

UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE ED
ELETTRICA E MATEMATICA APPLICATA



CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

High Performance Computing (HPC) Report

Nome	Cognome	Matricola	E-Mail
Emanuele	Relmi	0622702368	e.relmi@studenti.unisa.it

ANNO ACCADEMICO 2024/2025

INDICE

1 WP1 - Generazione Dataset e Algoritmo Sequenziale	4
1.1 Generazione delle stringhe random	4
1.1.1 Output e organizzazione dei dataset	4
1.1.2 Considerazioni metodologiche	5
1.2 Algoritmo Sequenziale di Manber-Myers	5
1.2.1 Strutture dati	6
1.2.2 Fasi dell'algoritmo (doubling)	6
1.2.3 Algoritmo di Kasai per l'LCP	7
1.2.4 Flusso di esecuzione	8
1.2.5 Analisi di complessità	8
1.2.6 Note implementative e casi limite	8
1.3 Validazione	8
2 WP2 - Parallelizzazione con OpenMP	9
2.1 Struttura del codice	9
2.2 Punti di parallelizzazione effettivi	10
2.3 Scelte di sincronizzazione e sicurezza	10
2.4 Ricerca LRS	11
2.5 Gestione dei thread e parametri di runtime	11
2.6 Flusso di esecuzione	11
2.7 Analisi di complessità e scalabilità	11
2.8 Validazione	12
3 WP3 - Parallelizzazione con MPI	13
3.1 Struttura del codice	13

INDICE

3.2	Distribuzione dei dati e I/O	14
3.3	Costruzione locale del SA	14
3.4	k -way merge globale	14
3.5	Calcolo LCP e LRS	15
3.6	Comunicazione e orchestrazione	15
3.7	Analisi di complessità e scalabilità	16
3.8	Validazione	16
3.9	Limiti e possibili estensioni	16
4	WP4 - Parallelizzazione ibrida MPI + OpenMP	18
4.1	Struttura del codice	18
4.2	Distribuzione dei dati e I/O	18
4.3	Costruzione locale del SA con OpenMP	19
4.4	k -way merge globale	20
4.5	Calcolo LCP e LRS	20
4.6	Orchestrazione, misure e report	21
4.7	Analisi di complessità e scalabilità	21
4.8	Validazione	21
4.9	Limiti e possibili estensioni	22
5	WP5 - Parallelizzazione su GPU con CUDA	23
5.1	Versione single-stream	23
5.1.1	Struttura del codice	23
5.1.2	Algoritmo e mappatura su GPU	24
5.1.3	Dettagli implementativi	25
5.1.4	Flusso di esecuzione end-to-end	26
5.1.5	Analisi di complessità	26
5.1.6	Note di implementazione	27
5.1.7	Note sulle prestazioni	27
5.1.8	Validazione	27
5.2	Versione multi-stream	28
5.2.1	Struttura del codice	28
5.2.2	Partizionamento in chunk e creazione degli stream	28
5.2.3	Pipeline di un'iterazione di <i>doubling</i>	29
5.2.4	Validazione dell'output	29
5.2.5	Metriche e reporting	29
5.2.6	Complessità e scalabilità	30
5.2.7	Note di implementazione	30

INDICE

5.2.8	Footprint di memoria su device	30
5.2.9	Confronto con single-stream	30
5.2.10	Limitazioni e possibili estensioni	30
5.2.11	Validazione	31
6	Analisi delle prestazioni	32
6.1	Versione Sequenziale	33
6.1.1	Analisi dei tempi	33
6.1.2	Profiling della memoria	35
6.2	Versione OpenMP	36
6.2.1	Analisi dei tempi	36
6.2.2	Profiling della memoria	37
6.3	Versione MPI	38
6.3.1	Analisi dei tempi	38
6.3.2	Profiling della memoria	40
6.4	Versione MPI+OpenMP	41
6.4.1	Analisi dei tempi	41
6.4.2	Profiling della memoria	43
6.5	Versione CUDA (single-stream)	44
6.5.1	Analisi dei tempi	44
6.5.2	Profiling della memoria	46
6.6	Versione CUDA <i>multi-stream</i>	47
6.6.1	Analisi dei tempi	47
6.6.2	Profiling della memoria	49
6.6.3	Confronto con single-stream	49
6.6.4	Conclusioni	49
6.7	Confronto complessivo tra modelli	50

CAPITOLO 1

WP1 - GENERAZIONE DATASET E ALGORITMO SEQUENZIALE

Questo work package è incentrato sulle fasi di preparazione dei dati di input e di implementazione sequenziale per la costruzione dell'array dei suffissi (Suffix Array, SA) e del relativo array delle lunghezze dei prefissi comuni massimi (Longest Common Prefix, LCP). La costruzione del SA è realizzata tramite l'algoritmo di **Manber-Myers**, la cui complessità è $O(n \log n)$, mentre l'array LCP è calcolato tramite l'algoritmo di **Kasai** con complessità $O(n)$. Queste componenti costituiscono la baseline funzionale e prestazionale su cui si innestano i successivi work package dedicati alle versioni parallele su CPU (OpenMP, MPI, ibrida OMP+MPI) e GPU (CUDA).

1.1 Generazione delle stringhe random

La generazione degli input testuali è stata realizzata mediante uno script in *bash*, denominato `generate_random_strings.sh`. Lo scopo dello script è creare file contenenti stringhe casuali di lunghezza predefinita, espressa in megabyte (MB), utilizzando come sorgente `/dev/urandom`, al fine di fornire dataset sintetici di dimensioni fisse per i benchmark. Il comando impiegato è `dd`, configurato con blocchi da 1 B.

1.1.1 Output e organizzazione dei dataset

Lo script produce un file binario contenente una stringa casuale di lunghezza opportunamente ridotta rispetto al target nominale, in modo tale che *stringa + strutture*

1. WP1 - GENERAZIONE DATASET E ALGORITMO SEQUENZIALE

ausiliarie (*SA*, *LCP*, *workspace*) occupino complessivamente circa la memoria desiderata. Sono stati considerati target complessivi pari a 1, 50, 100, 200 e 500 MB, dove per ciascun valore, la dimensione effettiva della stringa è stata calcolata dividendo il budget totale per un fattore di overhead fissato a 21, corrispondente al rapporto tra il testo e le strutture dati necessarie all'algoritmo di Manber-Myers e di Kasai. I file risultanti sono organizzati nella cartella `random.strings/` e contengono byte arbitrari nell'intervallo [0, 255]. Non viene applicata alcuna restrizione sull'alfabeto (non soltanto ASCII stampabile); possono pertanto comparire byte NUL e qualsiasi altro valore, simulando condizioni realistiche di input eterogeneo.

Target complessivo (MB)	Dimensione stringa effettiva (MB)
1	≈ 0.05
50	≈ 2.38
100	≈ 4.77
200	≈ 9.54
500	≈ 23.8

Table 1.1: Relazione tra target di memoria complessivo e dimensione effettiva della stringa generata.

1.1.2 Considerazioni metodologiche

La scelta di utilizzare `/dev/urandom` consente di ottenere sequenze a distribuzione approssimativamente uniforme, evitando pattern ripetuti che avrebbero potuto influenzare i tempi di ordinamento. Inoltre, la generazione diretta a livello binario garantisce efficienza e scalabilità, permettendo di produrre dataset di dimensioni elevate senza overhead computazionale aggiuntivo.

1.2 Algoritmo Sequenziale di Manber-Myers

La prima implementazione sviluppata è di tipo sequenziale e costituisce la baseline per il confronto con le versioni parallele. L'obiettivo è fornire una misura di riferimento in termini di correttezza e prestazioni, su cui valutare i benefici delle successive ottimizzazioni.

Il codice è suddiviso in tre file principali:

- `suffix_array.h`: definizione delle strutture dati e delle funzioni pubbliche per la costruzione del SA e del LCP;
- `suffix_array.cpp`: implementazione delle funzioni secondo l'algoritmo di Manber-Myers e l'algoritmo di Kasai;

1. WP1 - GENERAZIONE DATASET E ALGORITMO SEQUENZIALE

- `seq_main.cpp`: programma principale che gestisce il caricamento del file di input, invoca gli algoritmi e stampa i risultati.

1.2.1 Strutture dati

Il suffix array è rappresentato da un vettore di interi `sa` di lunghezza n , contenente gli indici di partenza dei suffissi ordinati. Sono inoltre utilizzate strutture ausiliarie $O(n)$:

- `rank`: vettore di interi con il rango corrente del suffisso che inizia in i (inverso di `sa`);
- `lcp`: vettore di interi per le lunghezze dei prefissi comuni massimi tra suffissi consecutivi in `sa`;
- `cnt` e `next`: workspace per il conteggio per-bucket e per le posizioni di scrittura nel raffinamento;
- `bh` e `b2h`: vettori booleani per marcare rispettivamente le teste dei bucket correnti e le nuove teste dopo un passo di raffinamento.

