

DNA Sequence Alignment

Implementazione MPI OpenMP CUDA
MPI+CUDA



SAPIENZA
UNIVERSITÀ DI ROMA

DNA Sequence Alignment

Implementazione MPI



SAPIENZA
UNIVERSITÀ DI ROMA

```

1 // Initialize MPI size
2     int size;
3     MPI_Comm_size( MPI_COMM_WORLD, &size );
4
5 /* 2.1. Allocate and fill sequence */
6     char *sequence = (char *)malloc( sizeof(char) * seq_length );
7     if ( sequence == NULL ) {
8         fprintf(stderr,"\n-- Error allocating the sequence for size: %lu\n", seq_length );
9         MPI_Abort( MPI_COMM_WORLD, EXIT_FAILURE );
10    }
11
12 // Only rank 0 generates the sequence and sends it to all other processes
13 if(rank == 0){
14     random = rng_new( seed );
15     generate_rng_sequence( &random, prob_G, prob_C, prob_A, sequence, seq_length );
16 }
17 MPI_Bcast(sequence, seq_length, MPI_CHAR, 0, MPI_COMM_WORLD);
18 MPI_Barrier( MPI_COMM_WORLD );
19
20 /* 2.3.2. Other results related to the main sequence */
21 // Each rank allocate its own seq_matches
22 int *seq_matches;
23 seq_matches = (int *)malloc( sizeof(int) * seq_length );
24 if ( seq_matches == NULL ) {
25     fprintf(stderr,"\n-- Error allocating aux sequence structures for size: %lu\n", seq_length );
26     MPI_Abort( MPI_COMM_WORLD, EXIT_FAILURE );
27 }
28
29 /* 4. Initialize ancillary structures */
30 // Each rank allocate its own pat_found and initialize it to NOT_FOUND
31 for( ind=0; ind<pat_number; ind++) {
32     pat_found[ind] = (unsigned long)NOT_FOUND;
33 }
34 // Each rank allocate its own seq_matches and initialize it to NOT_FOUND
35 for( lind=0; lind<seq_length; lind++) {
36     seq_matches[lind] = NOT_FOUND;
37 }
38
39 // Each rank takes a number, the start and the last patterns to search
40 int my_patterns = pat_number/size;
41 int remainder = pat_number % size;
42 int my_first_pattern = rank * my_patterns;
43
44 if(rank == size - 1){
45     my_patterns += remainder;
46 }
47
48 int my_last_pattern = my_first_pattern + my_patterns -1;
49 MPI_Barrier( MPI_COMM_WORLD );
50

```

Ogni Rank:

- Alloca la sequenza (DNA) generata solo dal Root e successivamente inviata a tutti i processi mediante Bcast
- Possiede una copia locale di pat_found, array utilizzato per tenere traccia di dove viene trovato il pattern, e seq_matches utilizzato per specificare per ogni posizione della sequenza quanti pattern sono stati trovati con un nucleotide in quella posizione.
- Calcola il numero di pattern che deve cercare, il primo e l'ultimo. Se ci sono rimanenze vengono assegnate all'ultimo rank.

Fase di ricerca:

```
1 /* 5. Search for each pattern */
2 // Each rank searches for its own patterns
3 unsigned long start;
4 int pat;
5 for( pat = my_first_pattern; pat <= my_last_pattern; pat++ ) {
6
7     /* 5.1. For each possible starting position */
8     for( start=0; start <= seq_length - pat_length[pat]; start++) {
9         /* 5.1.1. For each pattern element */
10        for( lind=0; lind<pat_length[pat]; lind++) {
11            /* Stop this test when different nucleotids are found */
12            if ( sequence[start + lind] != pattern[pat][lind] ) break;
13        }
14        /* 5.1.2. Check if the loop ended with a match */
15        if ( lind == pat_length[pat] ) {
16            pat_matches++;
17            pat_found[pat] = start;
18            break;
19        }
20    }
21    /* 5.2. Pattern found */
22    if ( pat_found[pat] != (unsigned long)NOT_FOUND ) {
23        /* 4.2.1. Increment the number of pattern matches on the sequence positions */
24        increment_matches( pat, pat_found, pat_length, seq_matches );
25    }
26 }
```

- Ogni rank cerca nella sequenza solo il proprio range di pattern
- Ogni rank aggiornerà esclusivamente pat_matches (numero di pattern trovati), seq_matches e pat_found locali.

Problema

Nella seguente funzione ogni processo incrementerà il proprio array seq_matches, successivamente quando dovrà essere sommato in uno unico succederà più volte che lo stesso indice passerà da -1 (NOT_FOUND) a 0.

