

# Progetto Sistemi Multicore

Cola Valerio & Federico Cerboni

15 Gennaio 2025

## 1 Implementazione MPI

### 1.1 Introduzione

L'approccio impiegato per implementare Sequence in MPI suddivide il numero dei pattern tra i Rank.

La divisione dei pattern è calcolata in modo da distribuire uniformemente i compiti; eventuali eccedenze sono assegnate all'ultimo processo. Per mantenere la coerenza con l'esecuzione sequenziale, dopo che ogni processo ha concluso la fase di "Ricerca" dei pattern assegnati, gli array e le variabili locali vengono progressivamente inviate al processo principale Root, che li unifica gestendo conflitti di valori come nel caso di pat\_found il quale presentava valori inferiori una volta concluso il programma. Questo processo di unione evita conteggi doppi o mancanti, salvaguardando i risultati complessivi e permettendo un calcolo accurato dei checksum. L'approccio segue un flusso logico che integra comunicazioni point-to-point (Send, Recv) e collettive (AllGather, Gatherv), minimizzando sprechi di risorse e massimizzando la scalabilità. Con questo approccio è possibile avere un programma scalabile rispetto alla lunghezza della sequenza, numero di pattern e lunghezza dei pattern.

### 1.2 Codice

Viene generata la sequenza "sequence" solo dal Root e inviata a tutti gli altri Rank con la funzione MPI\_Bcast.

Ogni Rank:

- Possiede una copia di pat\_matches.
- Crea una propria copia degli array seq\_matches e pat\_found, inizializzati a NOT\_FOUND.
- Calcola il numero di pattern assegnati, l'indice del primo e dell'ultimo pattern. Se è l'ultimo Rank, gli verranno assegnati i restanti non compresi nella divisione intera.
- Analizza solo i pattern che gli sono stati assegnati in base a my\_first\_pattern e my\_last\_pattern

Alla fine dell'esecuzione, il Root si occuperà del calcolo e la stampa dei checksum, unificando i dati delle variabili locali seq\_matches, pat\_found e pat\_matches di ogni rank.

#### 1.2.1 Gestione di seq\_matches

I Rank > 0 inviano i propri seq\_matches al Root con la MPI\_Send.

```
1 if(rank > 0){  
2     MPI_Send(seq_matches, seq_length, MPI_INT, 0, 0, MPI_COMM_WORLD);  
3 }
```

## Problema:

“Ogni Rank parte da seq\_matches = -1. Se un pattern si ripete su più Rank o se più Rank trovano un pattern allo stesso indice, seq\_matches va da -1 a 0 più volte, creando discrepanze rispetto all'esecuzione sequenziale nella quale per ogni posizione dell'array si passa solo una volta da -1 a 0. Ciò causa problemi nel calcolo del checksum finale poiché dopo che il Root ha sommato tutte le posizioni di seq\_matches di ogni Rank in un unico array avremo valori inferiori rispetto al sequenziale.”

Il Root crea local\_seq\_matches e vi inserisce man mano i seq\_matches di ogni altro Rank, sommando i valori al proprio seq\_matches.

```
1 if (rank == 0) {  
2     int *local_seq_matches;  
3     local_seq_matches = (int *)malloc( sizeof(int) * seq_length );  
4     for (int i = 1; i < size; i++) { // (1)  
5         MPI_Recv(local_seq_matches, seq_length, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);  
6  
7         for (unsigned long j = 0; j < seq_length; j++) { // (2)  
8             if (seq_matches[j] >= 0 && local_seq_matches[j] >= 0) { // (3)  
9                 seq_matches[j] = seq_matches[j] + local_seq_matches[j] + 1;  
10            } else {  
11                if(seq_matches[j] == NOT_FOUND){ // (4)  
12                    seq_matches[j] = local_seq_matches[j];  
13                }  
14            }  
15        }  
16    }  
17    free(local_seq_matches);  
18}  
19}
```

Itera per i volte in base al numero dei Rank in ogni iterazione riceverà con la MPI\_Recv il seq\_matches del Rank i e lo inserisce nel proprio locale **(1)**.

Itera per ogni indice j di seq\_matches e local\_seq\_matches **(2)**:

- Se entrambi i valori sono  $\geq 0$ , si aggiunge 1 oltre alla somma **(3)**.
- Se il seq\_matches del Root è NOT\_FOUND, si imposta al seq\_matches il valore di local\_seq\_matches **(4)**.