Footprint lineare

Oltre a `text` (byte dell'input), i vettori `sa`, `rank`, `cnt`, `next`, `lcp` richiedono ciascuno $4n$ byte (interi a 32 bit), per un totale di circa $20n$ byte, cui si sommano i due vettori booleani bit-compattati.

1.2.2 Fasi dell'algoritmo (doubling)

L'implementazione segue il *doubling* con raffinamento per *bucket* usando i vettori ausiliari `cnt`, `next` e i marcatori `bh/b2h`:

1. **Inizializzazione.** Si inizializza `sa[i]=i` e si ordina `sa` per `text[i]`. Si costruiscono i *bucket* iniziali (suffissi con lo stesso primo byte), marcando le teste in `bh`, e si assegna `rank` coerentemente.
2. **Ciclo di raddoppio.** Per $h \in \{1, 2, 4, \dots\}$ si ripete finché tutti i ranghi sono distinti:
 - (a) **Preparazione delle teste di bucket.** Per ogni testa `bh`, si azzera `cnt` e si calcolano le posizioni di scrittura `next` (prefisso cumulato per-bucket).
 - (b) **Pre-posizionamento dei suffissi corti.** Si inseriscono per primi gli indici $i \in [n - h, n)$ nelle posizioni di testa dei rispettivi bucket (in base a `rank[i]`). Questi sono i suffissi per cui il “secondo rango” sarebbe mancante; il loro trattamento anticipato equivale ad assegnare implicitamente una sentinella -1 senza memorizzarla.

- (c) **Raffinamento stabile per-bucket.** Si scansiona l'array `sa` nell'ordine corrente (che riflette il secondo rango crescente) e, per ciascun elemento t , si pone $j = \text{sa}[t] - h$. Se $j \geq 0$, si inserisce j nella prossima posizione disponibile del suo bucket: $\text{sa2}[\text{next}[\text{rank}[j]]] \leftarrow j$ e $\text{next}[\text{rank}[j]] \leftarrow \text{next}[\text{rank}[j]] + 1$. Questo realizza un ordinamento per coppie $(\text{rank}[j], \text{rank}[j + h])$ senza costruire esplicitamente le coppie e senza confronti fuori-bordo.
- (d) **Ricalcolo dei ranghi e nuove teste.** Si copia `sa2` in `sa`, quindi si scansiona `sa` e si aggiorna `rank`: si marca una nuova testa in `b2h` quando la coppia di ranghi tra posizioni adiacenti cambia; altrimenti si *propaga* il rango. Si pone `bh` $\leftarrow \text{b2h}$.
- (e) **Criterio di arresto.** Se $\text{rank}[\text{sa}[n - 1]] = n - 1$ tutti i bucket hanno ranghi distinti e l'ordinamento è completo.

Note implementative

L'array `sa2` non è utilizzato nel codice effettivo, in quanto viene effettuata una sostituzione *in-place*. L'utilizzo di tale vettore nel flusso appena descritto è da intendersi come puramente didattico.

La procedura sopra descritta, inoltre, è equivalente all'ordinamento delle coppie $(\text{rank}[i], \text{rank}[i + h])$:

- il passo di pre-posizionamento implementa la sentinella -1 per i suffissi con $i + h \geq n$;
- la scansione di `sa` in ordine corrente garantisce la stabilità rispetto al “secondo rango”;
- il posizionamento tramite `next` e il ricalcolo di `rank` e `b2h` realizzano il raffinamento per “primo rango”.

1.2.3 Algoritmo di Kasai per l'LCP

Dati `sa` e il suo inverso `rank`, l'array `lcp` è calcolato in $O(n)$:

1. si mantiene una variabile h (lunghezza corrente del match) che non viene azzerata a ogni passo;
2. per i con $\text{rank}[i] > 0$, si confronta il suffisso in i con quello precedente in ordine lessicografico, $j = \text{sa}[\text{rank}[i]-1]$, prolungando il match finché $\text{text}[i+h] = \text{text}[j+h]$;
3. si assegna $\text{lcp}[\text{rank}[i]] = h$ e, se $h > 0$, si decrementa h di 1 prima di passare a $i + 1$ (*riuso* del prefisso comune).

1.2.4 Flusso di esecuzione

Il flusso sequenziale è il seguente:

1. caricamento del dataset binario in memoria (`text`);
2. costruzione di `sa` mediante Manber-Myers;
3. calcolo di `lcp` con Kasai;
4. estrazione della *Longest Repeated Substring* come $\max_k \text{lcp}[k]$.

1.2.5 Analisi di complessità

- **Tempo.** Manber-Myers esegue $O(\log n)$ iterazioni di raffinamento; ogni iterazione è $O(n)$ grazie al conteggio per-bucket e alla scansione sequenziale, dunque la complessità totale è $O(n \log n)$. Kasai è $O(n)$.
- **Memoria.** Oltre a `text`, lo spazio ausiliario è $O(n)$: `sa`, `rank`, `lcp`, `cnt`, `next`, `bh`, `b2h`.

1.2.6 Note implementative e casi limite

- **Input binari.** I confronti lessicografici si effettuano su byte non firmati (`uint8_t`); ciò assicura l'ordine corretto su valori $[0, 255]$ e la corretta gestione dei byte NUL.
- **Sentinella.** Nel confronto della coppia $(\text{rank}[i], \text{rank}[i + h])$, per $i + h \geq n$ si usa un rango sentinella (es. -1) inferiore a qualunque rango valido, così da posizionare in coda i suffissi più corti.
- **Stabilità.** Il raffinamento per-bucket deve preservare la stabilità rispetto al primo rango; l'uso di `next` e `cnt` realizza una forma di counting-stable per il secondo rango.
- **Arresto precoce.** La verifica $\text{rank}[\text{sa}[n - 1]] = n - 1$ consente di interrompere il ciclo quando tutti i bucket sono di cardinalità unitaria.

1.3 Validazione

La correttezza del SA viene verificata controllando che per indici consecutivi $i < j$ valga $S[\text{sa}[i]\dots] \leq S[\text{sa}[j]\dots]$ in ordine lessicografico. La correttezza dell'LCP si accerta verificando che, per $k > 0$, risulti $\text{lcp}[k] = \text{lcp}(S[\text{sa}[k]\dots], S[\text{sa}[k - 1]\dots])$. L'individuazione della *Longest Repeated Substring* (LRS) si ottiene come $\max_k \text{lcp}[k]$; oltre alla lunghezza, si riporta la posizione nell'array dei suffissi e una stampa esadecimale dei byte corrispondenti.

CAPITOLO 2

WP2 - PARALLELIZZAZIONE CON OPENMP

Questo work package introduce la parallelizzazione a memoria condivisa mediante **OpenMP**, mantenendo invariata la logica della baseline sequenziale. L'obiettivo è ridurre il tempo di esecuzione intervenendo sulle porzioni a scansione lineare e sulle operazioni indipendenti per indice, lasciando sequenziali i passaggi con dipendenze dati intrinseche (ordinamento iniziale, calcolo dei bucket e parte centrale del raddoppio, ciclo principale di Kasai).

2.1 Struttura del codice

La versione OpenMP riusa le stesse interfacce della baseline:

- `suffix_array_omp.cpp`: costruzione del SA (Manber-Myers) e dell'LCP (Kasai) con direttive `#pragma omp` dove possibile;
- `suffix_array.h`: header comune;
- `omp_main.cpp`: programma principale, gestione I/O, calcolo metriche e confronto con baseline sequenziale.

2.2 Punti di parallelizzazione effettivi

L'inserimento delle direttive segue un criterio conservativo: parallelizzare solo loop *embarrassingly parallel* o con scritture disgiunte, evitando race condition.

Costruzione del SA

- **Inizializzazione di sa:** `#pragma omp parallel for su sa[i]=i` (iterazioni indipendenti).
- **Marcatura teste di bucket:** `#pragma omp parallel for` per impostare `bh[i]` e azzerare `b2h[i]` (accessi disgiunti).
- **Ciclo di raddoppio ($h = 1, 2, 4, \dots$):**
 - *Ricostruzione di sa e aggiornamento bh:* `#pragma omp parallel for` su due vettori in scrittura disgiunta: $(sa[rank[i]] = i)$ e $(bh[i] = bh[i] \vee b2h[i])$.
- **Rank inverso finale:** `#pragma omp parallel for su (rank[sa[i]] = i)`.

Costruzione dell'LCP

- **Inizializzazione del rank inverso:** `#pragma omp parallel for su (rank[sa[i]] = i)`.

L'inizializzazione del rank inverso è parallelizzata, mentre il ciclo principale di Kasai rimane sequenziale per via delle dipendenze su h .

