SAPIENZA
UNIVERSITÀ DI ROMA

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Dipartimento di Informatica

DNA Sequence

Autori:
Simone Lidonnici - (2061343)
Marco Casu - (2041262)

20 gennaio 2025

1 Introduzione

Le parti principali di codice da parallelizzare sono la creazione della sequenza e la ricerca dei pattern. Per sequenze medio-piccole quest'ultima compone la maggior parte del tempo di compilazione, mentre aumentando la grandezza della sequenza (nell'ordine dei miliardi), la maggior parte del tempo è richiesto per generare la sequenza. Da notare che nel programma sequenziale il tempo non conta la generazione di quest'ultima mentre nei programmi paralleli (MPI, OpenMP, CUDA e MPI+OpenMP) si, cosa che causa una diminuzione dello speedup.

1.1 Test e calcolo dello speedup

Le varie versioni del programma sono state testate sul cluster eseguendo 10 test e controllando il tempo medio di essi, scartando dal calcolo della media il caso peggiore ed il caso migliore, in particolare il test è stato eseguito con i seguenti parametri:

```
500000 0.35 0.2 0.25 30000 2000 1000 30000 2000 1000 500 100 M 4353435
```

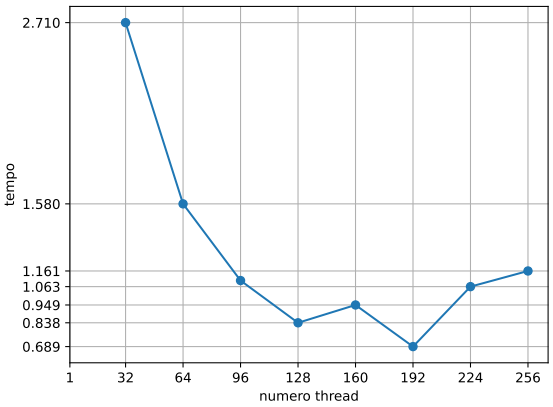
2 MPI

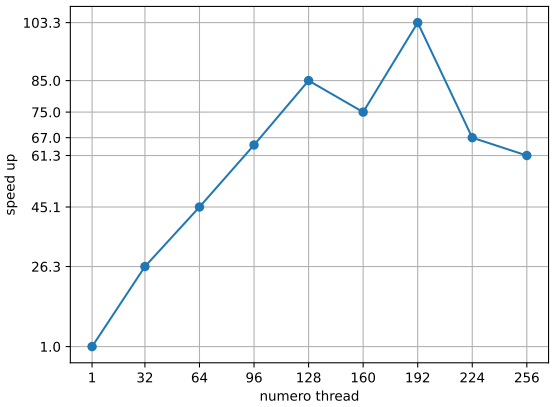
Nel programma MPI la ricerca dei pattern è stata distribuita uniformemente tra i rank, se t è il numero dei processi, allora ognuno ricercherà $\lceil \frac{n}{t} \rceil$ pattern, con n numero totale di pattern (sia sample che random). Se $\frac{n}{t}$ non è un numero intero, l'ultimo rank ricercherà meno pattern rispetto ad ogni altro. Per avere poi il valore corretto dei pattern trovati e dei pattern in ogni punto della sequenza vengono eseguite delle collettive `MPI_Reduce` su `seq_matches` e `pat_found`.

Riguardo la generazione della sequenza, è stato osservato che, questa, risulta più veloce nella versione sequenziale, anche se viene adoperato un singolo thread, la funzione `generate_random_sequence` impiega sempre meno tempo ad essere eseguita nella versione sequenziale piuttosto che in quella parallela. Sono stati tentati due approcci:

- Si è inizialmente provato a far generare l'intera sequenza ad un solo processo, per poi eseguire una `MPI_Broadcast`, condividendola a tutti gli altri processi. Tale versione si è rivelata più veloce della versione iniziale (in cui ogni rank genera autonomamente la sequenza) ma comunque più lenta della versione sequenziale.
- La seconda opzione, presente nel file finale, è stata quella di dividere la sequenza tra i vari rank, nello stesso modo in cui vengono divisi i pattern e poi eseguire una `MPI_Allreduce` per far avere a tutti i rank la sequenza completa. Data la natura puramente sequenziale delle funzioni rng, ogni rank esegue un `rng_skip` per poter iniziare a generare i suoi numeri random da un punto avanzato della sequenza.

Di seguito si può vedere il grafico dei tempi e dello speed up in relazione al numero di processi MPI, i test sono stati eseguiti con il numero massimo di processi su un nodo (32) e aumentando i nodi progressivamente:



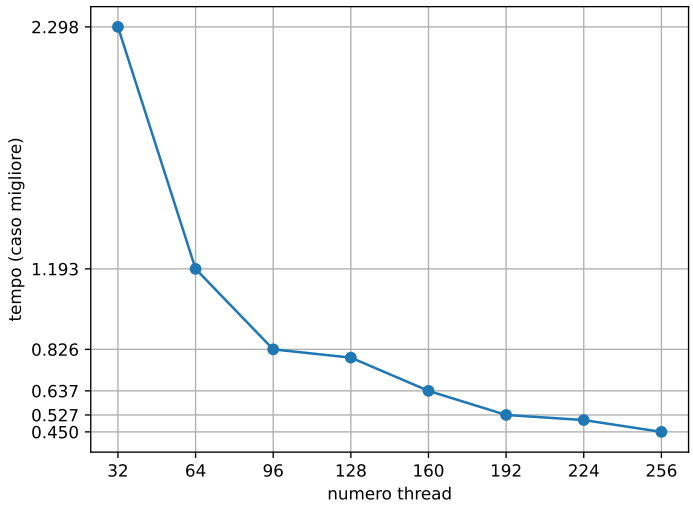


L'esecuzione sequenziale (non riportata nel grafico) ha impiegato 71.27 secondi.

