Progetto Sistemi Multicore

Cola Valerio & Federico Cerboni

15 Gennaio 2025

**1 Implementazione MPI**

* 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 al numero e lunghezza dei pattern.

* 1. **Codice**

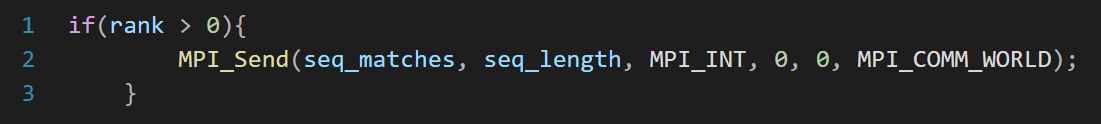
Viene generata la sequenza “sequence” solo dal Root e inviata a tutti gli altri Rank con la funzione MPI\_Bcast.

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

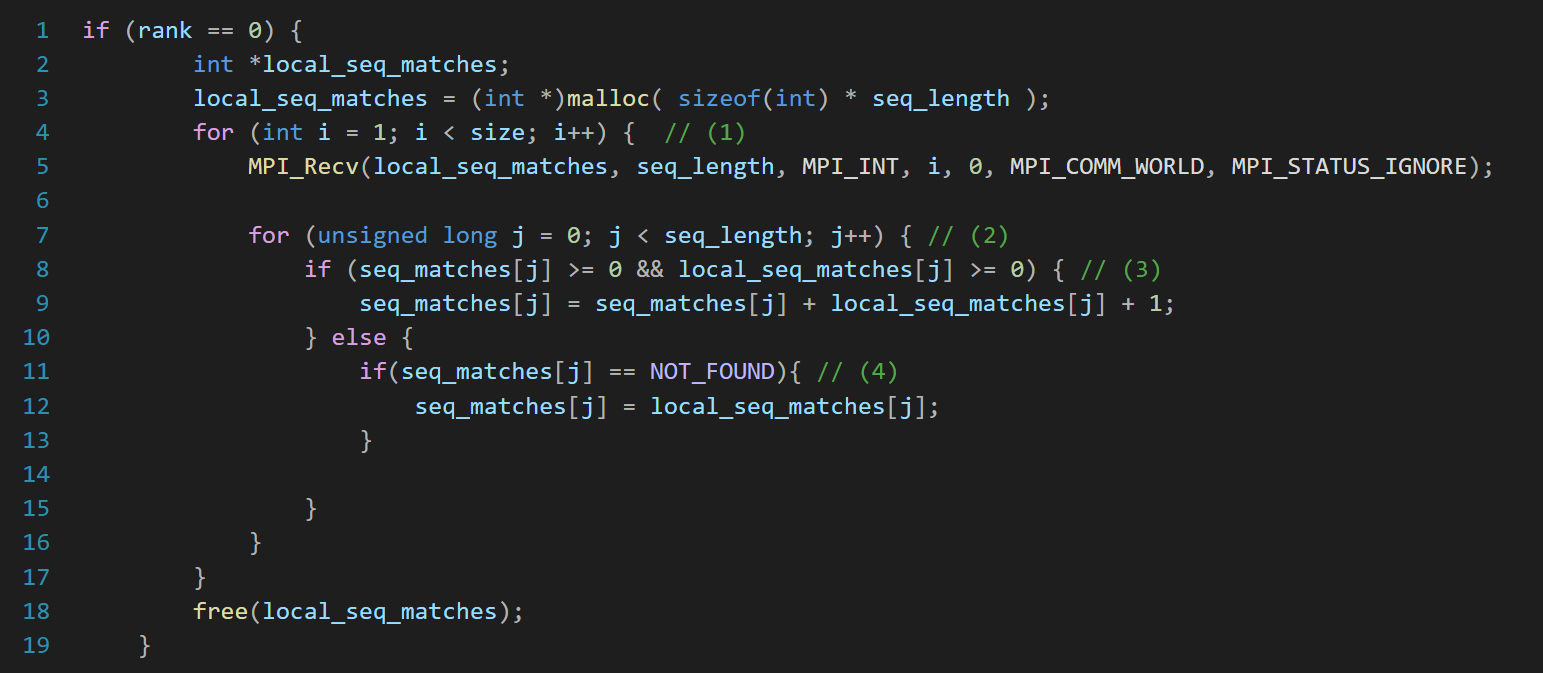
Alla fine dell'esecuzione, il Root si occuperà del calcolo e la stampa dei checksum, unificando i dati delle variabili locali seq\_matches, pat\_found e pat\_matches di ogni rank.