2.3 Scelte di sincronizzazione e sicurezza

Nel codice non sono dichiarate esplicitamente clausole `private` o `shared`, si fa affidamento sulle regole di default di OpenMP, le quali stabiliscono che:

- le variabili di loop (`i`, `j`, ecc.) sono rese automaticamente *private*;
- i vettori allocati al di fuori della regione parallela (`text`, `sa`, `rank`, `bh`, `b2h`, `cnt`, `next`, `lcp`) sono *shared*;
- le variabili dichiarate all'interno del corpo del loop (`s`, `head`, contatori temporanei) sono locali allo scope del thread e quindi *private*.

Questa combinazione consente di evitare race condition senza bisogno di ulteriori clausole in quanto i vettori condivisi sono sempre scritti in posizioni disgiunte, mentre gli indici e i temporanei sono gestiti in autonomia da ciascun thread.

2.4 Ricerca LRS

La scansione della LRS su `lcp` è parallelizzata con una riduzione personalizzata:

1. ogni thread mantiene un *record locale* (`local_len`, `local_i`, `local_pos`);
2. al termine del proprio blocco, il thread entra in una sezione `critical` per aggiornare il *best globale* (`best_len`, `best_i`, `best_pos`) rispettando i tie-break: massima lunghezza, poi indice LCP minimo, poi posizione SA minima;
3. la politica di scheduling è `static` (default esplicito nei `for`).

Questa scelta evita condizioni di race pur mantenendo costante l'ordine dei tie-break.

2.5 Gestione dei thread e parametri di runtime

Il numero di thread effettivo è ricavato con `omp_get_max_threads()` e riportato nel log, tuttavia può essere controllato dall'utente tramite il parametro `OMP_NUM_THREADS`. Le misure temporali usano `omp_get_wtime()`. Non sono presenti trasferimenti H↔D o comunicazioni di rete (`time_transfers_comm = 0`).

2.6 Flusso di esecuzione

1. lettura del dataset binario (`text`);
2. allocazione dei vettori di lavoro;
3. costruzione del SA e dell'LCP;
4. ricerca della LRS in parallelo;
5. calcolo delle metriche: `time_compute_pure = time_sa + time_lcp`, `time_total_compute`, `time_alloc`, `memory_overhead_ratio`, `throughput` in MB/s.

Se disponibile, vengono caricati i valori medi della baseline sequenziale dal CSV (`seq_summary`) per stimare `speedup` ed `efficiency` (calcolata come $\frac{speedup}{num_threads}$).

2.7 Analisi di complessità e scalabilità

- **Complessità asintotica:** invariata rispetto alla baseline ($O(n \log n)$ per SA, $O(n)$ per LCP).
- **Accelerazione attesa:** i benefici derivano dai `for` in parallelizzazione pura (inizializzazione, ricostruzione SA, rank inverso, teste di bucket, LRS-scan). La parte

centrale del raddoppio rimane sequenziale; l'accelerazione globale è pertanto limitata dalla frazione seriale (legge di Amdahl).

- **Bottleneck:** al crescere dei thread, la banda di memoria diventa il vincolo dominante sulle sezioni $O(n)$; oltre un certo numero di core, lo speedup tende a saturare.
- **Memoria:** identica alla versione sequenziale ($O(n)$). Le strutture private dei thread sono trascurabili.

2.8 Validazione

La versione OpenMP produce `sa` e `lcp` identici alla baseline, con LRS coincidente (uguale lunghezza e posizione). La riproducibilità è facilitata dall'uso di dataset binari determinati per dimensione e dal confronto con la baseline (`seq_summary.csv`). Il report a console include: dimensione input, numero di thread, tempi parziali (`time_io`, `time_alloc`, `time_sa`, `time_lcp`, `time_lrs_scan`), throughput, speedup ed efficiency quando disponibile la baseline.

CAPITOLO 3

WP3 - PARALLELIZZAZIONE CON MPI

Questo work package introduce la parallelizzazione distribuita tramite **MPI**, con decomposizione a blocchi (*chunks*) del testo d'ingresso. Ogni processo costruisce il suffix array (*SA*) del proprio *chunk* locale; successivamente il rank 0 esegue un *k-way merge* delle liste ordinate per ottenere il *SA globale*, calcola l'*LCP* e ricava la *LRS*.

La costruzione locale usa il *doubling* con ordinamento per coppie tramite `std::sort` (quindi complessità $O(n_r \log^2 n_r)$), mentre il merge finale è basato su un heap con confronto lessicografico diretto sui suffissi del testo completo.

3.1 Struttura del codice

- `suffix_array_mpi.h`: dichiarazione delle funzioni di costruzione del SA locale e di merge globale.
- `suffix_array_mpi.cpp`: implementazione della costruzione locale (`build_suffix_array_subset`) e del *k-way merge* (`merge_k_sorted_lists`).
- `mpi_main.cpp`: orchestrazione MPI (I/O distribuito con `MPI_I0`, raccolta risultati, merge su rank 0, calcolo LCP e LRS), reporting delle metriche e confronto con la baseline sequenziale.

3.2 Distribuzione dei dati e I/O

Il testo di dimensione n byte è suddiviso in P chunk (uno per processo), con dimensioni quasi uniformi:

$$\text{chunk_size} = \left\lceil \frac{n}{P} \right\rceil, \quad \text{start}_r = r \cdot \text{chunk_size}, \quad \text{actual_chunk_r} = \min(\text{chunk_size}, n - \text{start}_r)$$

Ogni rank legge il *proprio* segmento mediante I/O collettivo:

```
MPI_File_read_at_all(file, start_r, chunk_r, ...)
```

Il parametro **mb** (dimensione del dataset) è validato dal rank 0 e diffuso a tutti con **MPI_Bcast**.

3.3 Costruzione locale del SA

La costruzione del SA sul chunk locale adotta il paradigma *doubling* con ordinamento per coppie:

1. **Rank iniziali** Si inizializza **rk[i]** al valore del byte **chunk[i]**.
2. **Iterazioni di raddoppio** Per $k = 1, 2, 4, \dots$:
 - (a) si costruisce un vettore **bucket** di triple $((\text{rk}[i], \text{rk}[i+k]), i)$, dove il secondo rango è -1 se $i + k \geq n_r$;
 - (b) si ordina **bucket** con **std::sort** (confronto lessicografico su coppie di interi);
 - (c) si riassegnano i ranghi in **tmp** (ranghi uguali per coppie uguali; incremento su cambio di coppia) e si copia in **rk**;
 - (d) se i ranghi sono tutti distinti ($r = n_r - 1$) si termina in anticipo (*early stop condition*).
3. **Estrazione dell'ordine** Si scrive **sa_out[i] = bucket[i].second**.

Nota sulla complessità

L'uso di **std::sort** a ogni iterazione implica $O(n_r \log n_r)$ per iterazione e $\Theta(\log n_r)$ iterazioni, per un totale $O(n_r \log^2 n_r)$. Questo differisce dalla baseline sequenziale (bucketed, $O(n \log n)$), ma risulta pratico e auto-contenuto per la fase locale.

3.4 *k*-way merge globale

Il rank 0 riceve da tutti i **SA locali** (concatenati in **all_sa**) e i **conteggi counts[r]** (lunghezze dei chunk) tramite **MPI_Gather/Gatherv**. A questo punto esegue un *k*-way merge con min-heap:

- si prepara un heap di elementi `Item{idx, which}`, dove `idx` è l'**indice globale** del suffisso nel testo completo e `which` identifica il chunk di provenienza;
- il comparatore confronta direttamente i suffissi `text[idx]` e `text[jdx]` byte-per-byte finché divergono o si raggiunge la fine del testo;
- a ogni estrazione si inserisce il successivo suffisso dalla stessa lista locale, finché si esauriscono tutti i suffissi.

Costo computazionale del merge

Con P liste e n suffissi totali, il merge ha costo $O(n \log P)$ per confronto dell'heap. Poiché il comparatore effettua un confronto lessicografico diretto sui suffissi, il costo effettivo è $O(L)$ caratteri per confronto, dove L è la lunghezza del prefisso comune atteso.

3.5 Calcolo LCP e LRS

Una volta ottenuto `sa_global`, il rank 0:

1. rilegge il testo *completo* in memoria;
2. calcola l'array `lcp` tramite `build_lcp(...)` (algoritmo di Kasai, tempo $O(n)$);
3. esegue una scansione lineare per determinare la *Longest Repeated Substring* come $\max_k lcp[k]$, registrando anche la posizione in `sa_global`.

Queste fasi sono locali al rank 0 e non comportano ulteriori comunicazioni.

3.6 Comunicazione e orchestrazione

- **Broadcast parametri:** `MPI_Bcast` per disporre `mb` su tutti i rank.
- **I/O distribuito:** `MPI_File_read_at_all` per leggere ciascun chunk dal file condiviso.
- **Raccolta SA locali:** `MPI_Gather` (lunghezze) + `MPI_Gatherv` (concatenazione degli SA) verso rank 0.
- **Barrier di misura:** `MPI_Barrier` prima delle sezioni temporizzate (SA locale, comunicazione, merge).
- **Metriche per-rank:** raccolta di tempi locali (`time_sa_local`, `time_comm_local`, `time_io`, `time_alloc`) e stampa su rank 0.