### 1.2.2 Gestione di pat\_found

```
1 int send_counts[size]; // (1)  
2     send_counts[rank] = my_patterns;  
3     int displs[size]; // (2)  
4     displs[rank] = my_first_pattern;  
5  
6     // (3)  
7     MPI_Allgather(&my_patterns, 1, MPI_INT, send_counts, 1, MPI_INT, MPI_COMM_WORLD);  
8     MPI_Allgather(&my_first_pattern, 1, MPI_INT, displs, 1, MPI_INT, MPI_COMM_WORLD);  
9  
10    MPI_Barrier( MPI_COMM_WORLD );  
11    // (4)  
12    MPI_Gatherv(pat_found + my_first_pattern, my_patterns, MPI_UNSIGNED_LONG,  
13                  pat_found, send_counts, displs, MPI_UNSIGNED_LONG, 0, MPI_COMM_WORLD);
```

Ogni Rank:

- Crea un array send\_counts per sapere quanti elementi arrivano da ogni processo ①.
- Crea un array displs per sapere dove posizionare i dati di ogni processo ②.
- Usa MPI\_Allgather per far sapere a tutti i processi quanti dati invierà ciascuno, questa azione è necessaria per utilizzare Gatherv che ha bisogno degli array send\_counts e displs completi per ogni Rank ③.
- MPI\_Gatherv raccoglie tutti i dati (pat\_found) nel processo Root e vengono ricomposti nell'ordine corretto usando le informazioni di send\_counts e displs all'interno di pat\_found del Root ④. Il pat\_found + my\_first\_pattern viene utilizzato per puntare al primo elemento assegnato al Rank

### 1.2.3 Gestione di pat\_matches

```
1 int total_pat_matches = 0;
2 MPI_Reduce(&pat_matches, &total_pat_matches, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
3 MPI_Barrier( MPI_COMM_WORLD );
4 if (rank == 0) {
5     pat_matches = total_pat_matches;
6 }
```

Ogni Rank invia al total\_pat\_matches del Root i propri pat\_matches effettuando una somma mediante MPI\_Reduce. Successivamente il Root assocerà al proprio pat\_matches la variabile total\_pat\_matches.

## 2 Implementazione CUDA

### 2.1 Introduzione

L'approccio impiegato per implementare Sequence in CUDA assegna un singolo pattern a ciascun thread, concentrando il calcolo principale all'interno del kernel dedicato alla fase di ricerca. I dati necessari sono caricati in memoria globale del Device e, al termine dell'esecuzione, sono trasferiti in memoria host per il calcolo del checksum. Durante l'esecuzione del kernel, ciascun thread verifica di dover effettivamente lavorare ed esegue operazioni atomiche per mantenere la coerenza dei risultati. Questo modello consente di parallelizzare gran parte della computazione, garantendo buone prestazioni. Tuttavia, la scalabilità è limitata dal numero massimo di thread supportato dalla GPU, che costituisce un vincolo hardware. Ad esempio nel caso del Cluster Sapienza la GPU utilizzata è una NVIDIA Quadro RTX 6000 Turing, questo significa che in linea teorica è possibile utilizzare 2,147,483,647 di blocchi ognuno con 1024 thread quindi 2.2 trilioni; con i suoi 4608 core CUDA, la scheda grafica può gestire un totale di circa 4.7 milioni di thread alla volta. Ciò è possibile solo in linea teorica poiché siamo limitati principalmente dalla Memoria che in questo caso ammonta a 24GigaByte.

### 2.2 Codice

L'host si occupa di generare la sequenza sequence, seq\_matches ecc...

Con cudaMalloc() e cudaMemcpy() sposta in memoria globale tutte le variabili necessarie per effettuare correttamente la ricerca delle sequenze.

Il kernel sequencer viene invocato con:

```
1 sequencer<<<ceil(pat_number/1024.0), 1024>>>(g_seq_length, g_pat_number, g_sequence,
2                                     d_pat_length, d_pattern, g_seq_matches, g_pat_matches, g_pat_found);
3
```

- ceil(pat\_number/1024) blocchi, in questo modo ogni thread si occuperà di un solo pattern, nel caso ci siano meno di 1024 pattern verrà utilizzato un solo blocco.
- 1024 thread per blocco, questo perché il massimo numero di thread per SM nell'architettura è 1024