Ciò risulta un problema nel calcolo del checksum perché l'array finale potrebbe contenere numeri inferiori.

```
● ● ●
1 void increment_matches( int pat, unsigned long *pat_found, unsigned long *pat_length, int *seq_matches ) {
2     unsigned long ind;
3     for( ind=0; ind<pat_length[pat]; ind++ ) {
4         if ( seq_matches[ pat_found[pat] + ind ] == NOT_FOUND )
5             seq_matches[ pat_found[pat] + ind ] = 0;
6         else
7             seq_matches[ pat_found[pat] + ind ]++;
8     }
9 }
```

Gestione seq_matches:

```
1 // Each rank sends its own seq_matches to rank 0
2 if(rank > 0){
3     MPI_Send(seq_matches, seq_length, MPI_INT, 0, 0, MPI_COMM_WORLD);
4 }
5
6 // Rank 0 receives all the seq_matches from all the other ranks and sums them up
7 if (rank == 0) {
8     int *local_seq_matches;
9     local_seq_matches = (int *)malloc( sizeof(int) * seq_length );
10
11    for (int i = 1; i < size; i++) {
12        MPI_Recv(local_seq_matches, seq_length, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
13
14        for (unsigned long j = 0; j < seq_length; j++) {
15            if (seq_matches[j] >= 0 && local_seq_matches[j] >= 0) {
16                seq_matches[j] = seq_matches[j] + local_seq_matches[j] + 1;
17            } else {
18                if(seq_matches[j] == NOT_FOUND){
19                    seq_matches[j] = local_seq_matches[j];
20                }
21            }
22        }
23    }
24
25    free(local_seq_matches);
26 }
27 }
```

- Ogni Rank utilizza la Send() per inviare il proprio seq_matches
- Il Root alloca un array locale in cui immagazzina ad ogni iterazione il seq_matches che riceve
- Utilizzando il proprio seq_matches vi somma quello che ha ricevuto
- Per ogni indice se sono entrambi a 0 li somma e aggiunge 1, se invece il seq_matches del root è -1 verrà sostituito con l'altro valore.
- In tutti gli altri casi non fa nulla

Gestione pat_found e pat_matches:

- pat_found è modificato solo in parte, in base al range di pattern, e l'unico modo per inviare solo quel determinato pezzo di array è utilizzare Gatherv()
- Ogni rank specificherà in send_counts e displs il numero di pattern che ha analizzato e il primo
- Il root in questo modo riceverà da ognuno il pezzo che prenderà il posto del proprio locale
- Pat_matches è stato gestito semplicemente con una Reduce()

```
1 // Each rank sends its own pat_found to rank 0 but only the data that he processed
2 int send_counts[size];
3 send_counts[rank] = my_patterns;
4
5 int displs[size];
6 displs[rank] = my_first_pattern;
7
8 MPI_Allgather(&my_patterns, 1, MPI_INT, send_counts, 1, MPI_INT, MPI_COMM_WORLD);
9 MPI_Allgather(&my_first_pattern, 1, MPI_INT, displs, 1, MPI_INT, MPI_COMM_WORLD);
10
11 MPI_Barrier( MPI_COMM_WORLD );
12
13 // Rank 0 receive all the pat_found from all the other ranks
14 MPI_Gatherv(pat_found + my_first_pattern, my_patterns, MPI_UNSIGNED_LONG, pat_found, send_counts, displs,
15             MPI_UNSIGNED_LONG, 0, MPI_COMM_WORLD);
16
17 // Each rank sends its own pat_matches to rank 0 and rank 0 sums them up
18 int total_pat_matches = 0;
19 MPI_Reduce(&pat_matches, &total_pat_matches, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
20
21 MPI_Barrier( MPI_COMM_WORLD );
22
23 // Rank 0 assing the value of total_pat_matches to pat_matches
24 if (rank == 0) {
25     pat_matches = total_pat_matches;
26 }
```

DNA Sequence Alignment

Implementazione CUDA



SAPIENZA
UNIVERSITÀ DI ROMA

Gestione Memoria:

- L'host alloca i dati necessari nella memoria globale della GPU
- Abbiamo notato che nella parte di codice non modificabile `d_pat_length` non è stato inviato nella GPU
- Il kernel viene invocato considerando tutti e 1024 thread che può contenere un blocco
- Il numero di blocchi è calcolato mediante la divisione numero di pattern / 1024 approssimato per difetto nel caso in cui ci siano meno di 1024 pattern

```
● ● ●
1 // Allocate and copy data to the GPU memory
2 unsigned long *g_seq_length;
3 int *g_pat_number;
4 char *g_sequence;
5 int *g_seq_matches;
6 int *g_pat_matches;
7 unsigned long *g_pat_found;
8
9 CUDA_CHECK_FUNCTION(cudaMalloc(&g_seq_length, sizeof(unsigned long)));
10 CUDA_CHECK_FUNCTION(cudaMalloc(&g_pat_number, sizeof(int)));
11 CUDA_CHECK_FUNCTION(cudaMalloc(&g_sequence, seq_length * sizeof(char)));
12 CUDA_CHECK_FUNCTION(cudaMalloc(&g_seq_matches, seq_length * sizeof(int)));
13 CUDA_CHECK_FUNCTION(cudaMalloc(&g_pat_matches, sizeof(int)));
14 CUDA_CHECK_FUNCTION(cudaMalloc(&g_pat_found, pat_number * sizeof(unsigned long)));
15
16 CUDA_CHECK_FUNCTION(cudaMemcpy(g_seq_length, &seq_length, sizeof(unsigned long), cudaMemcpyHostToDevice));
17 CUDA_CHECK_FUNCTION(cudaMemcpy(g_pat_number, &pat_number, sizeof(int), cudaMemcpyHostToDevice));
18 CUDA_CHECK_FUNCTION(cudaMemcpy(g_sequence, sequence, seq_length * sizeof(char), cudaMemcpyHostToDevice));
19 CUDA_CHECK_FUNCTION(cudaMemcpy(g_seq_matches, seq_matches, seq_length * sizeof(int), cudaMemcpyHostToDevice));
20 int init_value = 0;
21 CUDA_CHECK_FUNCTION(cudaMemcpy(g_pat_matches, &init_value, sizeof(int), cudaMemcpyHostToDevice));
22 CUDA_CHECK_FUNCTION(cudaMemcpy(g_pat_found, pat_found, pat_number * sizeof(unsigned long), cudaMemcpyHostToDevice));
23
24 CUDA_CHECK_FUNCTION(cudaMemcpy(d_pat_length, pat_length, pat_number * sizeof(unsigned long), cudaMemcpyHostToDevice));
25
26 // Launch the kernel
27 sequencer<<<ceil(pat_number/1024.0), 1024>>>(g_seq_length, g_pat_number, g_sequence,
28 d_pat_length, d_pattern, g_seq_matches, g_pat_matches, g_pat_found);
29
```

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     // Get the process number
8     int tid = blockIdx.x * blockDim.x + threadIdx.x;
9
10    // Check if the thread is within the number of patterns
11    if (tid < *g_pat_number) {
12
13        // Get the pattern number
14        pat = tid;
15
16        /* 5.1. For each possible starting position */
17        for( start=0; start <= *g_seq_length - d_pat_length[pat]; start++) {
18            /* 5.1.1. For each pattern element */
19            for( lind=0; lind<d_pat_length[pat]; lind++) {
20                /* Stop this test when different nucleotids are found */
21                if ( g_sequence[start + lind] != d_pattern[pat][lind] ) break;
22            }
23            /* 5.1.2. Check if the loop ended with a match */
24            if ( lind == d_pat_length[pat] ) {
25                atomicAdd(g_pat_matches, 1);
26                atomicExch((unsigned long long*)&g_pat_found[pat], (unsigned long long)start);
27                break;
28            }
29        }
30        /* 5.2. Pattern found */
31        if ( g_pat_found[pat] != (unsigned long)NOT_FOUND ) {
32            /* 4.2.1. Increment the number of pattern matches on the sequence positions */
33            increment_matches( pat, g_pat_found, d_pat_length, g_seq_matches );
34        }
35    }
36    __syncthreads();
37 }
```

- Verifichiamo che il thread deve lavorare calcolando il numero successivamente al numero di processo equivale anche il numero del pattern
- Il codice è il medesimo del sequenziale, vengono utilizzate le operazioni atomiche per incrementare g_pat_matches e per assegnare start a g_pat_found[pat]

Nella seguente funzione è stata utilizzata l'operazione atomica per aggiornare seq_matches

Abbiamo dovuto modificare il corpo della funzione, precedentemente se il valore era NOT_FOUND veniva impostata a 0 altrimenti incrementata di 1.

Ciò causava problemi nonostante l'utilizzo dell'atomica perché nell'array finale risultavano valori inferiori.

```
● ● ●  
1  __device__ void increment_matches( int pat, unsigned long *pat_found, unsigned long *pat_length, int *seq_matches ) {  
2      unsigned long ind;  
3      for( ind=0; ind<pat_length[pat]; ind++ ) {  
4          atomicAdd(&seq_matches[ pat_found[pat] + ind ], 1);  
5      }  
6  }
```

Riportiamo nell'host i dati necessari al checksum e liberiamo la memoria del device