Le metriche aggregate usate per il report globale sono:

$$\text{time_compute_pure} = \max_r \text{time_sa_local}^{(r)} + \text{time_merge} + \text{time_lcp}$$

escludendo I/O e comunicazioni. La **throughput** (MB/s) è calcolata su `time_compute_pure`; lo **speedup** rispetto alla baseline sequenziale è basato sui tempi medi presi dal file *CSV seq_summary* caricato da rank 0.

3.7 Analisi di complessità e scalabilità

- **Costruzione locale (per rank)** $O(n_r \log^2 n_r)$ con `std::sort` su coppie a ogni iterazione di doubling; $n_r \approx \lceil n/P \rceil$.
- **Merge globale** $O(n \log P \cdot \bar{L})$, dove \bar{L} è la lunghezza media del prefisso comune esaminata dal comparatore.
- **LCP (Kasai)** $O(n)$ su rank 0.
- **Scalabilità** Lo speedup è determinato da:
 1. il fattore P nella fase locale ($n_r \approx n/P$),
 2. il costo del merge $O(n \log P)$ sul rank 0,
 3. l'eventuale sbilanciamento dei chunk (ultimo rank più corto) e la saturazione di banda del filesystem in **MPI-IO**.

In pratica, al crescere di P il merge e l'LCP sul rank 0 diventano il collo di bottiglia dominante.

3.8 Validazione

La correttezza del SA globale è verificata confrontando l'ordinamento lessicografico dei suffissi in `sa_global` rispetto al testo completo. L'LCP è validato verificando che, per $k > 0$, valga $\text{lcp}[k] = \text{lcp}(S[\text{sa_global}[k]], S[\text{sa_global}[k-1]])$.

La LRS è infine $\max_k \text{lcp}[k]$, con posizione corrispondente in `sa_global`.

3.9 Limiti e possibili estensioni

- **Merge più efficiente** Evitare i confronti byte-per-byte usando ranghi globali a finestra (ad es. coppie $(\text{rk}[i], \text{rk}[i+h])$ condivise) o strategie di *string sample sort / radix-based* per ridurre \bar{L} .

- **Costruzione locale** $O(n_r \log n_r)$ Sostituire `std::sort` con ordinamenti *radix* sulle coppie di ranghi per recuperare la complessità teorica del doubling $O(n \log n)$.
- **Parallelizzazione LCP** Possibile parallelizzare parzialmente Kasai su segmenti con correzioni agli overlap, oppure calcolare LCP *on-the-fly* durante il merge con mantenimento di uno stato di confronto.

CAPITOLO 4

WP4 - PARALLELIZZAZIONE IBRIDA MPI + OPENMP

Questo work package adotta un modello *ibrido* che combina parallelismo distribuito (**MPI**) e parallelismo a memoria condivisa (**OpenMP**). Ogni rank MPI costruisce il suffix array (*SA*) del proprio *chunk* sfruttando OpenMP all'interno del nodo; il rank 0 esegue poi il *k-way merge* dei SA locali, calcola l'*LCP* e ricava la *LRS*. L'obiettivo è ridurre i tempi rispetto a MPI puro, accelerando la costruzione locale con i thread OpenMP.

4.1 Struttura del codice

- `suffix_array_mpi_omp.cpp`: implementazione ibrida della costruzione locale del SA (`build_suffix_array_subset`, con OpenMP) e del *k-way merge* (`merge_k_sorted_lists`).
- `mpi_omp_main.cpp`: orchestrazione MPI, I/O distribuito, raccolta degli SA locali, merge su rank 0, calcolo LCP e LRS, raccolta delle metriche e confronto con baseline.

4.2 Distribuzione dei dati e I/O

La partizione del testo da n byte è identica alla versione MPI pura:

$$\text{chunk_size} = \left\lceil \frac{n}{P} \right\rceil, \quad \text{start}_r = r \cdot \text{chunk_size}, \quad \text{actual_chunk}_r = \min(\text{chunk_size}, n - \text{start}_r),$$

con lettura parallela tramite `MPI_File_read_at_all` del segmento locale `chunk`. Il parametro `mb` è validato su rank 0 e diffuso con `MPI_Bcast`. Ogni rank registra anche `omp_get_max_threads()` per tracciare i thread OpenMP utilizzabili localmente.

Inoltre, il numero di thread per rank è controllabile tramite il parametro `OMP_NUM_THREADS`.

4.3 Costruzione locale del SA con OpenMP

La funzione `build_suffix_array_subset(chunk, sa_out)` adotta il *doubling* con ordinamento per coppie ($rk[i], rk[i+k]$), introducendo OpenMP dove le iterazioni sono indipendenti:

1. Inizializzazione dei ranghi

- *Parallelizzato*: `#pragma omp parallel for` su $rk[i] \leftarrow chunk[i]$.

2. Iterazioni di raddoppio per $k = 1, 2, 4, \dots$:

(a) Riempimento delle coppie (bucket)

- *Parallelizzato*: `#pragma omp parallel for` su $bucket[i] \leftarrow ((rk[i], rk[i+k] \vee -1), i)$.

(b) Ordinamento per coppie

- *Sequenziale*: `std::sort(bucket.begin(), bucket.end())` per avere determinismo e semplicità.

(c) Riassegnazione dei ranghi

- *Sequenziale*: scansione di `bucket` per assegnare nuovi ranghi in `tmp` (r incrementa quando la coppia cambia).

(d) Scrittura dei ranghi

- *Parallelizzato*: `#pragma omp parallel for` su $rk[i] \leftarrow tmp[i]$.

(e) Early stopping

- Se $r = n_r - 1$ (tutti i ranghi distinti) si esce dal ciclo.

3. Estrazione dell'ordine finale

- *Parallelizzato*: `#pragma omp parallel for` su $sa_out[i] \leftarrow bucket[i].second$.

Note implementative su default OpenMP

Non sono usate clausole esplicite `private/shared`. Si sfruttano i *default OpenMP*:

- le variabili di loop sono *private*;

- i vettori allocati all'esterno (`rk`, `tmp`, `bucket`, `sa_out`) sono *shared*;
- le scritture avvengono in posizioni disgiunte.

Le due sezioni lasciate sequenziali (`sort` e riassegnazione ranghi) hanno dipendenze dati intrinseche.

Complessità locale

Come nella versione MPI pura, l'uso di `std::sort` per ogni k comporta $O(n_r \log n_r)$ per iterazione e $\Theta(\log n_r)$ iterazioni: $O(n_r \log^2 n_r)$. L'accelerazione rispetto a MPI puro deriva dal parallelismo delle fasi di riempimento coppie, scrittura ranghi ed estrazione finale.

4.4 k -way merge globale

Il rank 0 raccoglie **gli SA locali** con `MPI_Gather/Gatherv` e li fonde con un min-heap (`std::priority_queue`) che confronta lessicograficamente i suffissi del *testo completo*:

- `Item{idx, which}` incapsula l'indice *globale* nel testo e il chunk di provenienza.
- Il comparatore `Cmp` confronta i suffissi `text[idx]` e `text[jdx]` avanzando su byte uguali e fermandosi alla prima divergenza o a fine testo.
- Dopo ogni estrazione, si inserisce dall'elenco *which* il successivo suffisso (indice globale aggiornato).

Costo del merge

Con P liste e n elementi, $O(n \log P)$ operazioni heap; ogni confronto costa $O(L)$ caratteri dove L è il prefisso comune medio.

4.5 Calcolo LCP e LRS

A merge concluso, il rank 0:

1. carica il testo *completo* in memoria;
2. calcola `lcp` con **Kasai** ($O(n)$);
3. scansiona `lcp` per ottenere la *LRS* come $\max_k lcp[k]$, registrando la posizione in `sa_global`.

4.6 Orchestrazione, misure e report

- **Broadcast parametri:** MPI_Bcast per disporre `mb` su tutti i rank.
- **I/O distribuito:** MPI_File_read_at_all per leggere ciascun chunk dal file condiviso.
- **Raccolta SA locali:** MPI_Gather (lunghezze) + MPI_Gatherv (concatenazione degli SA) verso rank 0.
- **Barrier di misura:** MPI_Barrier prima delle sezioni temporizzate (SA locale, comunicazione, merge) misurate con MPI_Wtime().
- **Raccolta metriche:** per ciascun rank si raccolgono `time_sa_local`, `time_comm_local`, `time_io`, `time_alloc` e `omp_threads`.
- **Aggregati globali (rank 0):** si considera il massimo per-rank come collo di bottiglia distribuito:

$$\text{time_compute_pure} = \max_r \text{time_sa_local}^{(r)} + \text{time_merge} + \text{time_lcp}.$$

Si riportano inoltre `time_total_compute`, throughput in MB/s, `time_transfers_comm`, `memory_overhead_ratio` e `speedup` ed `efficiency` rispetto alla baseline sequenziale.

