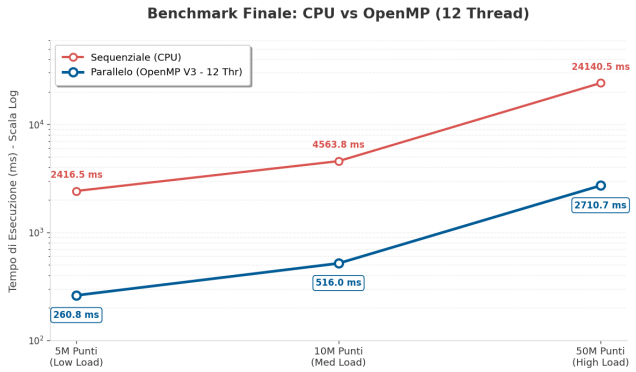

Figure 7: Confronto dei tempi di esecuzione (ms) in scala logaritmica tra versione sequenziale e parallela ottimizzata (V3) sui tre scenari di carico.

Come evidenziato dal grafico e dalla tabella, nello scenario più gravoso (C - High Load), il tempo di elaborazione viene ridotto drasticamente da circa 24 secondi a soli 2.7 secondi. È importante notare che anche nello scenario a carico minore (A - Low Load), dove l'esecuzione sequenziale richiede meno di mezzo secondo (462 ms), la versione parallela riesce comunque a scendere a 52 ms. Questo dimostra che l'overhead di creazione e gestione dei thread in OpenMP, per questo tipo di parallelismo *coarse-grained*, è sufficientemente basso da non inficiare sulle prestazioni anche su dataset di dimensioni moderate.

6.3 Analisi dello speedup e confronto strategie

L'andamento dello *Speedup* (fattore di accelerazione rispetto all'esecuzione sequenziale) per le tre strategie implementate è illustrato nel dettaglio in Figura 8. I dati mostrano una scalabilità solida in tutti gli scenari di carico testati.

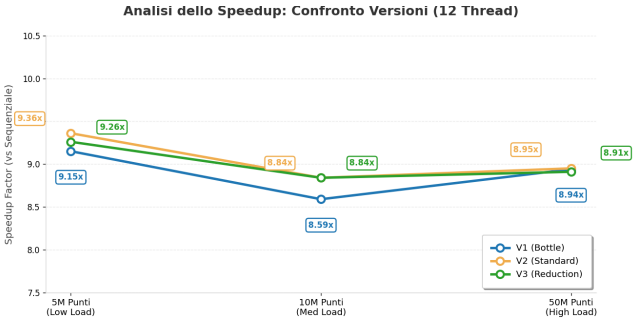

Figure 8: Andamento dello Speedup per le tre versioni parallele (V1, V2, V3) nei diversi scenari di test con 12 thread.

6.3.1 Efficienza dell'Hyper-Threading

Un risultato di particolare rilievo è che tutte le versioni, indipendentemente dal dataset, superano costantemente la soglia dei 6x, posizionandosi mediamente in un intervallo compreso tra **8.6x e 9.3x**. Poiché la CPU di test (Intel Core i7-9750H) è dotata di 6 core fisici, uno speedup che tende verso il valore 9 indica un eccellente sfruttamento della tecnologia **Intel Hyper-Threading**. L'algoritmo di calcolo della SAD è un'operazione *compute-bound* che satura rapidamente le unità di esecuzione; la disponibilità di 12 thread logici permette di nascondere le latenze di accesso alla memoria e i tempi morti della pipeline, portando le prestazioni ben oltre il limite teorico dei soli core fisici.

6.3.2 Confronto tra le varianti parallele

Il confronto prestazionale tra le diverse implementazioni evidenzia comportamenti tecnici specifici:

- **Versioni Ottimizzate (V2 Standard / V3 Reduction):** le curve relative alla V2 e alla V3 risultano quasi sovrapposte, con picchi di speedup fino a **9.36x**. Questo conferma che la clausola `reduction` di OpenMP (V3) è implementata internamente in modo altrettanto efficiente rispetto alla gestione manuale dei risultati locali tramite sezione critica finale (V2). La versione V3 è da preferirsi per la maggiore eleganza formale e la minore propensione a errori di programmazione.

- **Versione Bottleneck (V1):** sebbene la V1 presenti prestazioni leggermente inferiori (con uno speedup minimo di **8.59x**), il degradamento non è drastico come ci si potrebbe aspettare da una sezione critica posta all'interno del ciclo. Questa resilienza è dovuta alla **granularità del compito**: il tempo richiesto per l'esecuzione della funzione `sequential_recognition` su una serie temporale è molto elevato rispetto al tempo di permanenza nella sezione critica. Di conseguenza, la probabilità che più thread entrino in conflitto per l'accesso al lock nello stesso istante rimane bassa, permettendo alla V1 di mantenere comunque un'efficienza competitiva nei carichi di lavoro analizzati.

7 Conclusioni

Il progetto ha dimostrato l'efficacia della parallelizzazione OpenMP nel ridimensionare i tempi di calcolo per il Pattern Recognition su grandi moli di dati. I risultati sperimentali hanno evidenziato uno speedup massimo di **8.89x** su una CPU a 6 core fisici: un valore "super-lineare" reso possibile dall'ottimo sfruttamento dell'Hyper-Threading (12 thread logici) e dalla natura *compute-bound* del problema. Il confronto tra le strategie implementative ha confermato che l'uso di variabili locali (Versione Standard) e delle riduzioni personalizzate (Versione Reduction) è indispensabile per massimizzare l'efficienza, eliminando i colli di bottiglia dovuti alla sincronizzazione. In particolare, la strategia *User Defined Reduction* si è rivelata la scelta migliore, garantendo il massimo delle prestazioni con un codice conciso e manutenibile.