```
1
2 // Copy the results back to the host
3 CUDA_CHECK_FUNCTION(cudaMemcpy(seq_matches, g_seq_matches, seq_length * sizeof(int), cudaMemcpyDeviceToHost));
4 CUDA_CHECK_FUNCTION(cudaMemcpy(&pat_matches, g_pat_matches, sizeof(int), cudaMemcpyDeviceToHost));
5 CUDA_CHECK_FUNCTION(cudaMemcpy(pat_found, g_pat_found, pat_number * sizeof(unsigned long), cudaMemcpyDeviceToHost));
6
7 // Free the GPU memory
8 CUDA_CHECK_FUNCTION(cudaFree(g_seq_matches));
9 CUDA_CHECK_FUNCTION(cudaFree(g_pat_matches));
10 CUDA_CHECK_FUNCTION(cudaFree(g_pat_found));
11 CUDA_CHECK_FUNCTION(cudaFree(d_pattern));
12 CUDA_CHECK_FUNCTION(cudaFree(d_pat_length));
13 CUDA_CHECK_FUNCTION(cudaFree(g_seq_length));
14 CUDA_CHECK_FUNCTION(cudaFree(g_pat_number));
15 CUDA_CHECK_FUNCTION(cudaFree(g_sequence));
```

DNA Sequence Alignment

Implementazione MPI+CUDA



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.

- 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().
- 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_pat_number, char *g_sequence,
2                           unsigned long *d_pat_length, char **d_pattern, int *g_seq_matches,
3                           int *g_pat_matches, unsigned long *g_pat_found, int *g_my_first_pattern) {
4     unsigned long start;
5     int pat;
6     unsigned long lind;
7
8     // Get the process number
9     int tid = blockIdx.x * blockDim.x + threadIdx.x;
10
11    // Check if the thread is within the number of patterns
12    if (tid < *g_pat_number) {
13
14        // Get the pattern number
15        pat = tid + *g_my_first_pattern;
16
17        /* 5.1. For each possible starting position */
18        for( start = 0; start <= *g_seq_length - d_pat_length[pat]; start++) {
19            /* 5.1.1. For each pattern element */
20            for( lind=0; lind < d_pat_length[pat]; lind++) {
21                /* Stop this test when different nucleotids are found */
22                if ( g_sequence[start + lind] != d_pattern[pat][lind] ) break;
23            }
24            /* 5.1.2. Check if the loop ended with a match */
25            if ( lind == d_pat_length[pat] ) {
26                atomicAdd(g_pat_matches, 1);
27                atomicExch((unsigned long long*)&g_pat_found[pat], (unsigned long long)start);
28                break;
29            }
30        }
31
32        /* 5.2. Pattern found */
33        if ( g_pat_found[pat] != (unsigned long)NOT_FOUND ) {
34            /* 4.2.1. Increment the number of pattern matches on the sequence positions */
35            increment_matches( pat, g_pat_found, d_pat_length, g_seq_matches );
36        }
37    }
38}

```

Kernel:

- Si differenzia dalla versione CUDA, passando `g_my_pat_number` e `g_my_first_pattern`.
- Ciascun thread lavora sui pattern corrispondenti al proprio intervallo.
- Viene effettuata una verifica per far lavorare solo il numero di thread necessari e per allinearsi al numero di pattern corretto vengono sommati il numero di thread e `g_my_first_pattern`.
- 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.

DNA Sequence Alignment

Implementazione OpenMP



Blocco Parallelo:

- Il primo ciclo for è stato parallelizzato poiché, nonostante i nostri tentativi, non siamo riusciti a parallelizzare anche i cicli annidati con collapse.
- 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.
- Lo schedule utilizzato è dynamic poichè 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.

```
1  /* 5. Search for each pattern */
2  unsigned long start;
3  int pat;
4
5 // Parallelize the block of code that searches for each pattern
6 #pragma omp parallel shared(pat_found, seq_matches)
7 {
8
9    // Parallelize the first for loop
10   #pragma omp for private(start, pat, lind) \
11       reduction(+:pat_matches) schedule(dynamic,4)
12
13   /* 5.1. For each possible starting position */
14   for( pat=0; pat < pat_number; pat++ ) {
15       for( start=0; start <= seq_length - pat_length[pat]; start++) {
16           for( lind=0; lind<pat_length[pat]; lind++) {
17               /* Stop this test when different nucleotids are found */
18               if ( sequence[start + lind] != pattern[pat][lind] ) break;
19           }
20           /* 5.1.2. Check if the loop ended with a match */
21           if ( lind == pat_length[pat] ) {
22               pat_matches++;
23               pat_found[pat] = start;
24               break;
25           }
26       }
27       /* 5.2. Pattern found */
28       if ( pat_found[pat] != (unsigned long)NOT_FOUND ) {
29           // Critical section
30           #pragma omp critical
31           /* 4.2.1. Increment the number of pattern matches on the sequence positions */
32           increment_matches( pat, pat_found, pat_length, seq_matches );
33       }
34   }
35 }
```