4.7 Analisi di complessità e scalabilità

- **Locale per rank (OpenMP)** $O(n_r \log^2 n_r)$ per la costruzione (`std::sort` per iterazione), con accelerazione data dai `for` parallelizzati.
- **Merge globale** $O(n \log P \cdot \bar{L})$ su rank 0, dove \bar{L} è il prefisso comune medio.
- **LCP** $O(n)$ su rank 0.
- **Scalabilità** La fase locale beneficia di p thread per rank; all'aumentare di P , il merge e l'LCP su rank 0 tendono a dominare. La banda di memoria locale limita lo speedup OpenMP; la banda del filesystem e del network limita lo scaling MPI.

4.8 Validazione

Il *SA globale* risultante dal merge viene usato per calcolare `lcp` (Kasai) e la *LRS*. La correttezza è verificata rispetto alle definizioni:

$$S[\text{sa_glob}[i]] \leq S[\text{sa_glob}[i+1]], \quad \text{lcp}[k] = \text{lcp}(S[\text{sa_glob}[k]], S[\text{sa_glob}[k-1]]).$$

La LRS è $\max_k \text{lcp}[k]$ con la corrispondente posizione nel SA.

4.9 Limiti e possibili estensioni

- **Riduzione del costo locale:** rimpiazzare `std::sort` con *radix* sulle coppie di ranghi per avvicinarsi a $O(n_r \log n_r)$; preservare il determinismo.
- **Merge accelerato:** usare ranghi globali a finestra (*h-doubling* anche nel merge) o tecniche *string/radix* per abbattere il costo dei confronti byte-per-byte.

CAPITOLO 5

WP5 - PARALLELIZZAZIONE SU GPU CON CUDA

Questo work package introduce la parallelizzazione su GPU dell'algoritmo mediante **CUDA**. Si distinguono due varianti: *single-stream* e *multi-stream* (*MS*). La costruzione del SA è svolta integralmente su GPU, il calcolo dell'array LCP (Kasai) rimane su CPU, in quanto lineare e non dominante rispetto al SA. Questa versione parallelizzata fa uso di **Thrust** per le primitive di *sorting*, *scan* e *gather*, oltre a kernel CUDA personalizzati per le operazioni di inizializzazione, marcatura dei gruppi e *scatter* dei ranghi.

5.1 Versione single-stream

5.1.1 Struttura del codice

- `suffix_array_cuda.h`: interfaccia pubblica in namespace `cuda_sa` con struttura `Metrics` (tempi H/D) e overload utili per vettori host.
- `suffix_array_cuda.cu`: implementazione della costruzione SA su GPU (`build_suffix_array_cuda`), kernel CUDA (`k_init_ranks`, `k_build_key_r2`, `k_mark_groups_by_rank_u8`, `k_scatter_ranks`) e uso di primitive Thrust.
- `cuda_main.cpp`: driver host I/O file, invocazione SA su GPU, calcolo LCP su CPU, ricerca LRS con *tie-breaking* deterministico, reporting di metriche e velocità.

5.1.2 Algoritmo e mappatura su GPU

L'algoritmo segue il paradigma del *doubling* in cui, ad ogni iterazione k , i suffissi sono ordinati secondo le coppie di ranghi:

$$(r_1(i) = \text{rank}[i], r_2(i) = \text{rank}[i + k]),$$

con sentinella per $r_2(i)$ quando $i + k \geq n$. La versione CUDA realizza ogni iterazione con la sequenza:

1. **Inizializzazione ranghi** Kernel `k_init_ranks` per porre $\text{rank}[i] \leftarrow \text{text}[i]$ (byte $\in [0, 255]$).
2. **Costruzione di SA iniziale** `thrust::sequence` su `d_sa` per ottenere $[0, \dots, n-1]$.
3. **Ordinamento stabile per secondo rango r_2**
 - Kernel `k_build_key_r2`: genera chiavi 32-bit $\text{key}[i] \leftarrow \text{rank}[i+k]+1$ (oppure 0 se $i+k \geq n$) così che la sentinella risulti minima.
 - `thrust::stable_sort_by_key(d_key, d_sa)`: ordina gli indici di suffisso in base a r_2 .
4. **Ordinamento stabile per primo rango r_1**
 - `thrust::gather(d_sa, d_rank, d_key)` per costruire le chiavi r_1 nell'ordine attuale di `sa`.
 - `thrust::stable_sort_by_key(d_key, d_sa)`: ordina stabilmente per r_1 , preservando la stabilità rispetto a r_2 . Il risultato è l'ordinamento per coppie (r_1, r_2) *senza* materializzare le coppie stesse.
5. **Marcatura delle frontiere di gruppo** Kernel `k_mark_groups_by_rank_u8` confronta (r_1, r_2) dei suffissi adiacenti in `sa` e pone $\text{flags}[0]=0$ e i restanti $\text{flags}[i]=1$ quando cambia almeno uno dei due ranghi.
6. **Assegnazione dei nuovi ranghi**
 - `thrust::transform(d_flags, d_rank_scan, u8 → int)` e `thrust::exclusive_scan` su `d_rank_scan`: produce gli *ID di gruppo* (0, 1, 2, ...).
 - Kernel `k_scatter_ranks`: $\text{new_rank}[\text{sa}[i]] \leftarrow \text{rank_scan}[i]$.
 - `thrust::reduce(thrust::maximum)` su `d_new_rank` per determinare \max_r . Se $\max_r = n-1$, tutti i ranghi sono distinti quindi l'algoritmo si ferma.
 - `swap(d_rank, d_new_rank)` per l'iterazione successiva con $k \leftarrow 2k$.

L'uso combinato di *due sort stabili* (per r_2 e poi per r_1) è equivalente a un sort per la coppia (r_1, r_2) ed evita la costruzione esplicita di un array di coppie. La sentinella su r_2 è realizzata via `rank[i+k]+1` (con 0 per indici fuori range), rendendo i suffissi più corti correttamente minori.

5.1.3 Dettagli implementativi

Kernel CUDA e configurazione

- **Griglia:** blocchi da 256 thread (`BLOCK=256`); griglia $\lceil n/256 \rceil$.
- **Kernel:** `k_init_ranks` (caricamento byte→rank), `k_build_key_r2` (chiavi secondarie 32-bit con sentinella), `k_mark_groups_by_rank_u8` (flag tra adiacenti in SA), `k_scatter_ranks` (scrittura dei nuovi ranghi in ordine).
- **Cronometria GPU:** `cudaEventRecord` / `cudaEventElapsedTime` su stream di default (single-stream).

Primitive Thrust

- `sequence` per inizializzare SA come $[0, \dots, n-1]$;
- `stable_sort_by_key` per ordinamenti stabili su chiavi intere (r_2 e poi r_1); per chiavi a 32 bit Thrust usa tipicamente una variante *radix*, con costo lineare in numero di elementi;
- `gather` per mappare r_1 nell'ordine corrente di `sa`;
- `transform` ($u8 \rightarrow int$) e `exclusive_scan` per ottenere ID di gruppo crescenti.

Metriche raccolte

La struttura `cuda_sa::Metrics` rileva:

- `h_alloc_s`: tempo host per allocazioni/inizializzazioni (inclusa allocazione dei buffer device).
- `h_h2d_s` e `h_d2h_s`: copie Host→Device e Device→Host.
- `gpu_kernel_s`: tempo totale kernel (da evento GPU start a stop) che include tutte le iterazioni del *doubling* e le primitive Thrust invocate sullo stream di default.

Il `main` (`cuda_main.cpp`) combina `gpu_kernel_s` con il tempo di Kasai su CPU (`time_lcp_cpu`) per `time_compute_pure`, e riporta throughput, speedup ed efficienza rispetto alla baseline sequenziale.

Footprint di memoria su device

Sono allocati i seguenti vettori sul *device*:

Vettore	Size
d_text	nB
d_rank	4nB
d_new_rank	4nB
d_sa	4nB
d_rank_scan	4nB
d_key32	4nB
d_flags	nB

Il totale è $\approx (1+1)n + 5 \cdot 4n \simeq 22n$ byte, cui si somma l'overhead dei container Thrust.

5.1.4 Flusso di esecuzione end-to-end

1. **I/O host:** lettura del file binario (`cuda_main.cpp`).
2. **SA su GPU:** chiamata a `build_suffix_array_cuda` con riempimento delle metriche.
3. **LCP su CPU:** `build_lcp(text, sa, rk, lcp)` (Kasai).
4. **LRS:** ricerca del massimo in `lcp` con *tie-breaking* deterministico (*plateau-aware*, minima posizione nel testo).
5. **Report:** tempi `time_io`, `time_alloc_host_dev`, `time_h2d`, `time_kernel_gpu`, `time_lcp_cpu`, `time_d2h`; throughput, speedup, efficiency, memory_overhead_ratio.