**1.2.1 Gestione di seq\_matches**

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

**Problema:**

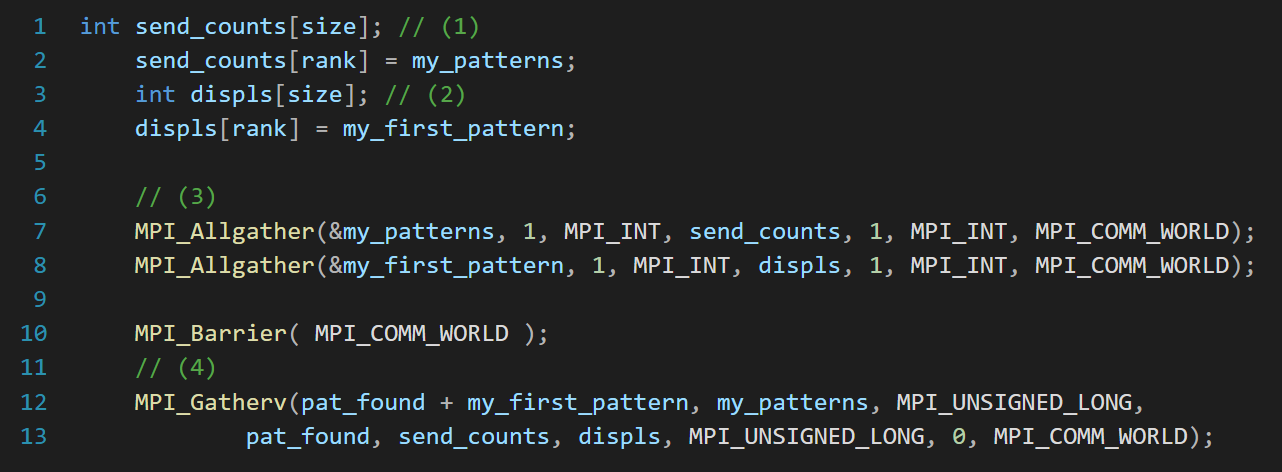
“Ogni Rank parte con ogni cella di seq\_matches = -1. Se quindi un pattern che si ripete tra più Rank o se più Rank trovano più pattern che attraversano la stessa cella , seq\_matches passerà da -1 a 0 più volte, creando discrepanze rispetto all’esecuzione sequenziale nella quale per ogni posizione dell’array 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 celle 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.

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**:

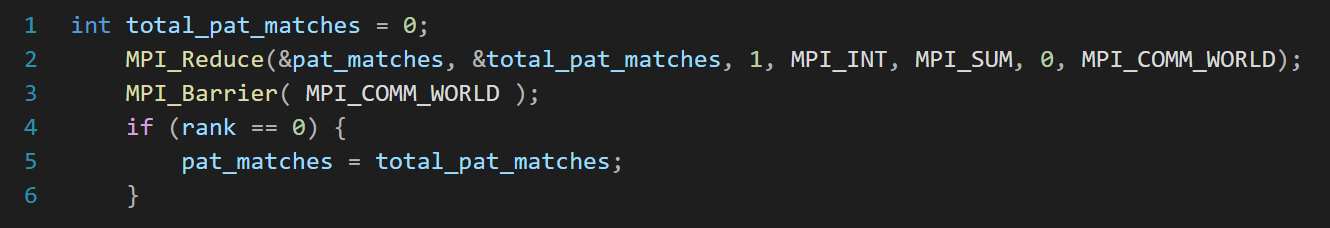
* Se entrambi i valori sono ≥ 0, si aggiunge 1 oltre alla somma **③**.
* Se il seq\_matches del Root è NOT\_FOUND, si imposta al seq\_matches il valore di local\_seq\_matches **④**.

**1.2.2 Gestione di pat\_found**

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**

****

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**

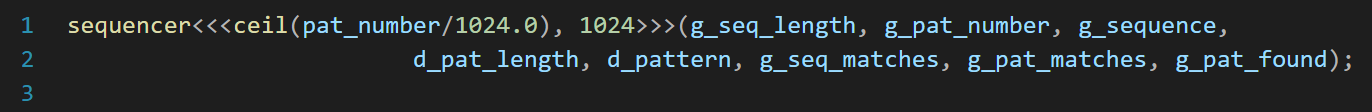
**2.1 Introduzione**

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 circa 2.2 trilioni; con i suoi 4608 core CUDA distribuiti su 72 SM la scheda grafica può gestire un totale di circa 147,356 di thread alla volta. Di conseguenza è possibile invocare un kernel con un quantitativo elevato di thread ma la GPU li gestirà schedulandoli e il quantitativo di dati sarà comunque limitato dalle risorse (registri, memoria condivisa, capacità degli SM ecc...).