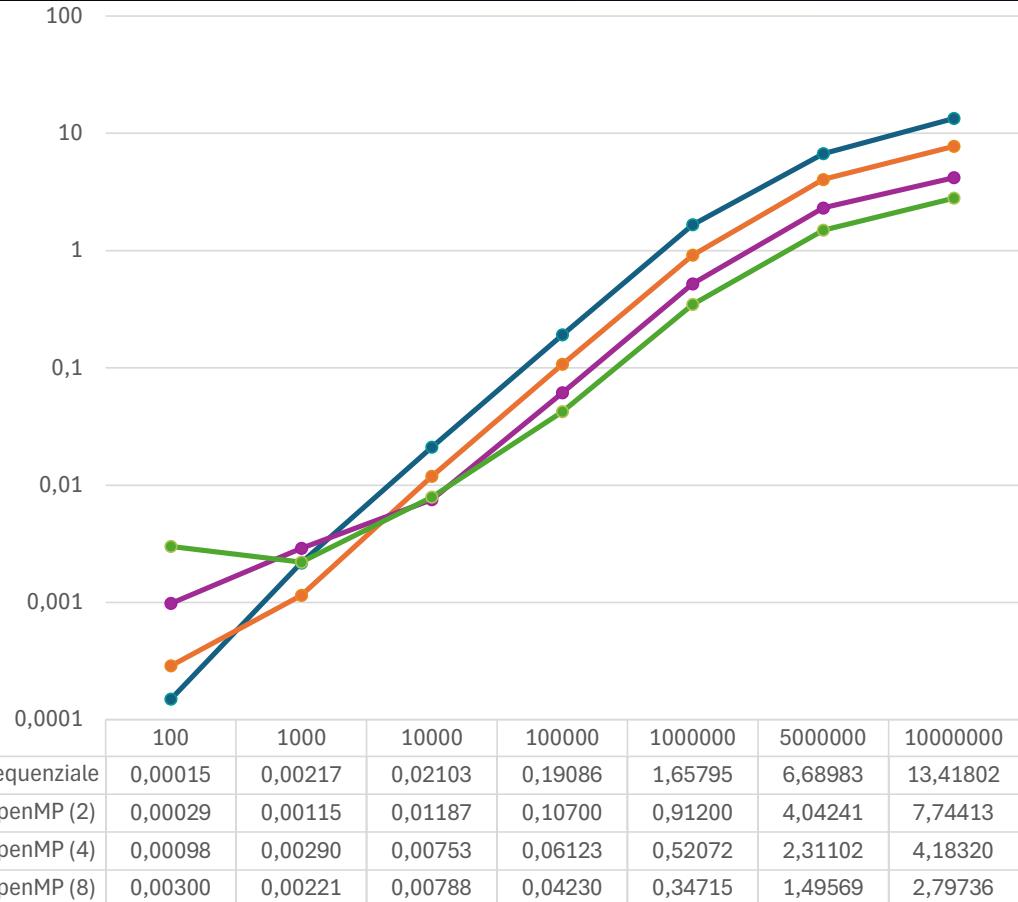
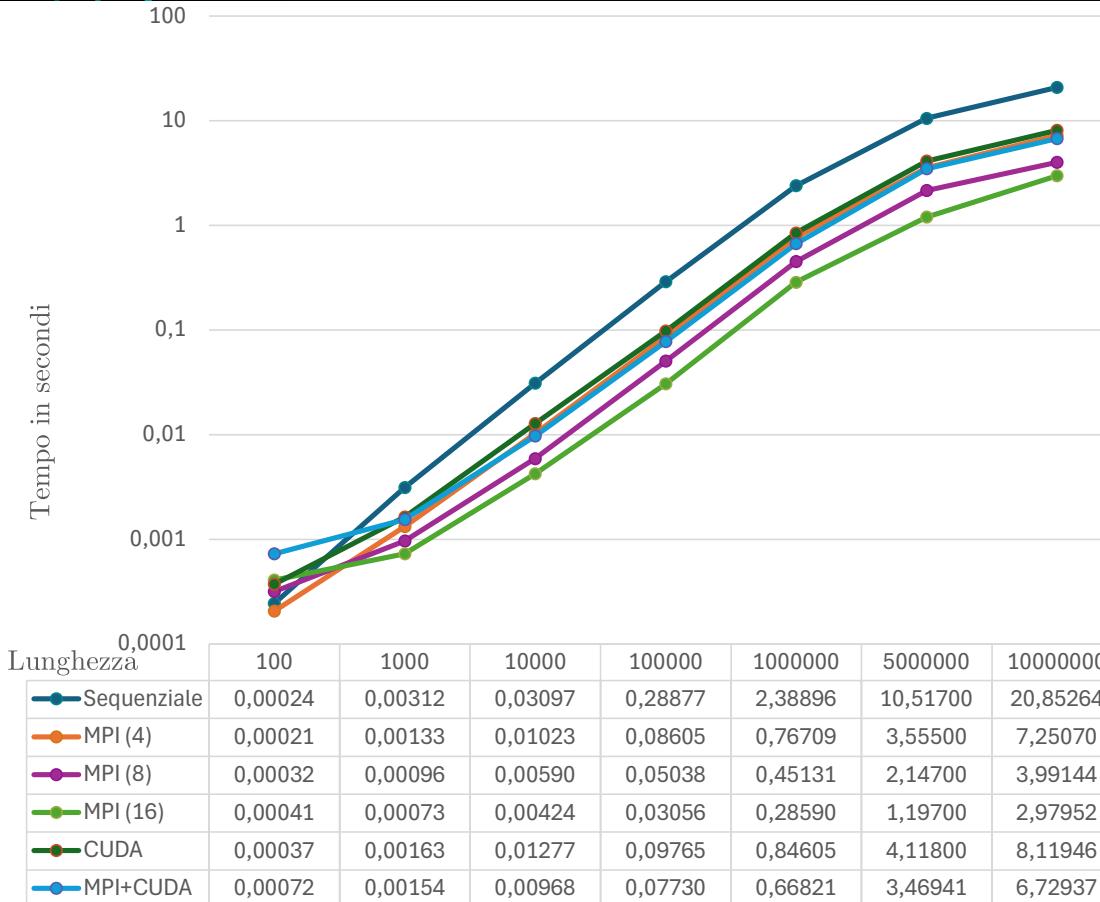
DNA Sequence Alignment