5.1.5 Analisi di complessità

- **Per iterazione di doubling:** Due `stable_sort_by_key` su chiavi intere 32-bit. Con backend *radix*, il costo è $O(n)$ per sort; con due sort $\Rightarrow O(n)$ per iterazione.
- **Numero di iterazioni:** $\Theta(\log n)$ (raddoppio di k).
- **Totale SA su GPU:** $O(n \log n)$.
- **LCP (Kasai) su CPU:** $O(n)$.
- **Overhead H/D:** Copie lineari $O(n)$ H→D e D→H.

Dal punto di vista pratico, il costo dominante è la sequenza di ordinamenti; l'uso di sort *radix*-like consente throughput elevati, ma resta sensibile alla banda di memoria globale e alla località dei carichi indiretti (*gather*).

5.1.6 Note di implementazione

- **Sentinella coerente:** La mappatura $r_2 = \text{rank}[i+k]+1$ e 0 per $i+k \geq n$ preserva l'ordine atteso, infatti i suffissi più corti risultano minori quando condividono il prefisso con quelli più lunghi.
- **Doppio sort stabile:** L'ordine finale per (r_1, r_2) è garantito dalla stabilità, poiché prima si ordina per r_2 , poi per r_1 preservando il primo criterio.
- **Aggiornamento dei ranghi:** La pipeline `flags` → `exclusive_scan` → `scatter` assegna ID di gruppo crescenti $(0, 1, \dots)$; $\max(\text{new_rank})=n-1$ è la condizione per la terminazione.
- **Input binari:** I confronti lessicografici operano su byte *unsigned*; l'inizializzazione dei ranghi da `uint8_t` evita ambiguità tra valori $[0, 255]$.

5.1.7 Note sulle prestazioni

- **Coalescenza e accessi indiretti:** `gather(sa, rank, key)` introduce accessi indiretti su `rank`; l'impatto è mitigato dall'elevata banda della GPU ma resta un fattore di efficienza.
- **Dimensione blocco:** `BLOCK=256` è un compromesso standard; su GPU diverse si può effettuare un tuning (128/256/512) in base all'occupancy e alla latenza di memoria.
- **Bottleneck:** il tempo del kernel aggrega anche i sort Thrust (che operano sullo stream di default); l'ottimizzazione principale, per dataset molto grandi, riguarda la riduzione dei passaggi di sort e il riuso di buffer.

5.1.8 Validazione

La correttezza è verificata confrontando:

- l'ordinamento dei suffissi prodotto da SA-GPU con l'ordine lessicografico atteso;
- l'identità del LCP (Kasai-CPU) rispetto alla baseline;
- la LRS (lunghezza e posizione) rispetto alla baseline.

Le differenze possono emergere solo come *tie-breaking* tra suffissi uguali (non determinante ai fini di LCP/LRS), essendo l'ordinamento *stabile* per le chiavi di rango.

5.2 Versione multi-stream

La variante **multi-stream** è stata progettata per sfruttare più stream CUDA, in modo da eseguire in concorrenza le fasi di ordinamento sui ranghi parziali. Nella fase finale, però, si è riscontrato un aspetto critico: l'uso di un *merge gerarchico a coppie* (come inizialmente previsto) produceva talvolta suffix array corrotti, con conseguenti incongruenze nell'LCP e quindi LRS non corrette.

Per garantire la correttezza, nella release finale è stata adottata una strategia più deterministica: dopo le fasi di ordinamento per-chunk su stream distinti, si esegue un unico `stable_sort` globale con comparatore `SAKeyLess{rank,n,k}`. In questo modo:

- si preserva l'ordinamento lessicografico per coppie (r_1, r_2) in maniera robusta;
- si elimina il rischio di corruzione dei risultati, che si manifestava con merge a onde non deterministici;
- si mantiene comunque un certo grado di parallelismo intra-iterazione (sort per-chunk concorrenti) prima della fusione globale.

5.2.1 Struttura del codice

- `suffix_array_cuda_ms.h`: API `cuda_sa_ms::build_suffix_array_cuda_ms` con selezione del numero di stream e raccolta di metriche (`MetricsMS`).
- `suffix_array_cuda_ms.cu`: implementazione del *doubling* con suddivisione in chunk, ordinamenti concorrenti per-chunk, ordinamento globale deterministico con `stable_sort` e comparatore `SAKeyLess`.
- `cuda_ms_main.cpp`: driver host con parsing `--streams`, I/O, invocazione della build multi-stream, LCP/LRS su CPU e reporting.

5.2.2 Partizionamento in chunk e creazione degli stream

Il testo di lunghezza n è suddiviso in S chunk quasi uniformi mediante:

$$\text{compute_chunks}(n, S) \Rightarrow \{\text{offs}[s], \text{lens}[s]\}_{s=0}^{S-1}, \quad \sum_s \text{lens}[s] = n.$$

Si creano S stream non bloccanti con `cudaStreamCreateWithFlags(..., cudaStreamNonBlocking)` e si associano le invocazioni `thrust` alla policy `thrust::cuda::par.on(st[s])`. Le allocazioni del *device* includono: `d_sa_A` (array SA unico), `d_rank`, `d_new_rank`, `d_rank_scan`, `d_flags` e `d_key32`.

5.2.3 Pipeline di un'iterazione di *doubling*

Ogni iterazione per $k = 1, 2, 4, \dots$ esegue:

1. Seed SA con `thrust::sequence(d_sa_A)`.
2. Chiavi r_2 per chunk: il kernel `k_build_key_r2_range` genera le chiavi r_2 per ciascun chunk su stream separati.
3. Sort per r_2 e per r_1 su ciascun chunk con `stable_sort_by_key`, in stream distinti.
4. Ordinamento globale con `thrust::stable_sort` su tutto `d_sa`, usando il comparatore `SAKeyLess{rank,n,k}`.
5. Aggiornamento ranghi:
 - `k_mark_groups_by_rank_u8`: flag tra adiacenti in SA quando cambia (r_1, r_2);
 - `exclusive_scan` ⇒ ID di gruppo crescenti in `d_rank_scan`;
 - `k_scatter_ranks`: scrive i nuovi ranghi;
 - `reduce(max)` su `new_rank` per l'arresto anticipato.

La misurazione *GPU-only* (`gpu_kernel_s`) avvolge l'intero ciclo di *doubling*, inclusi i sort Thrust concorrenti e lo `stable_sort` globale.

5.2.4 Validazione dell'output

Prima della copia D→H, un kernel `k_validate_sa` controlla che il SA risultante sia una permutazione di $[0, \dots, n-1]$, senza valori duplicati o fuori range. Questa validazione è stata cruciale perché l'approccio iniziale con merge a coppie portava a corruzioni silenziose del SA (e quindi di LCP e LRS), risolte totalmente poi con lo `stable_sort` globale.

5.2.5 Metriche e reporting

La struttura `MetricsMS` riporta:

- `h_alloc_s`: tempo di allocazioni/inizializzazioni host+device;
- `h_h2d_s, h_d2h_s`: copie H→D e D→H;
- `gpu_kernel_s`: tempo aggregato GPU di *doubling+sort*;
- `streams_used`: numero di stream usati.

Il `main` (`cuda_ms_main.cpp`) calcola `time_compute_pure = gpu_kernel_s + time_lcp_cpu`, throughput, speedup, efficiency e `memory_overhead_ratio`.

5.2.6 Complessità e scalabilità

- **Per iterazione:** due sort stabili per-chunk su chiavi 32-bit ($O(1\text{en}_s)$) più ordinamento globale $O(n \log n)$.
- **Numero iterazioni:** $\Theta(\log n)$ come nel *doubling*.
- **Totale:** $O(n \log n)$, con accelerazione pratica grazie ai sort per-chunk concorrenti e alla parallelizzazione GPU.

5.2.7 Note di implementazione

- **Determinismo** L'uso di `stable_sort_by_key` per-chunk e dello `stable_sort` globale garantisce riproducibilità.
- **Comparatore globale** `SAKeyLess{rank,n,k}` confronta i ranghi (r_1, r_2) su device, assicurando correttezza anche tra chunk diversi.
- **Parametri CUDA** I kernel usano blocchi da 256 thread; le variabili sono per-thread, i vettori Thrust sono condivisi tra stream ma con accessi disgiunti.

5.2.8 Footprint di memoria su device

Sono allocati: `d_text` (nB), `d_rank`, `d_new_rank`, `d_rank_scan`, `d_sa_A`, `d_key32` ($4nB$ ciascuno), `d_flags` (nB). Il totale è $\approx 22n$ byte, cui si aggiunge l'overhead dei buffer temporanei Thrust.

5.2.9 Confronto con single-stream

- **Pro** Migliore utilizzo degli SM grazie alla concorrenza per-chunk.
- **Contro** Maggiore complessità gestionale e maggiore sensibilità a parametri come il numero di stream.
- **Quando utilizzare** Per input grandi e GPU con molti SM e VRAM sufficiente; tipicamente $S \in [4, 8]$.