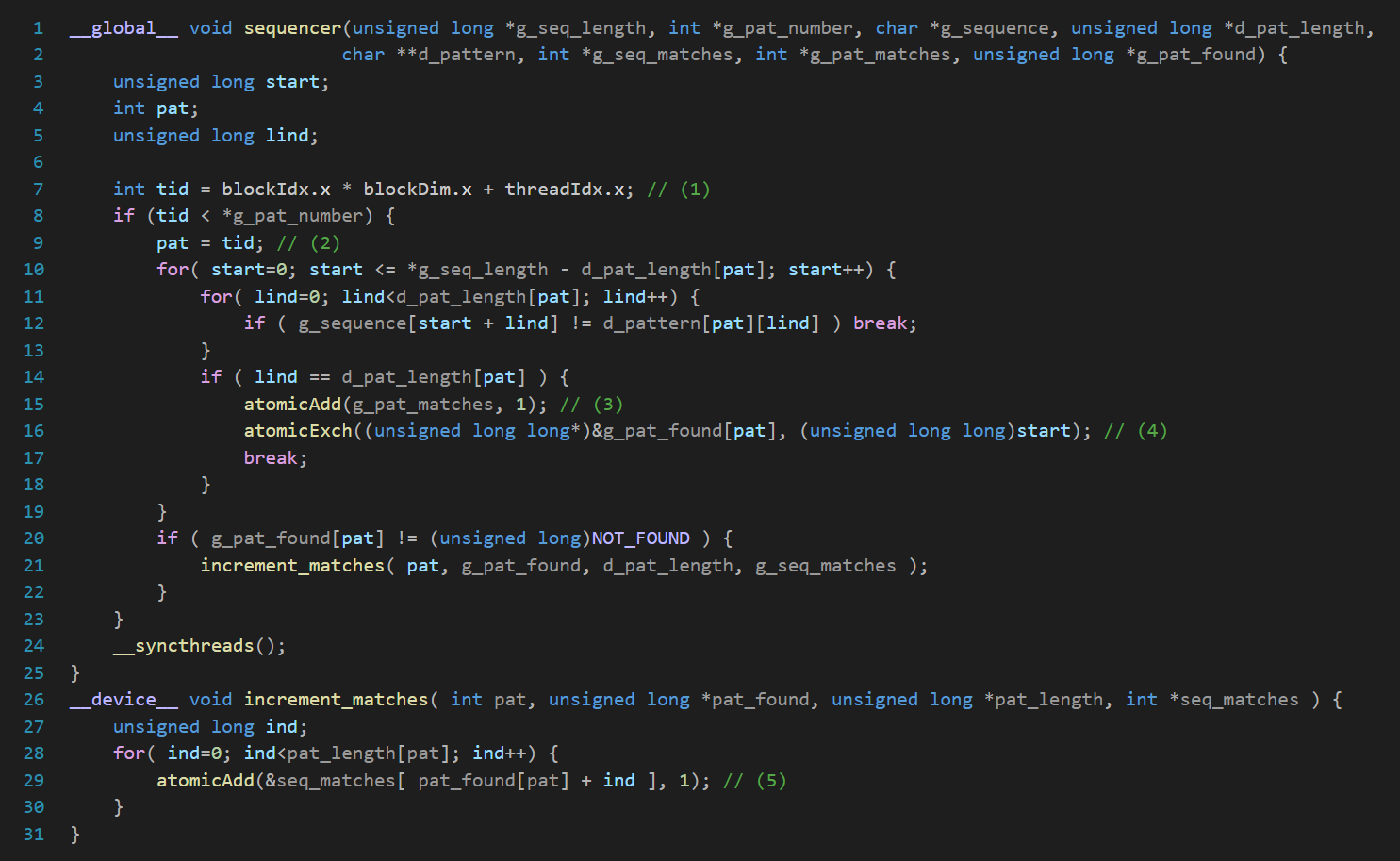
* 1. **Codice**

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:

* 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:

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, poichè 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 poichè lavorerà solo su una parte di esso.

**3.2 Codice**

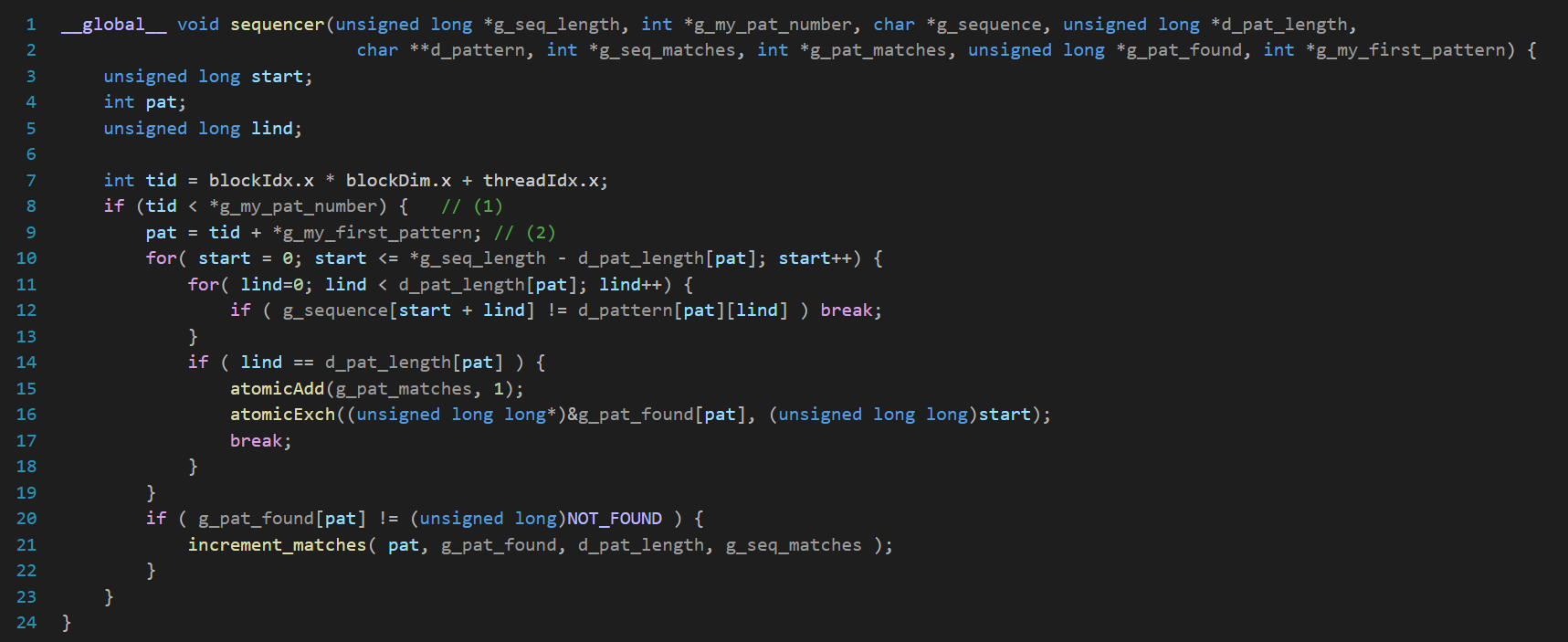
La versione MPI+CUDA estende il codice base della versione CUDA aggiungendo le funzioni di inizializzazione MPI all’inizio del main e utilizzando la variabile rank come argomento di cudaSetDevice().