Il kernel:

```
1 __global__ void sequencer(unsigned long *g_seq_length, int *g_pat_number, char *g_sequence, unsigned long *d_pat_length,
2                           char **d_pattern, int *g_seq_matches, int *g_pat_matches, unsigned long *g_pat_found) {
3     unsigned long start;
4     int pat;
5     unsigned long lind;
6
7     int tid = blockIdx.x * blockDim.x + threadIdx.x; // (1)
8     if (tid < *g_pat_number) {
9         pat = tid; // (2)
10        for( start=0; start <= *g_seq_length - d_pat_length[pat]; start++) {
11            for( lind=0; lind<d_pat_length[pat]; lind++) {
12                if ( g_sequence[start + lind] != d_pattern[pat][lind] ) break;
13            }
14            if ( lind == d_pat_length[pat] ) {
15                atomicAdd(g_pat_matches, 1); // (3)
16                atomicExch((unsigned long long*)&g_pat_found[pat], (unsigned long long)start); // (4)
17                break;
18            }
19        }
20        if ( g_pat_found[pat] != (unsigned long)NOT_FOUND ) {
21            increment_matches( pat, g_pat_found, d_pat_length, g_seq_matches );
22        }
23    }
24    __syncthreads();
25 }
26 __device__ void increment_matches( int pat, unsigned long *pat_found, unsigned long *pat_length, int *seq_matches ) {
27     unsigned long ind;
28     for( ind=0; ind<pat_length[pat]; ind++ ) {
29         atomicAdd(&seq_matches[ pat_found[pat] + ind ], 1); // (5)
30     }
31 }
```

Verifica il numero del thread e che sia uno di quelli che devono lavorare indipendentemente dal blocco ①, successivamente utilizza il suo indice come variabile pat in modo che coincida con l'indice del pattern con cui deve lavorare ②. È necessario l'utilizzo di operazioni atomiche:

- Somma per incrementare g\_pat\_matches (pat\_matches) ③.
- Scambio per assegnare start a g\_pat\_found[pat] ④.
- Somma per incrementare g\_seq\_matches (seq\_matches) all'interno della funzione increment\_matches ⑤. È stato anche necessario modificare questa funzione, è stato lasciato solo il corpo dell'else, poiché alla fine dell'esecuzione l'array seq\_matches presentava valori inferiori.

Dopo l'esecuzione del kernel vengono riportati in memoria host solo i dati necessari al checksum e viene liberata la memoria con cudaFree(). Si occuperà alla fine l'host di calcolare i checksum e stamparli.

## 3 Implementazione MPI + CUDA

### 3.1 Introduzione

L'approccio utilizzato per implementare Sequence in MPI+CUDA suddivide equamente il numero di pattern tra i Device, e ogni processo MPI gestisce una GPU. Anche in questo caso se la suddivisione non è intera, i pattern in eccesso vengono assegnati all'ultimo Rank/Device. Il codice non si limita all'utilizzo di due GPU, ma può essere eseguito su un numero maggiore di dispositivi. Ogni Processo alloca localmente i dati necessari ai calcoli e li invia al proprio Device, dove il kernel, in modo simile all'implementazione CUDA, esegue la ricerca sui dati esclusivamente sulla porzione decisa durante la spartizione.

Al termine, come nella versione MPI, i dati per il checksum vengono trasferiti nella memoria host di ciascun rank, e il processo Root unifica i risultati di ogni Processo nelle proprie variabili locali con le stesse modalità discusse per l'implementazione MPI. Siamo a conoscenza che per quanto riguarda la memoria non è molto efficiente allocare ad ogni GPU ad esempio tutto l'array `d_path_length` poiché lavorerà solo su una parte di esso.

### 3.2 Codice

La versione MPI+CUDA estende il codice base della versione CUDA aggiungendo le funzioni di inizializzazione MPI all'inizio del main e utilizzando la variabile rank come argomento di `cudaSetDevice()`.

Solo il Root genera la sequenza e la trasmette a tutti i processi con Bcast.

Nel medesimo modo della versione MPI ogni processo:

- Alloca copie locali delle variabili.
- Calcola quanti pattern elaborare (se è l'ultimo rank include quelle rimanenti).
- Trasferisce i dati sul proprio Device con `cudaMalloc` e `cudaMemcpy`.

```
1 __global__ void sequencer(unsigned long *g_seq_length, int *g_my_pat_number, char *g_sequence, unsigned long *d_pat_length,
2                           char **d_pattern, int *g_seq_matches, int *g_pat_matches, unsigned long *g_pat_found, int *g_my_first_pattern) {
3     unsigned long start;
4     int pat;
5     unsigned long lind;
6
7     int tid = blockIdx.x * blockDim.x + threadIdx.x;
8     if (tid < *g_my_pat_number) { // (1)
9         pat = tid + *g_my_first_pattern; // (2)
10        for( start = 0; start <= *g_seq_length - d_pat_length[pat]; start++) {
11            for( lind=0; lind < d_pat_length[pat]; lind++) {
12                if ( g_sequence[start + lind] != d_pattern[pat][lind] ) break;
13            }
14            if ( lind == d_pat_length[pat] ) {
15                atomicAdd(g_pat_matches, 1);
16                atomicExch((unsigned long long*)&g_pat_found[pat], (unsigned long long)start);
17                break;
18            }
19        }
20        if ( g_pat_found[pat] != (unsigned long)NOT_FOUND ) {
21            increment_matches( pat, g_pat_found, d_pat_length, g_seq_matches );
22        }
23    }
24 }
```