Efficienza

- 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.

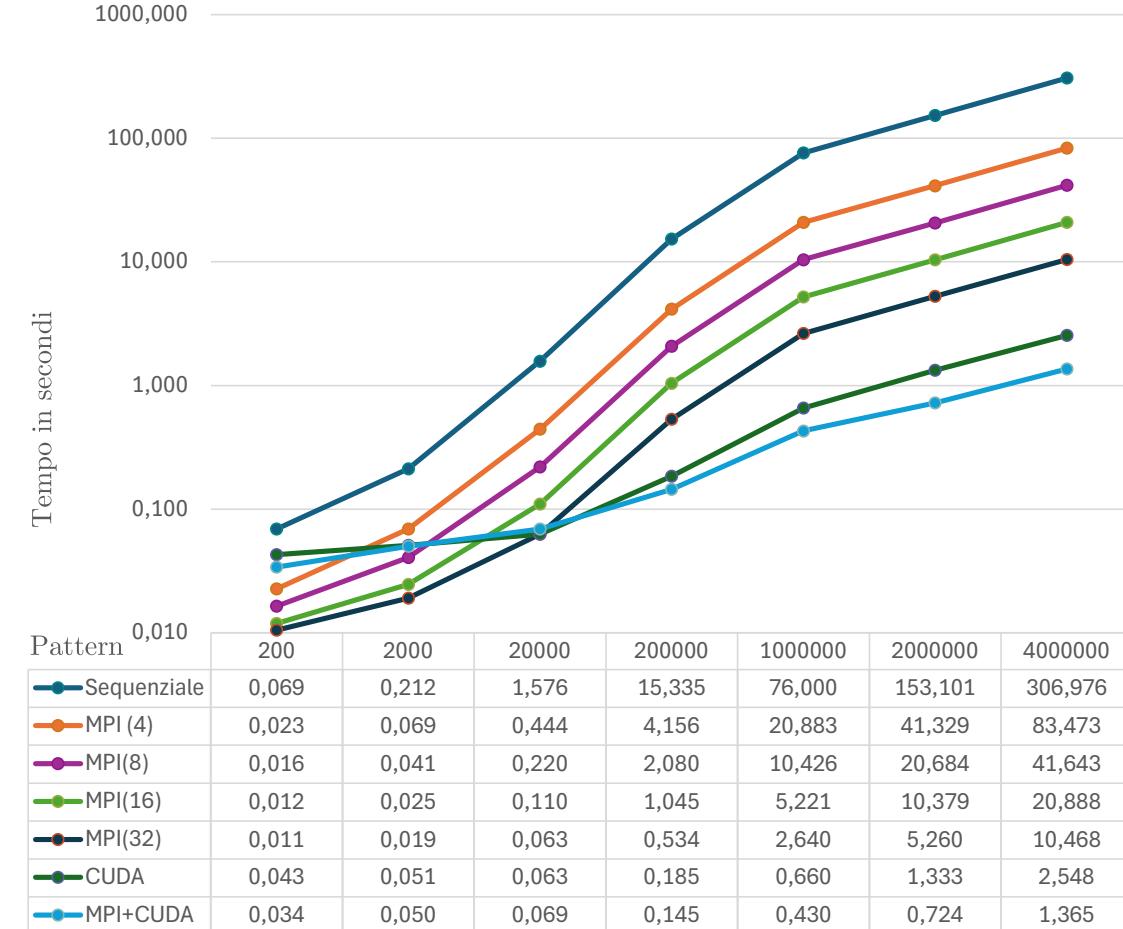


Test Lunghezza Sequenza DNA:



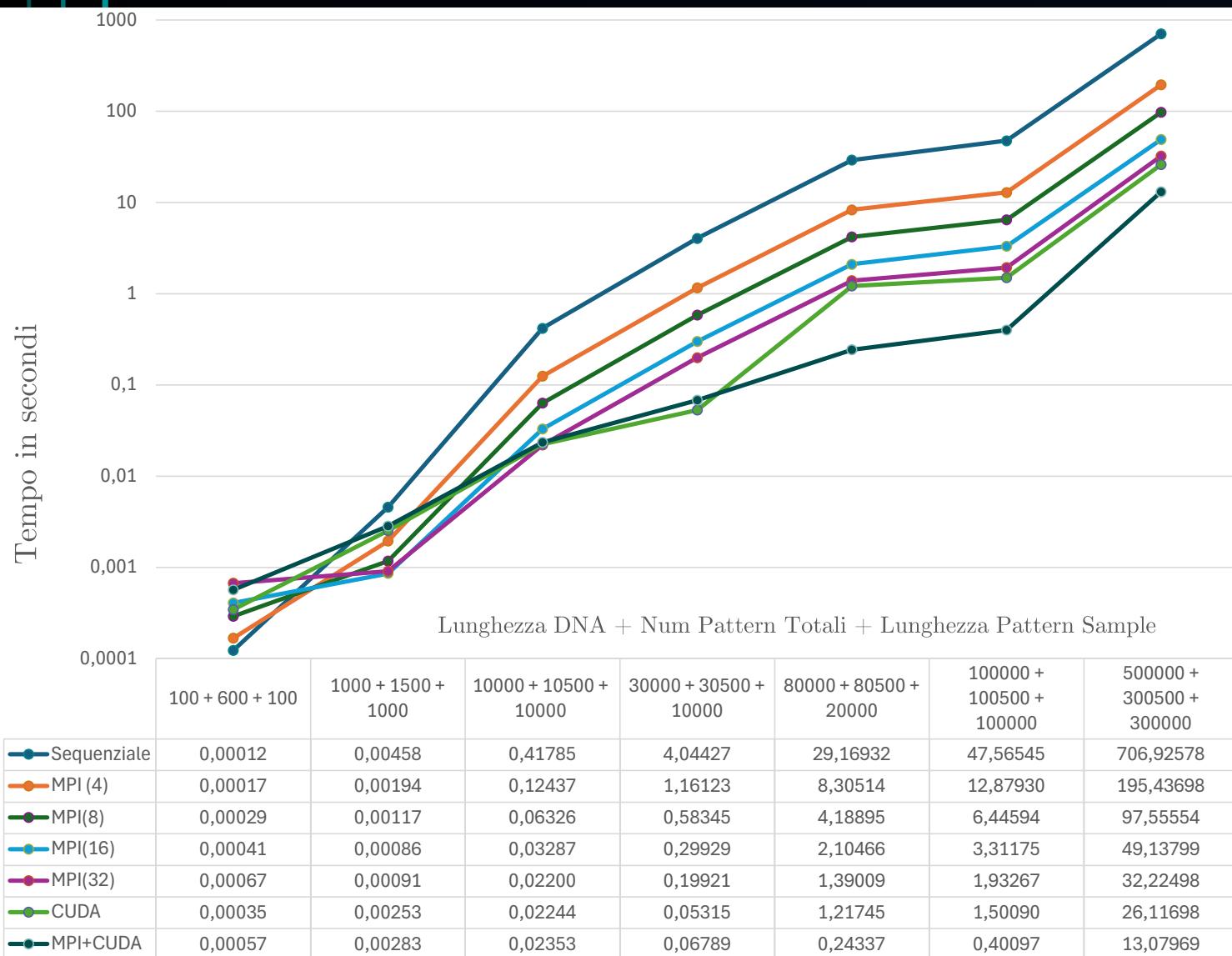
Dal secondo test MPI e OpenMP diventano più efficienti e scalano con l'aumentare del numero di processi/.
 In media MPI(16) è del 86,8% più efficiente del sequenziale e OpenMP(32) del 78,6%.
 CUDA e MPI+CUDA risultano sempre peggiori in efficienza rispetto al sequenziale del 61,3% e 67,9%.

Test numero di Pattern:



MPI e OpenMP sono più efficienti del sequenziale: MPI(32) è 96,6% più efficiente e OpenMP(32) 82,2% più efficiente. CUDA e MPI+CUDA migliorano con più pattern: CUDA è 99,1% più efficiente e MPI+CUDA 99,5%.