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.

• 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.

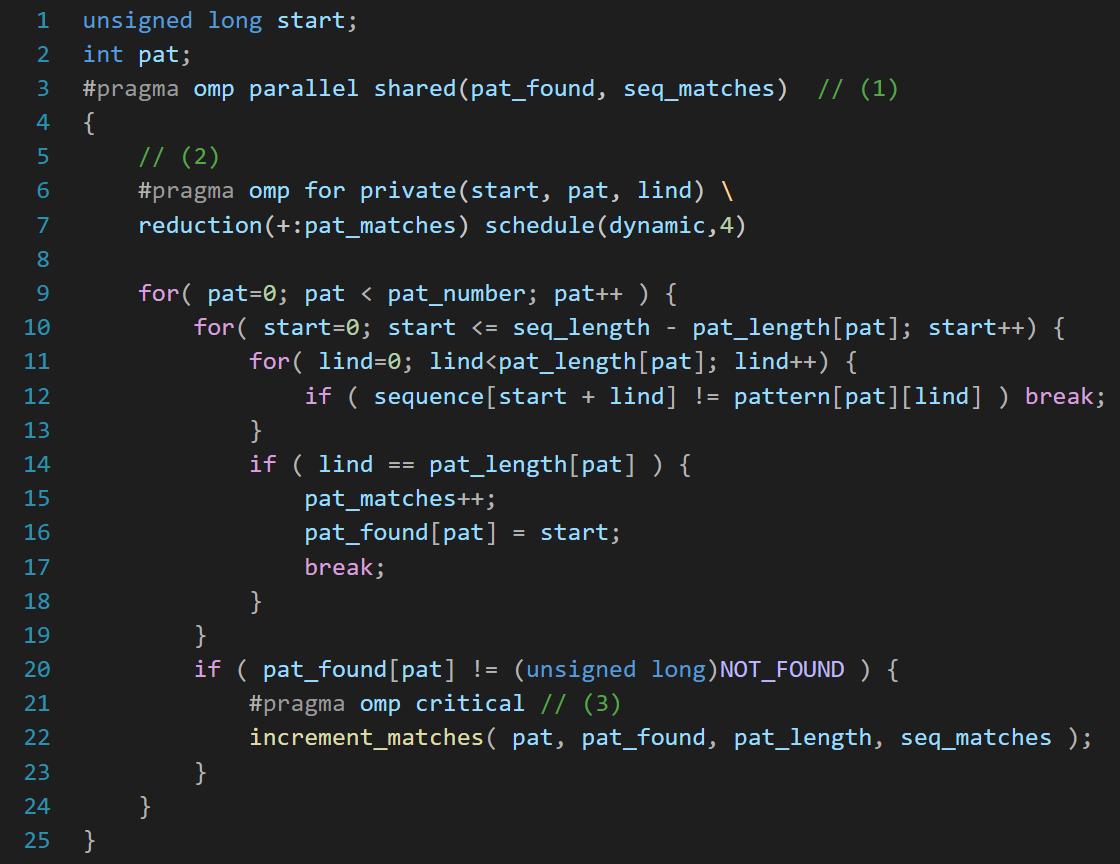
Viene effettuata una verifica per far lavorare solo il numero di thread necessari **①** e per allinearsi al numero di pattern corretto vengono sommati il numero di thread e g\_my\_first\_pattern **②**.

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**.

**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)

**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 in generale, confrontandolo con MPI a parità di processi\threads, risulta sempre più lento di quest’ ultimo nei test più piccoli, ma con input più grandi, tranne nei test del numero di pattern, diventa sempre più veloce. Questo è dovuto al fatto che OpenMP si basa sul paradigma di parallelizzazione a memoria condivisa, quindi a differenza di MPI, non introduce un significativo overhead di comunicazione. OpenMP mostra i suoi punti deboli solamente nei casi in cui si verificano molti conflitti di memoria, ovvero quando gli input dei test sono piccoli, e anche nel caso di un numero di pattern molto alto da cercare.