- Invoca il kernel `sequencer`, che si differenzia dalla versione CUDA, passando `g_my_pat_number` e `g_my_first_pattern`, così ciascun thread lavora sui pattern corrispondenti al proprio intervallo. Viene effettuata una verifica per far lavorare solo il numero di thread necessari ① e per allinearsi al numero di pattern corretto vengono sommati il numero di rank e `g_my_first_pattern` ②.

Se ad esempio ad ogni Device sono assegnati 100 pattern, il thread 5 del secondo Device lavora sul pattern  $5 + 100 = 105$  poiché i primi 100 pattern spettano al primo.

Al termine, i dati per il checksum sono copiati nell'host e la memoria del Device viene liberata. Il processo di unificazione e calcolo dei checksum è identico alla versione MPI.

## 4 Implementazione OpenMP

In questa implementazione il blocco di codice che è stato parallelizzato è quello della ricerca dei pattern.

Per definire il blocco di codice è stata utilizzata la direttiva pragma per condividere tra tutti i thread gli array pat\_found e pat\_matches ①.

Il primo ciclo for è stato parallelizzato poiché, nonostante i nostri tentativi, non siamo riusciti a parallelizzare anche i cicli annidati. Per utilizzare la clausola collapse() è necessario che i cicli siano perfettamente annidati senza istruzioni intermedie e non

contengano break che interrompono i cicli. Inoltre, i domini degli ultimi due for si intersecano poiché la variabile pat viene utilizzata come condizione sia per questi cicli che come variabile di controllo del primo for. Questa dipendenza impedisce l'applicazione efficace della clausola collapse(), che richiede domini indipendenti e senza interferenze.

Assieme all'omp for sono state rese private le variabili di controllo di ogni ciclo for, è stata utilizzata una reduction per effettuare la somma di pat\_matches alla fine del blocco parallelo poiché ogni thread calcola un determinato numero di pattern e incrementa privatamente quando ne trova uno ②.

Lo schedule utilizzato è dynamic poiché nei test le sequenze hanno lunghezza variabile, di conseguenza ogni thread potrebbe concludere il lavoro con tempistiche diverse.

In conclusione mediante la sezione critica si protegge l'accesso concorrente alla funzione increment\_matches, che si occupa di modificare seq\_matches, la quale è fondamentale per evitare che nell'array ci siano valori inferiori alla fine dell'esecuzione ③.

```
1 unsigned long start;
2 int pat;
3 #pragma omp parallel shared(pat_found, seq_matches) // (1)
4 {
5     // (2)
6     #pragma omp for private(start, pat, lind) \
7         reduction(+:pat_matches) schedule(dynamic,4)
8
9     for( pat=0; pat < pat_number; pat++ ) {
10         for( start=0; start <= seq_length - pat_length[pat]; start++ ) {
11             for( lind=0; lind<pat_length[pat]; lind++ ) {
12                 if ( sequence[start + lind] != pattern[pat][lind] ) break;
13             }
14             if ( lind == pat_length[pat] ) {
15                 pat_matches++;
16                 pat_found[pat] = start;
17                 break;
18             }
19         }
20         if ( pat_found[pat] != (unsigned long)NOT_FOUND ) {
21             #pragma omp critical // (3)
22             increment_matches( pat, pat_found, pat_length, seq_matches );
23         }
24     }
25 }
```

## 5 Testing Efficienza e Limitazioni

### 5.1 Introduzione

Per l'esecuzione dei test è stato realizzato uno script che ripete l'esecuzione di tutte le versioni del programma, con diversi argomenti in input, in modo da verificare diverse situazioni con complessità e quantità di dati crescenti.

Le tipologie di test effettuati sono i seguenti:

1. Test dove aumenta la lunghezza della sequenza
2. Test dove aumenta il numero di pattern da cercare
3. Test dove aumentano la lunghezza della sequenza, il numero di pattern e la lunghezza dei pattern stessi.