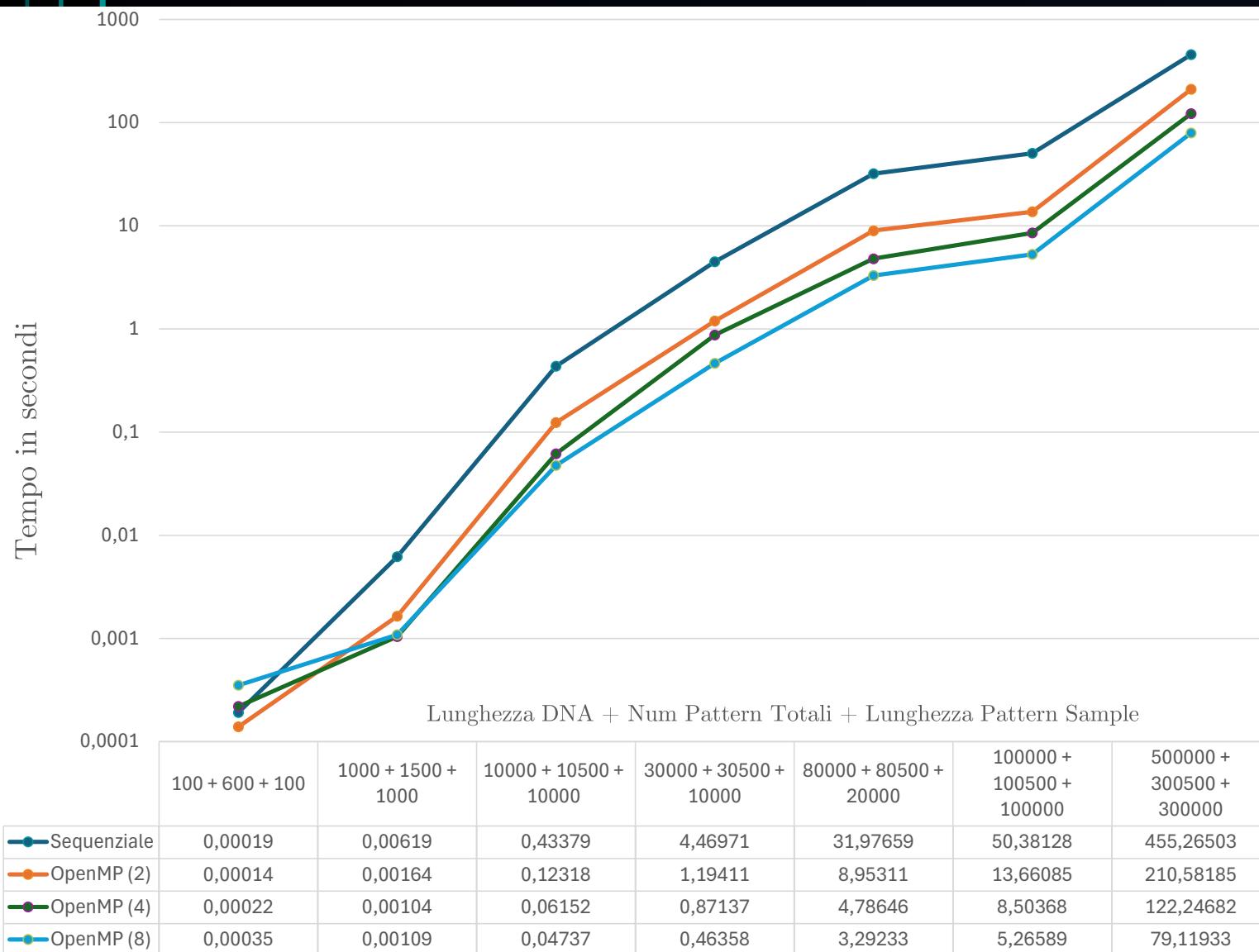
Test complessi:



Nei test più complessi, con input piccoli, le versioni parallele sono inizialmente meno efficienti rispetto al sequenziale, ad eccezione di OpenMP a due threads, che risulta subito più veloce. Man mano che i test proseguono, MPI e OpenMP scalano linearmente, migliorando rispettivamente del 95,5% e 83,7%. CUDA e MPI+CUDA diventano più veloci dal test 4 e 5, rispettivamente.

Tuttavia, le performance non scalano sempre linearmente: CUDA risulta molto più efficiente nel test 4, mentre MPI+CUDA domina nei test 5 e 6. In media, CUDA offre un miglioramento del 96,3% e MPI+CUDA del 98,2%.

Test complessi:



Nei test più complessi, con input piccoli, le versioni parallele sono inizialmente meno efficienti rispetto al sequenziale, ad eccezione di OpenMP a due threads, che risulta subito più veloce. Man mano che i test proseguono, MPI e OpenMP scalano linearmente, migliorando rispettivamente del 95,5% e 83,7%. CUDA e MPI+CUDA diventano più veloci dal test 4 e 5, rispettivamente.

Tuttavia, le performance non scalano sempre linearmente: CUDA risulta molto più efficiente nel test 4, mentre MPI+CUDA domina nei test 5 e 6. In media, CUDA offre un miglioramento del 96,3% e MPI+CUDA del 98,2%.