**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 memora 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.

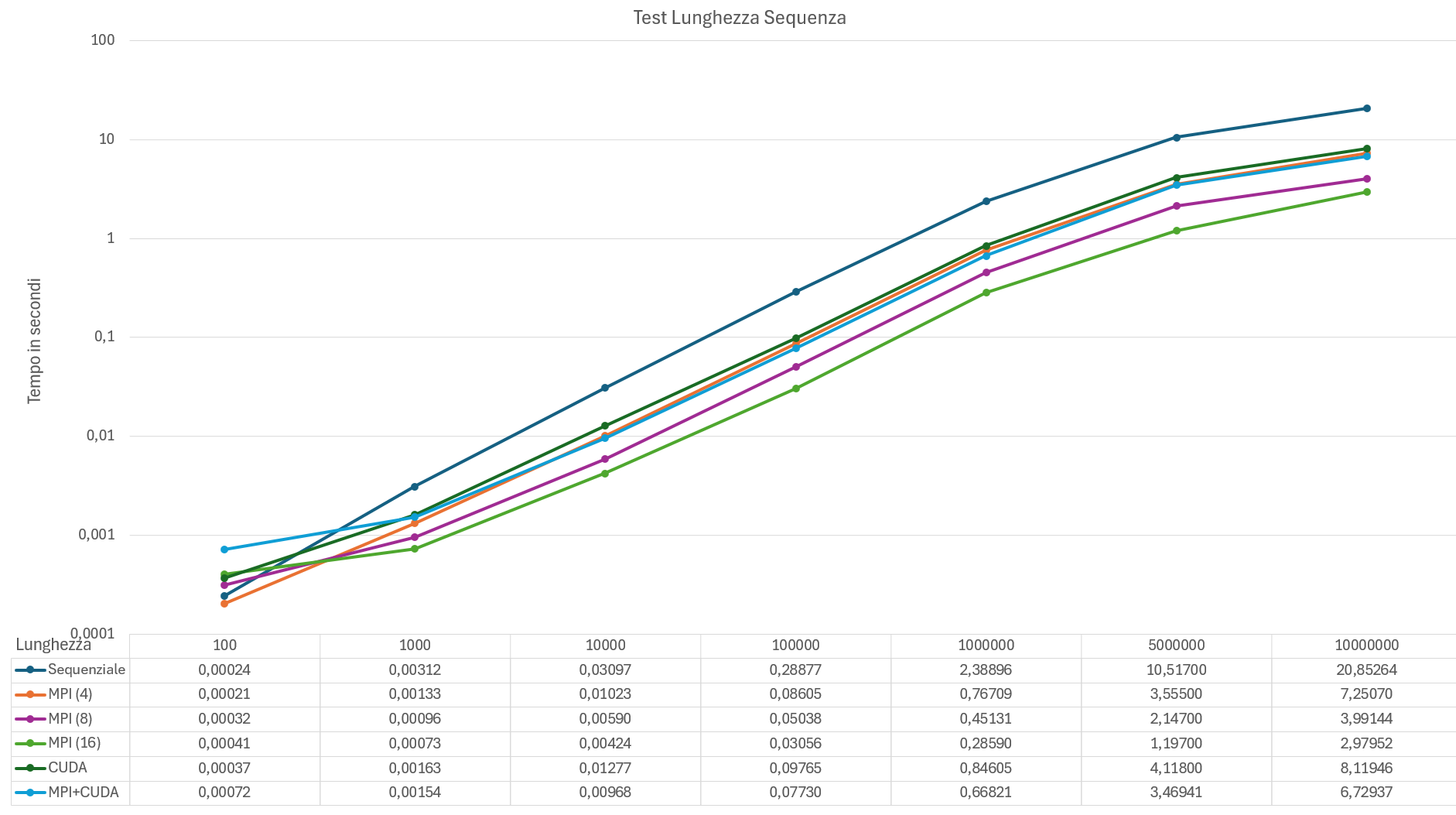
**5.2 Fase di test**

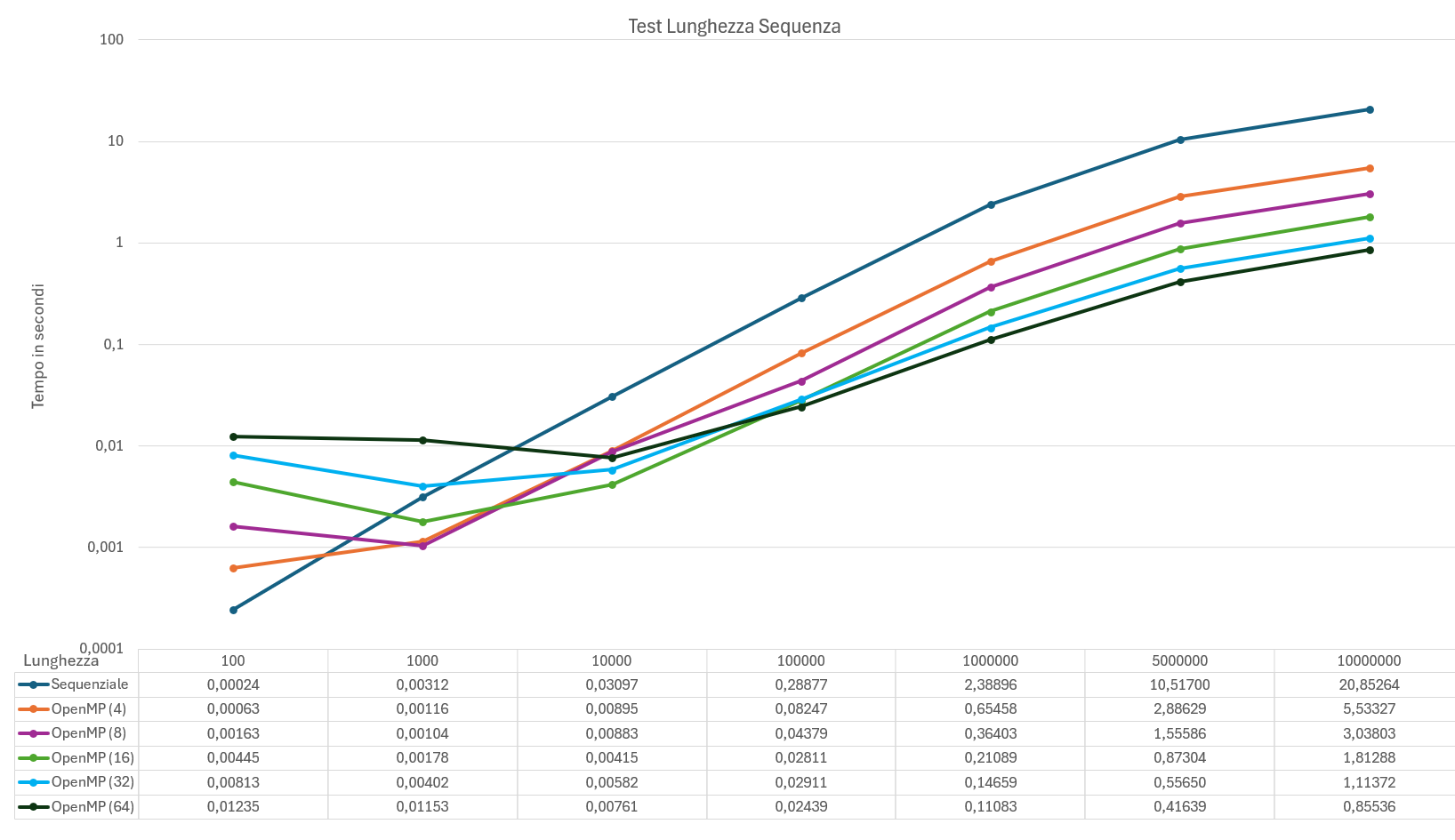
Inoltre ogni dato presente nelle seguenti tabelle è stato ottenuto effettuando la media dei risultati del test ripetuto per 5 volte.