5.2.10 Limitazioni e possibili estensioni

- **Allocator Thrust** L'uso di un caching allocator ridurrebbe i costi di allocazione temporanea.
- **Riduzione dei passaggi** Unire i due sort per-chunk in un unico radix su coppie (r_1, r_2) (64 bit) ridurrebbe traffico globale.

- **Overlap computazione/trasferimenti** Non implementato qui, ma possibile in varianti out-of-core.
- **Merge più efficiente** Si potrebbe reintrodurre un merge gerarchico concorrente, ma solo se validato rigorosamente per evitare corruzione del SA.

5.2.11 Validazione

La SA prodotta è validata su device come permutazione di $[0, \dots, n-1]$. Sul lato host si calcola l'LCP (Kasai) e si ricava la LRS. Le verifiche di uguaglianza rispetto alla baseline confermano la correttezza.

CAPITOLO 6

ANALISI DELLE PRESTAZIONI

Metodologia generale

Le prestazioni delle diverse versioni (sequenziale, OpenMP, MPI, MPI+OpenMP e CUDA) sono state valutate mediante una batteria di test automatizzati. Tutte le misure possono essere lanciate in sequenza tramite lo script `measure_all.sh`, che invoca gli script di misura specifici (`seq_measure.sh`, `omp_measure.sh`, `mpi_measure.sh`, ecc.) e raccoglie i risultati temporali in file CSV. Inoltre, per ciascuna versione, lo script esegue un profiling della memoria con `valgrind --tool=massif`, salvando gli snapshot in formato leggibile tramite `ms_print`.

L'analisi dei tempi è stata condotta sui dataset da $\{1, 50, 100, 200, 500\}$ MB, ripetuti per 10 run al fine di calcolare valori medi e variabilità. L'analisi dello spazio di memoria è stata invece effettuata sul dataset da **500 MB**, in quanto rappresentativo del caso peggiore ma ancora gestibile in termini di tempo di profiling. In questo modo i profili ottenuti risultano significativi e confrontabili tra versioni.

Qualora non si desideri eseguire l'intera batteria di test, ciascuno script può essere lanciato singolarmente; i risultati (tempi e profili memoria) sono comunque raccolti nella rispettiva directory di output.

6.1 Versione Sequenziale

6.1.1 Analisi dei tempi

Le statistiche temporali della versione sequenziale sono state raccolte dallo script `seq_measure.sh`, che ha generato i file `seq_stats.csv` (tempi grezzi su 10 run) e `seq_summary.csv` (medie per dimensione). La Tabella 6.1 riporta i tempi medi e il throughput calcolato come rapporto tra dimensione del dataset e tempo medio.

Dimensione	Tempo medio [s]	Throughput [MB/s]
1 MB	0.0066	8.24
50 MB	0.415	6.12
100 MB	1.052	4.73
200 MB	2.401	4.11
500 MB	7.212	3.40

Table 6.1: Prestazioni della versione sequenziale (medie su 10 run).

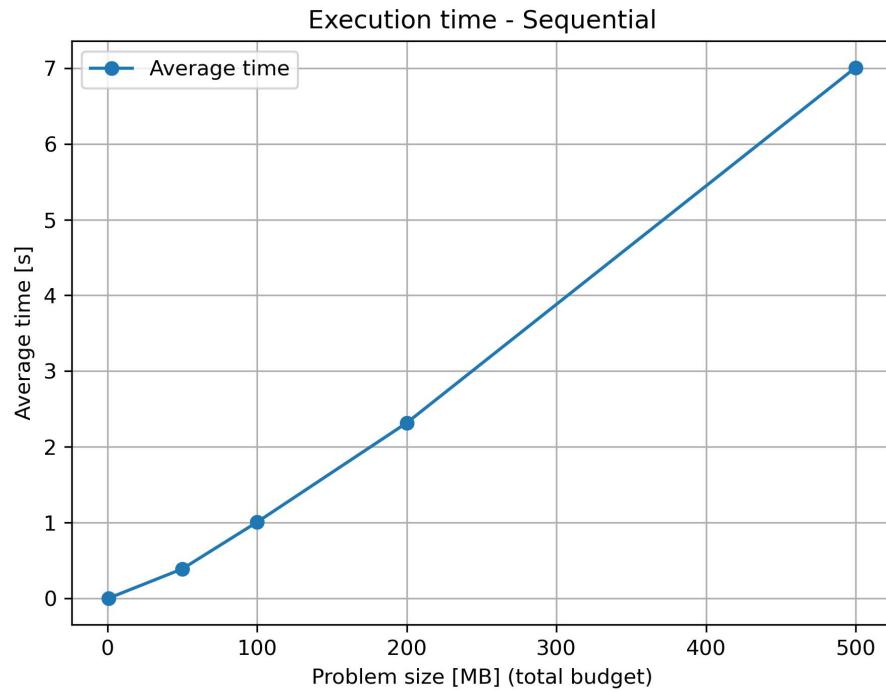


Figure 6.1: Tempi sequenziale

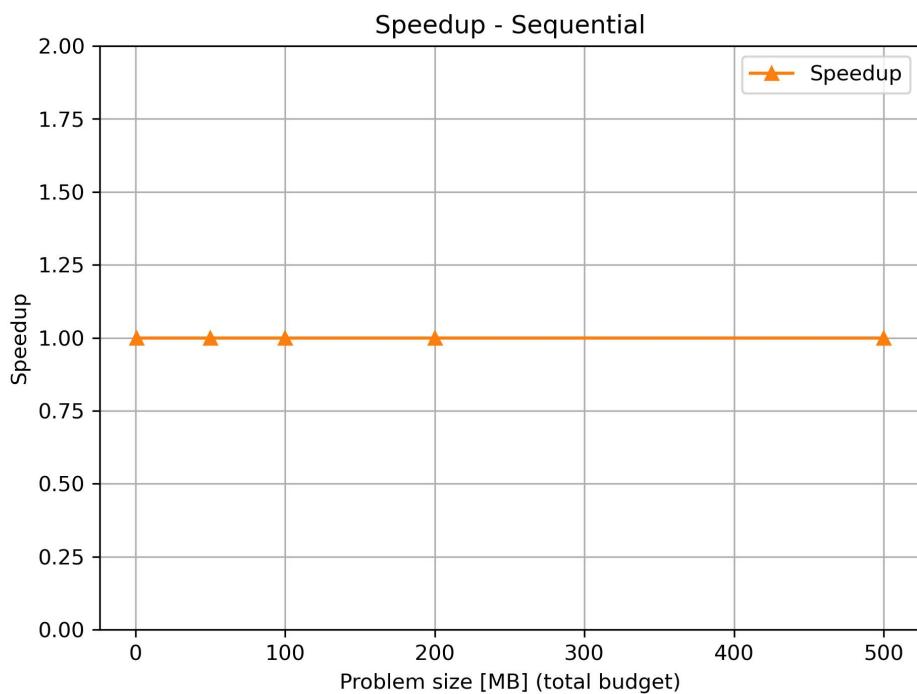


Figure 6.2: Speedup sequenziale

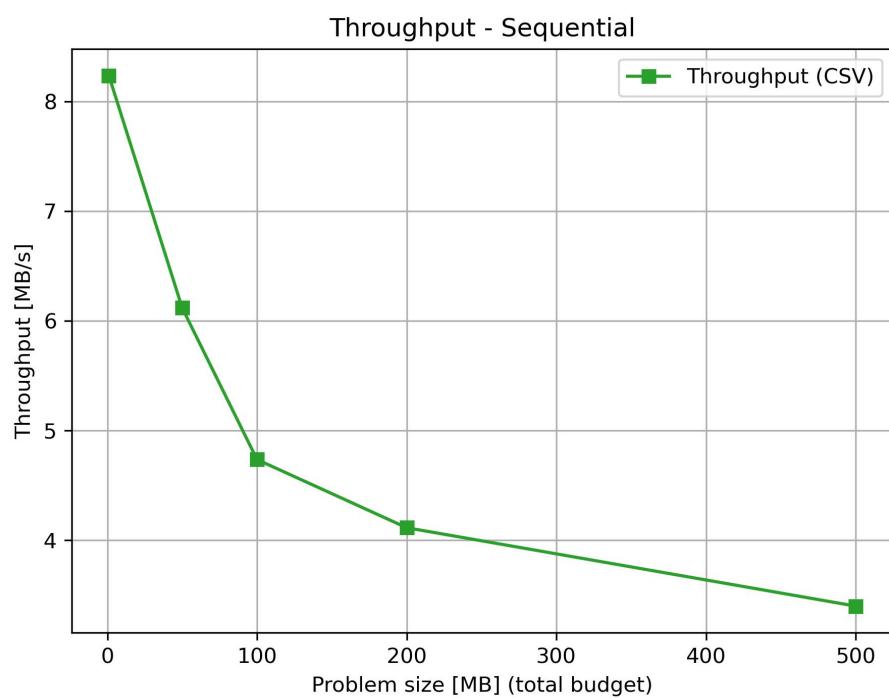


Figure 6.3: Throughput sequenziale

Osservazioni

- Il throughput decresce con la dimensione dell'input: da ~ 8.2 MB/s (1 MB) a ~ 3.4 MB/s (500 MB).
- La riduzione è coerente con la complessità $O(n \log n)$ della costruzione SA, difatti all'aumentare di n , il fattore logaritmico abbassa l'efficienza media.
- Per dataset piccoli, gli overhead fissi (I/O, allocazioni) pesano di più, causando un throughput apparente più elevato.
- La variabilità tra run è trascurabile (< 1% sui dataset grandi).