Non verranno mostrati test dove aumenta solo la grandezza dei pattern, perché nella nostra implementazione il parallelismo non scala con l'aumentare di tale grandezza (i tempi rimangono sempre uguali)

I test di OpenMP sono stati effettuati su una macchina diversa dal Cluster Sapienza con CPU AMD Ryzen 5 7535HS with Radeon Graphics, 3301 Mhz, 6 core, 12 processori logici.

#### 5.1.1 MPI

Con input di piccola dimensione, l'overhead incide pesantemente e riduce le performance del programma rispetto alla versione sequenziale, ma con input di media e grande dimensione, i costi di comunicazione vengono compensati dalla maggiore parallelizzazione, e quindi la velocità aumenta e cresce linearmente all'aumentare del numero dei processi.

Questo è grazie al fatto che la parallelizzazione con MPI risulta ottimale per questo tipo di programma, in quanto i pattern da cercare possono essere suddivisi in maniera equa ai vari processi, ed essi cercheranno tali pattern in maniera completamente indipendente l'uno dall'altro.

La comunicazione (overhead) avviene soltanto all'inizio e alla fine dell'esecuzione

#### 5.1.2 OPENMP

OpenMP risulta più efficiente del sequenziale già con input piccoli, perché a differenza di MPI, non viene introdotto un significativo overhead di comunicazione (OpenMP è basato su parallelismo a memoria condivisa), ma aumentando la quantità di dati, si verificano più frequentemente conflitti sulla memoria, contrastando l'efficacia del parallelismo, cosa che non avviene in MPI perché i dati vengono processati in maniera indipendente.

#### 4.1.3 CUDA

Il test dove ci sono tantissimi pattern, CUDA è molto efficiente per 2 motivi principali:

1. tanti pattern da cercare corrispondono a tanti thread in esecuzione sulla gpu, quindi viene sfruttato l'alto parallelismo della gpu (occupancy).
2. I thread quando cercano I pattern nelle sequenze, fanno tanti accessi in memoria, se un warp va in attesa per l'accesso in memoria lo scheduler assegna nuovi thread in modo che gli SM siano sempre occupati.

Invece quando i pattern sono pochi, l'occupancy della gpu è molto bassa, quindi non andiamo a sfruttare il parallelismo della gpu. Questo perché tutti i threads vengono assegnati a pochi SM, quindi la GPU non viene utilizzata nella sua completezza.

Se concettualmente andassimo a distribuire questi pochi pattern su tutti gli SM, essi sarebbero comunque per la maggior parte del tempo non attivi, a causa dell'attesa degli accessi in memoria, durante I quali lo scheduler non ha altri thread da sostituire.

In questo caso le CPU sono migliori, in quanto siccome in un processore (core) viene eseguito un thread alla volta, il processore avrà sicuramente un thread da sostituire, quindi meno tempo di attesa, anche se più overhead. Inoltre con pattern molto lunghi, un singolo thread eseguirà molti più accessi in memoria, e avrà bisogno di molta più memoria cache, cosa che le classiche CPU hanno di più rispetto alle GPU (cache L1).

### 5.1.4 MPI + CUDA

Le stesse cose si applicano ovviamente anche a MPI + CUDA, in quanto la computazione viene effettuata sempre dalle GPU. Infatti il vantaggio di usare due GPU in parallelo si vede solo in quei test che hanno tantissimi pattern da ricercare. Altrimenti il vantaggio di avere due GPU in parallelo rispetto alla versione CUDA normale viene meno per 2 motivi: per l'overhead introdotto da MPI, e soprattutto perché, siccome ogni GPU avrà la metà dei pattern da verificare, ovvero la metà dei thread da eseguire, l'occupancy diminuisce, soprattutto quando i pattern in partenza sono pochi, per gli stessi motivi precedenti.