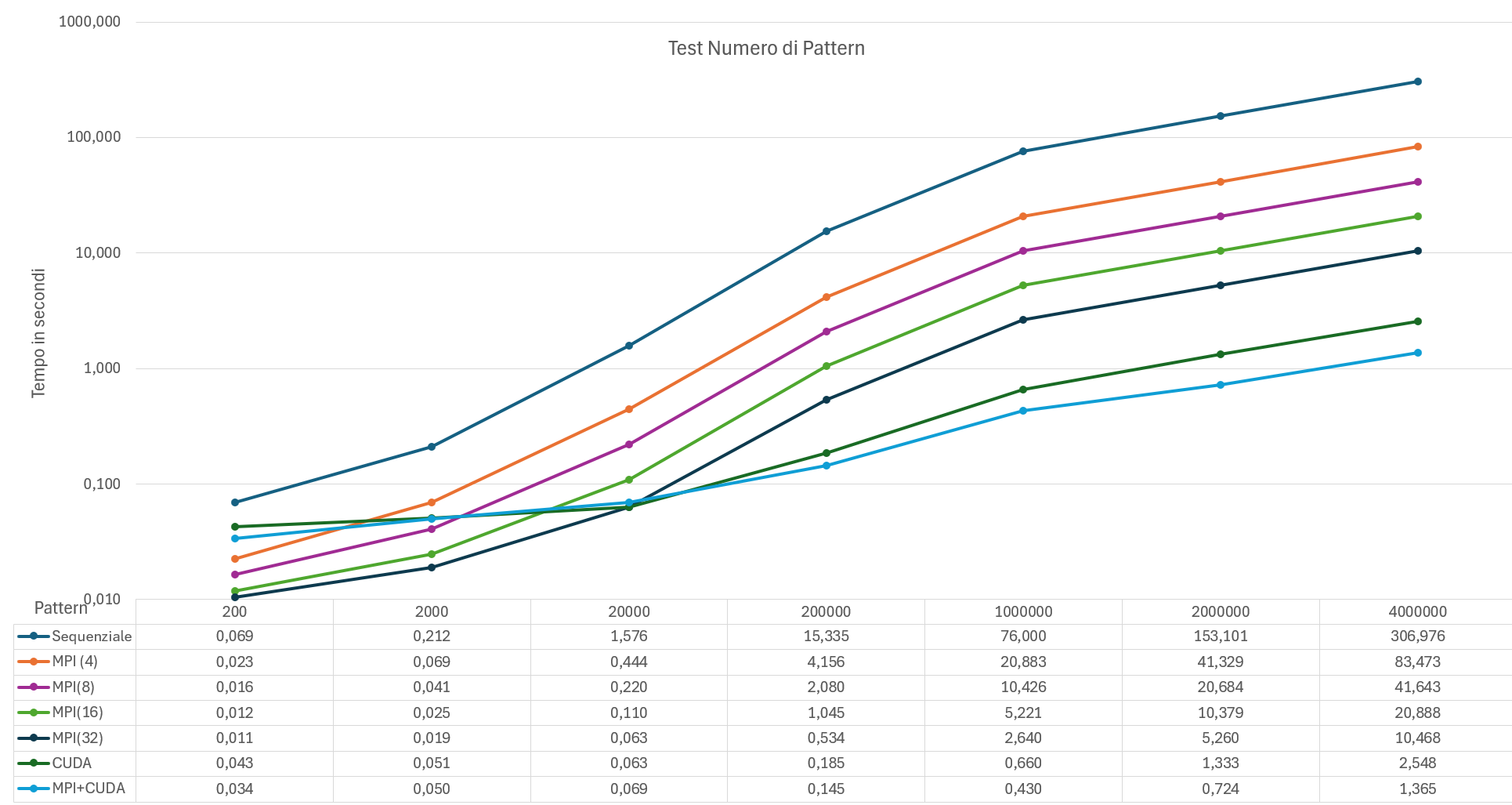
**5.2.1 Test lunghezza sequenza DNA**

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. In media MPI(16) è del 86,8% più efficiente del sequenziale e OpenMP(32) del 94.7%. MPI con 32 processi risulta sempre peggiore rispetto alla versione con 16 processi, mostrando i sintomi di un overhead di comunicazione eccessivo. MPI inizialmente (test 2 e 3) risulta più veloce di OpenMP, ma nei test più grandi quest’ ultimo diventa più efficiente.