Se si vogliono consultare le informazioni in maniera più pratica, si può controllare la seguente tabella:

numero di processi	tempo	speedup	efficienza
sequenziale	71.27	1	1
32	2.71	26.27	0.82
64	1.57	45.07	0.70
96	1.10	64.64	0.67
128	0.83	84.97	0.66
160	0.94	75.03	0.46
192	0.68	103.28	0.53
224	1.06	66.95	0.29
256	1.16	61.32	0.23

Nota : il tempo medio per l'esecuzione con 256 rank risulta maggiore al tempo medio con 96 rank, anche se il caso migliore risulta più rapido, non è chiaro se l'aleatoreità delle differenze di tempo fra differenti esecuzioni del programma sia dovuta al sovraccarico del cluster,è riportato il grafico dei tempi nei *casi migliori*:



Si noti come nel caso migliore, all'aumentare dei thread il tempo va sempre diminuendo.

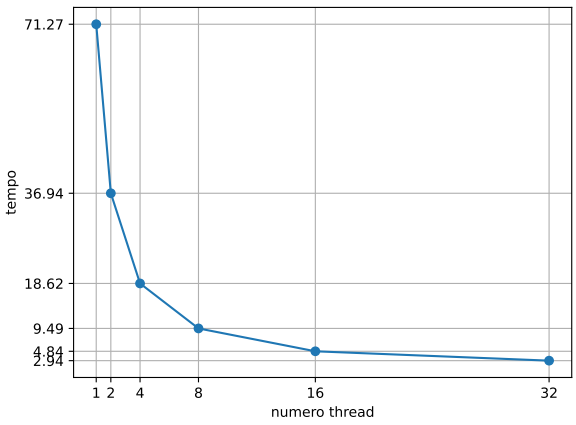
3 OpenMP

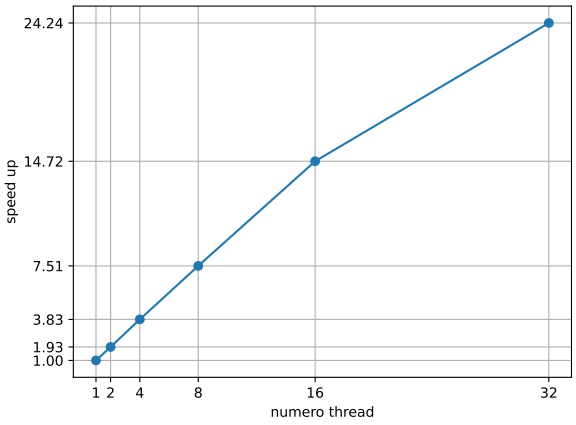
Nel programma OpenMP per quanto riguarda la ricerca dei pattern abbiamo creato delle variabili `pat_matches` e `seq_matches` private per ogni thread e abbiamo parallelizzato solamente il ciclo più esterno, tramite un `#pragma omp for`, questo perché i tre cicli non potevano essere collassati in uno unico data la presenza di `break` al loro interno.

Dopo il ciclo `for` relativo alla ricerca dei pattern ogni thread avrà operato sulle proprie copie private di `pat_matches` e `seq_matches`. Sarà necessario collossare i risultati parziali ottenuti da ogni thread, sommandone i valori. A tal proposito, sarà necessario sequenziare una porzione di codice con la direttiva `#pragma omp critical`, in cui ogni thread sommerà i valori di `pat_matches` e `seq_matches` per avere i valori finali corretti.

Per quanto riguarda invece la creazione della sequenza abbiamo usato lo stesso approccio di MPI, cioè dividerla in parti uguali in base all'id del thread e far eseguire ad ogni thread `rng_skip` fino al punto dove deve iniziare a generare i propri numeri random. Per impostare il numero di thread è stato aggiunto un argomento in input, in modo da poterli impostare da linea di comando.

Di seguito si può vedere il grafico dei tempi e dello speedup in relazione al numero di thread, i test sono stati eseguiti aumentando il numero di thread progressivamente:





Si osservi come la decrescita del tempo impiegato a terminare il programma è inizialmente esponenziale. In seguito, è riportata la tabella dei tempi e dello speed up:

numero di thread	tempo	speedup	efficienza
sequenziale	71.27	1	1
2	36.94	1.92	0.96
4	18.62	3.82	0.95
8	9.48	7.51	0.93
16	4.84	14.71	0.91
32	2.93	24.24	0.75

Si noti come a parità di processi (nel caso con 32 thread), l'esecuzione parallela in MPI risulta più efficiente rispetto quella in OpenMP.