## 5.2 Fase di test

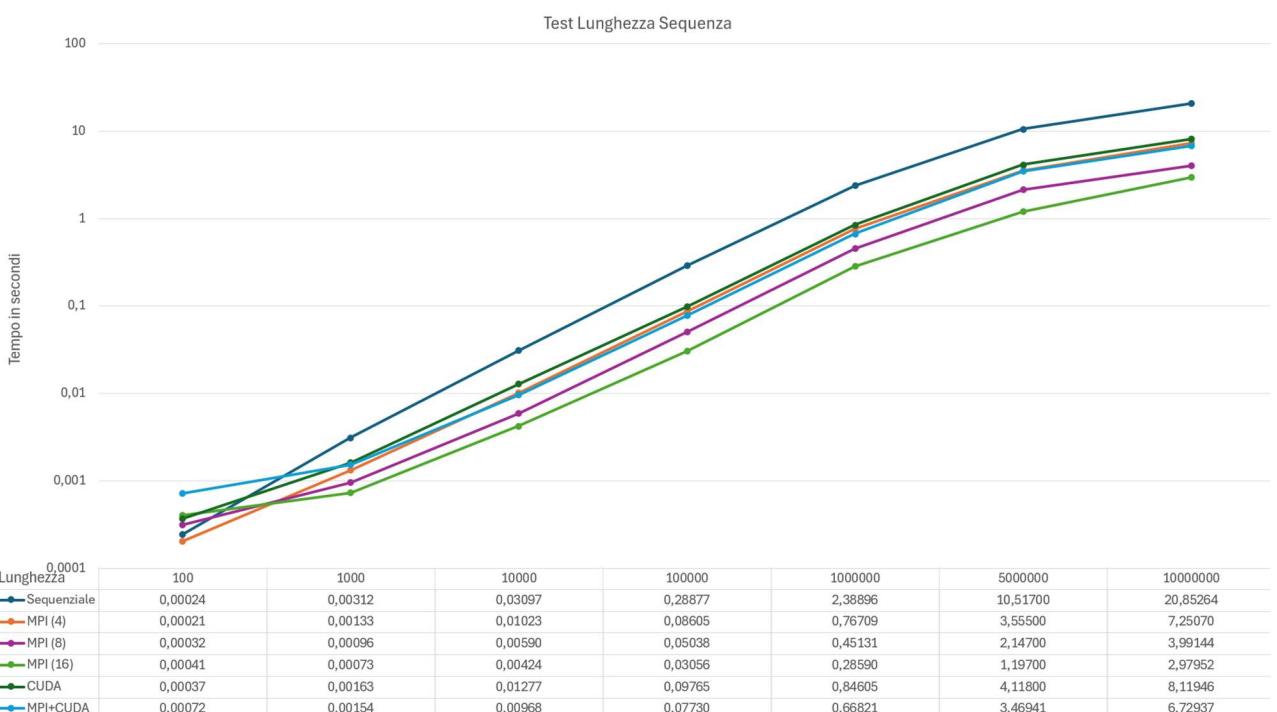
I test di OpenMP sono stati effettuati su una macchina diversa dal Cluster Sapienza con CPU AMD Ryzen 5 7535HS with Radeon Graphics, 3301 Mhz, 6 core, 12 processori logici.

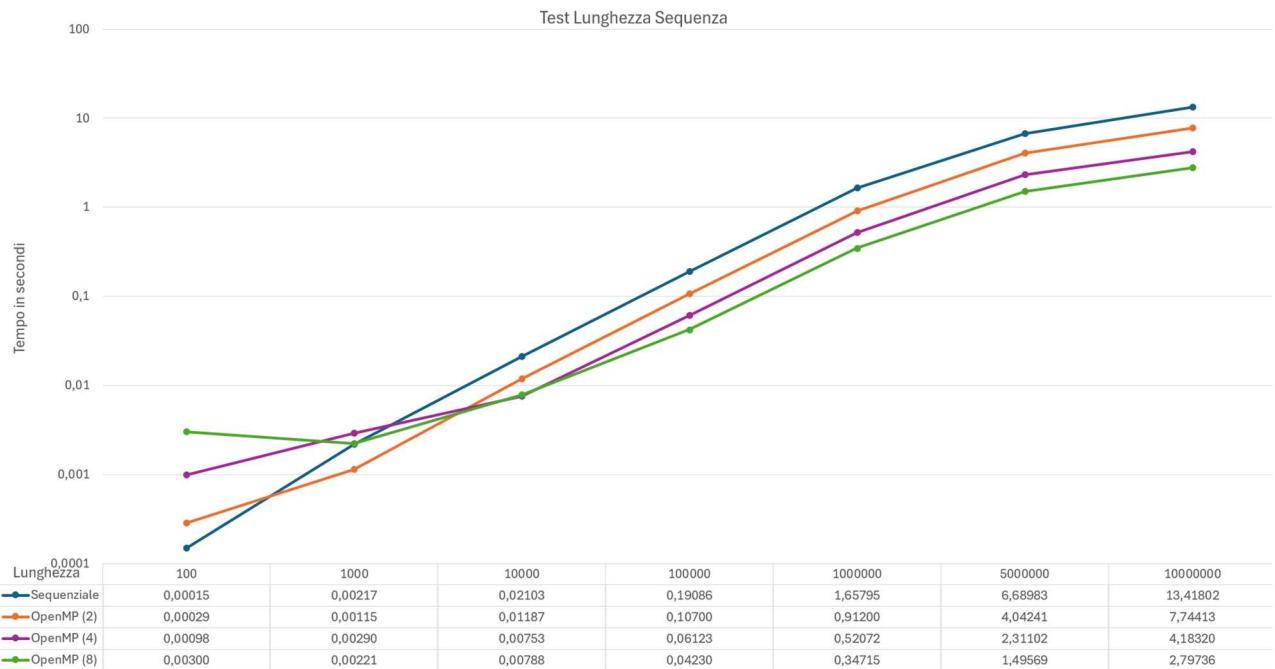
Inoltre ogni dato presente nelle seguenti tabelle è stato ottenuto effettuando la media dei risultati del test ripetuto per 5 volte.