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

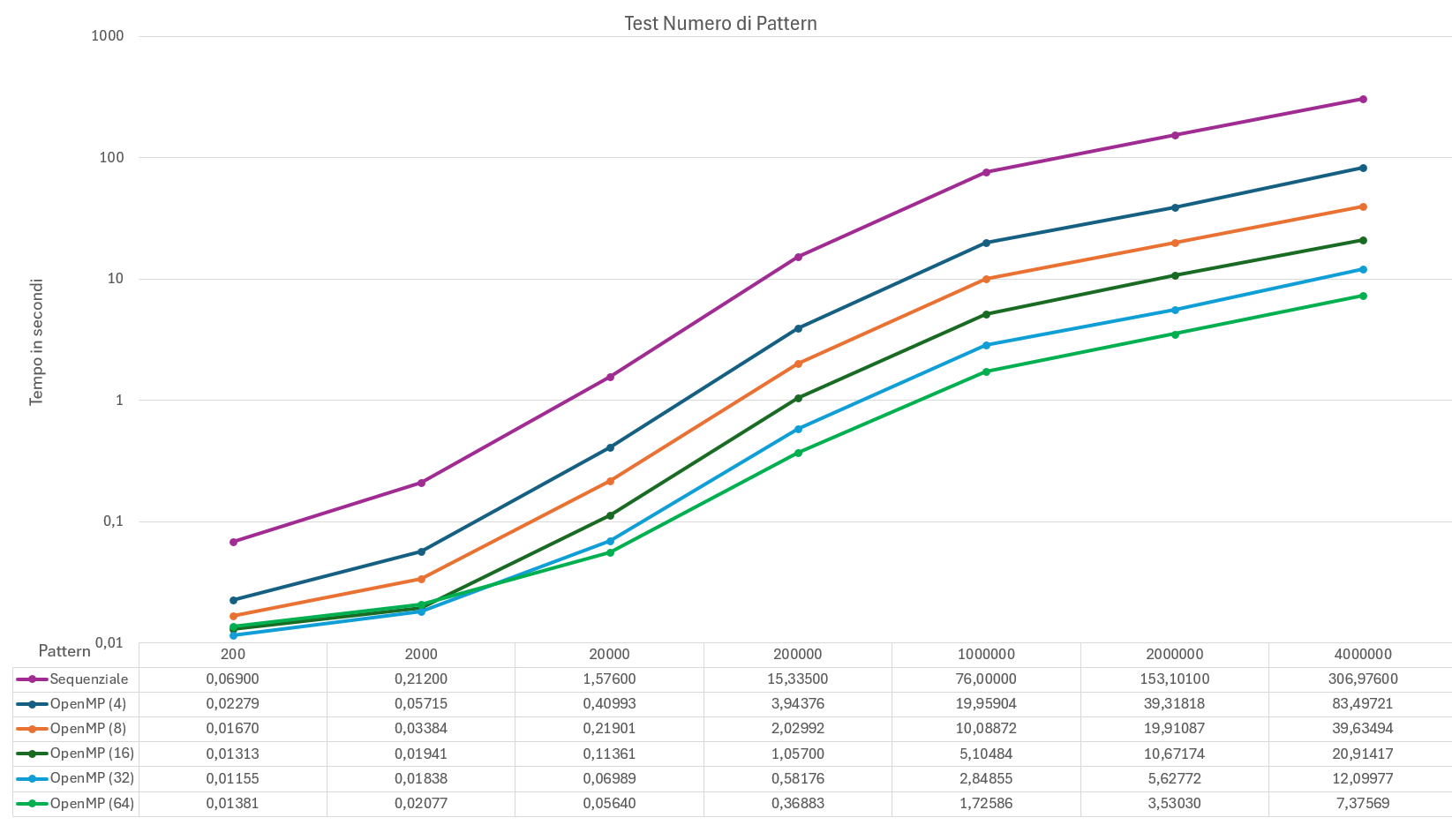


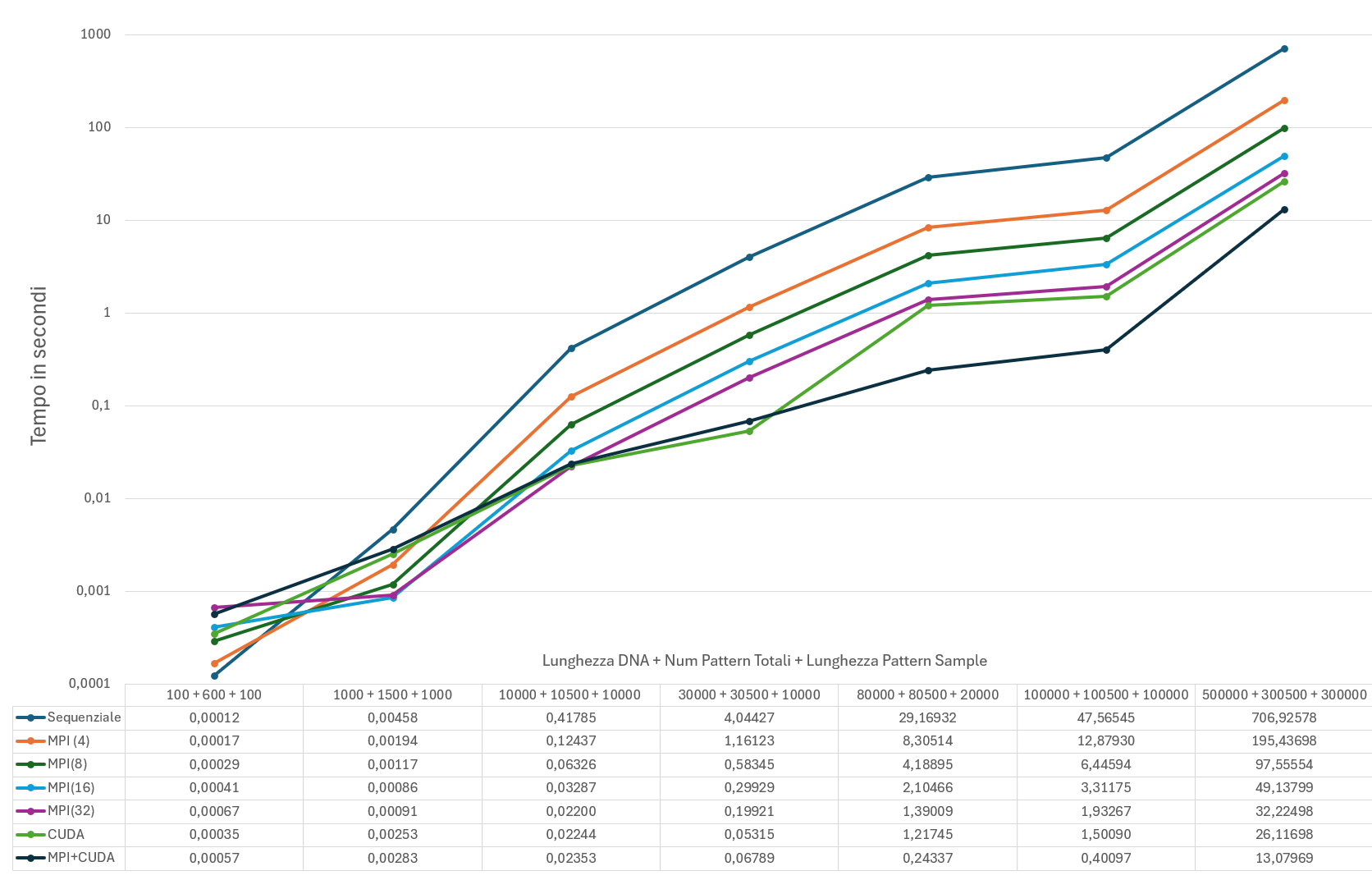
****

**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(64) del 96%. Nei primi 3 test i tempi di esecuzione di MPI e OpenMP sono simili, ma aumentando la grandezza di input, MPI scala in maniera quasi perfettamente lineare, a differenza di OpenMP, soprattutto nei test con un grande numero di thread.

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 meda 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 Complessi**

Il seguente test è stato scelto per studiare il comportamento dei programmi in casi complessi, unendo le caratteristiche dei test precedenti. Si può osservare che con gli input più piccoli le versioni parallele sono più inefficienti della versione sequenziale. Nei test successivi MPI e OpenMP scalano linearmente, con una percentuale di miglioramento rispettivamente del 95,5% e 96.2%. Confrontando MPI e OpenMP si può notare che si comportano allo stesso modo del test lunghezza sequenza, con MPI inizialmente più veloce ma con OpenMP che prevale nei test più complessi. 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.