### 5.2.1 Test lunghezza sequenza

In questo test, le versioni parallele del codice nei test più piccoli sono più inefficienti della versione sequenziale del programma. Dal secondo test MPI e OpenMP diventano più efficienti e scalano con l'aumentare del numero di processi/threads (OpenMP lo fa inizialmente soltanto con 4 threads). In media MPI(16) è del 86,8% più efficiente del sequenziale e OpenMP(32) del 78,6%.

CUDA e MPI+CUDA risultano sempre peggiori in efficienza rispetto alle controparti MPI e OpenMP rispettivamente rispetto al sequenziale il 61,3% e 67,9%.

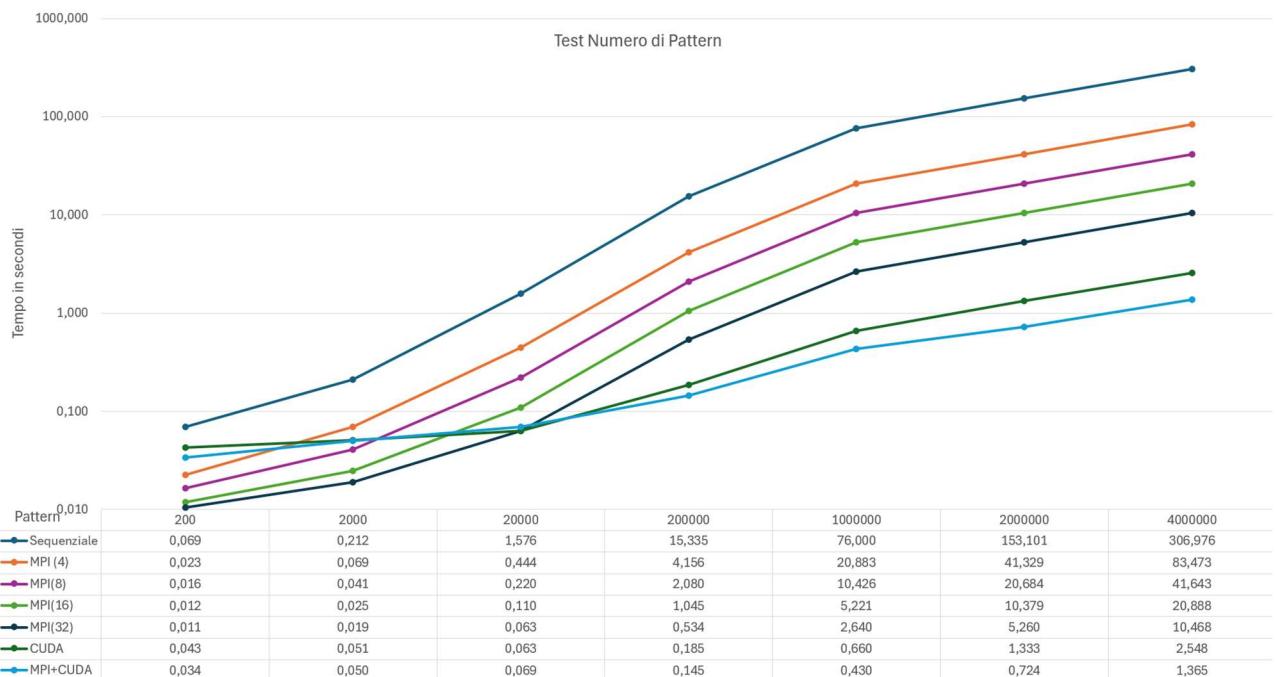


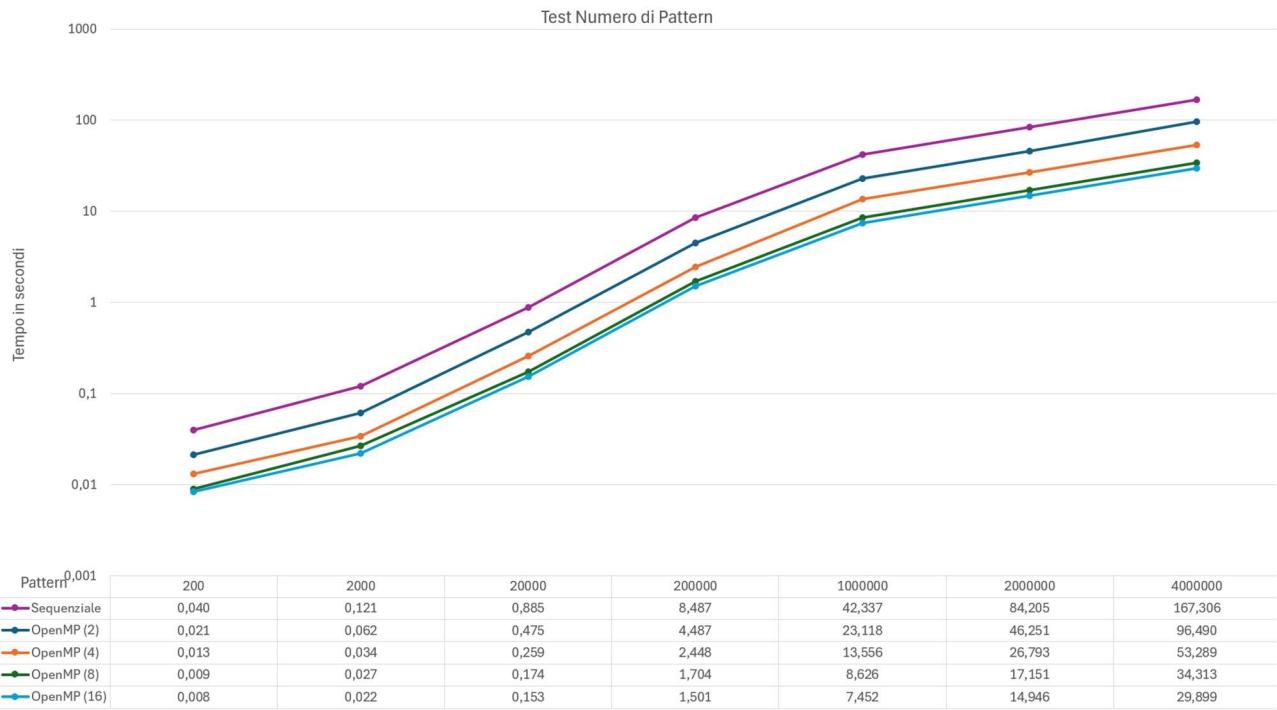


#### 4.2.2 Test numero pattern

In questo test MPI e OpenMP si mostrano subito più efficienti del sequenziale, e i tempi di esecuzione diminuiscono linearmente all'aumentare del numero di processi/threads in esecuzione. In media MPI(32) è del 96,6% più efficiente del sequenziale e OpenMP(32) del 82,2%.

CUDA e MPI+CUDA inizialmente sono più lenti di MPI e OpenMP, ma all'aumentare del numero di pattern in input, la velocità di esecuzione diventa superiore rispetto a tutte le altre versioni, con MPI+CUDA che raggiunge la metà del tempo di esecuzione rispetto a CUDA nei test più pesanti, mostrando un'eccellente scalabilità. In media rispetto al sequenziale CUDA ha una percentuale di miglioramento del 99,1% e MPI+CUDA del 99,5% rispetto al sequenziale.





### 4.3.2 Test Generale

Il seguente test è stato scelto per studiare il comportamento dei programmi in casi generici, unendo le caratteristiche dei test precedenti. Si può osservare che con gli input più piccoli le versioni parallele sono più inefficienti della versione sequenziale, con OpenMP a due threads che risulta un caso a parte essendo fin da subito più veloce. Nei test successivi MPI e OpenMP scalano linearmente con una percentuale di miglioramento rispettivamente del 95,5% e 83,7%. CUDA e MPI+CUDA diventano i più veloci rispettivamente dal test 4 e dal test 5, dove si può notare che in alcuni test, le performance non scalano linearmente, con CUDA che risulta molto più efficiente nel test 4 e MPI+CUDA nei test 5 e 6. In media abbiamo un miglioramento del 96,3% per CUDA e del 98,2% per MPI+CUDA.