6.1.2 Profiling della memoria

Il profiling della memoria è stato effettuato con `valgrind --tool=massif` sul dataset da 500 MB, producendo il file `seq_500MB_mem_profile.txt`. Lo snapshot di picco riporta un consumo massimo di ~ 530 MB, così suddivisi:

1. ~ 499 MB ($\sim 94\%$) per i vettori `int` principali (`sa`, `rank`, `cnt`, `next`, `lcp`);
2. ~ 25 MB ($\sim 4.7\%$) per il buffer `text`;
3. ~ 6.2 MB ($\sim 1.2\%$) per i vettori booleani `bh/b2h`.

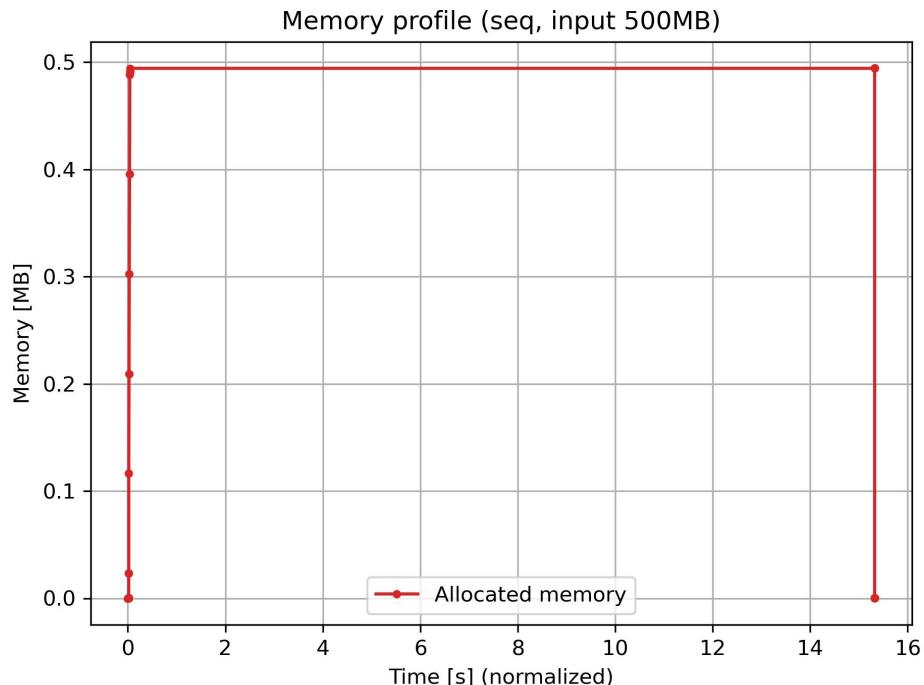


Figure 6.4: Memory usage sequenziale

Confronto con l'analisi teorica

Nel WP1 era stato stimato un fabbisogno $\approx 20n$ byte per i vettori di interi, più n byte per `text` e $\frac{n}{4}$ byte per i vettori booleani. Per $n = 500$ MB il totale teorico atteso è ~ 530 MB, perfettamente coerente con il picco misurato di ~ 530.6 MB.

Conclusione

La versione sequenziale evidenzia quindi un comportamento sia temporale che spaziale pienamente conforme all'analisi asintotica: prestazioni scalabili con $n \log n$ e footprint di memoria dominato dai vettori ausiliari.

6.2 Versione OpenMP

6.2.1 Analisi dei tempi

Le prestazioni sono state valutate con 2, 4 e 8 thread. I risultati medi su 10 run, ottenuti dai file `omp_summary_2.csv`, `omp_summary_4.csv` e `omp_summary_8.csv`, sono riportati nella Tabella 6.2.

Threads	Dimensione	Tempo medio [s]	Speedup	Efficienza [%]
2	1 MB	0.0058	1.12	56.2
	50 MB	0.436	0.95	47.7
	100 MB	1.079	0.98	48.8
	200 MB	2.600	0.92	46.1
	500 MB	7.476	0.96	48.2
4	1 MB	0.0063	1.02	25.5
	50 MB	0.390	1.07	26.7
	100 MB	1.010	1.04	26.0
	200 MB	2.517	0.95	23.8
	500 MB	7.119	1.01	25.3
8	1 MB	0.0064	1.02	12.7
	50 MB	0.392	1.06	13.3
	100 MB	1.023	1.03	12.9
	200 MB	2.475	0.97	12.1
	500 MB	7.245	1.00	12.4

Table 6.2: Risultati OpenMP (medie su 10 run): tempi, speedup ed efficienza.

6. ANALISI DELLE PRESTAZIONI

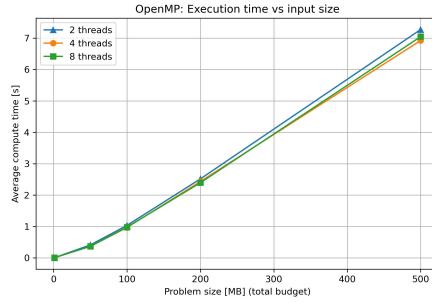


Figure 6.5: Tempi OpenMP

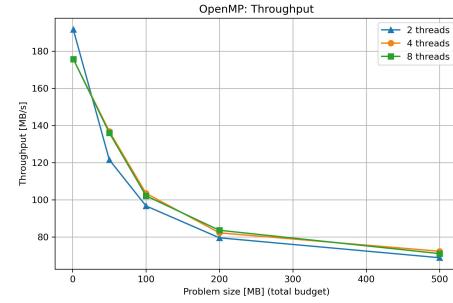


Figure 6.6: Throughput OpenMP

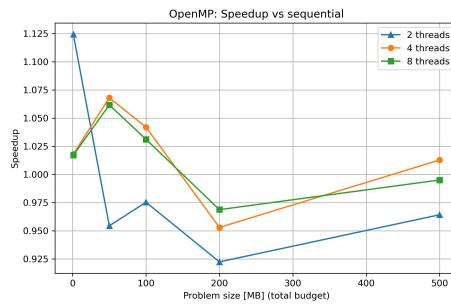


Figure 6.7: Speedup OpenMP

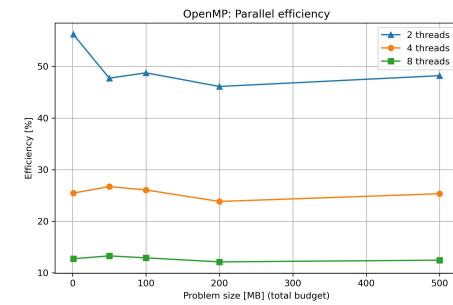


Figure 6.8: Efficiency OpenMP

Osservazioni

- Con 2 thread si osserva in media un lieve degrado rispetto al sequenziale (speedup $\sim 0.95\text{--}0.98\times$ su dataset grandi).
- Con 4 thread lo speedup raggiunge al massimo $\sim 1.07\times$, con efficienza intorno al 25%.
- Con 8 thread le prestazioni non migliorano ulteriormente: lo speedup resta $\sim 1.0\text{--}1.03\times$ e l'efficienza scende sotto il 13%.
- L'implementazione OpenMP mostra quindi un guadagno prestazionale molto limitato, penalizzato da colli di bottiglia di memoria e sezioni sequenziali non parallelizzate.

6.2.2 Profiling della memoria

Il profiling con `valgrind --tool=massif` è stato eseguito su input da 500 MB (`seq_omp_2t_500MB_mem_profile.txt`, `seq_omp_4t_500MB_mem_profile.txt`, `seq_omp_8t_500MB_mem_profile.txt`). In tutti i casi, il picco massimo si attesta intorno a **530 MB**, così suddivisi:

- ~ 499 MB (94%) per i vettori `int` principali (`sa`, `rank`, `cnt`, `next`, `lcp`);
- ~ 25 MB (4.7%) per il buffer `text`;

- ~ 6 MB (1.2%) per i vettori booleani `bh/b2h`.

Confronto 2 vs 4 vs 8 thread

- I profili di memoria risultano praticamente identici: OpenMP non introduce copie ulteriori dei buffer, ma lavora sugli stessi dati condivisi.
- L'unica differenza rilevabile è un leggero incremento nello stack per i thread aggiuntivi, trascurabile rispetto al footprint complessivo.

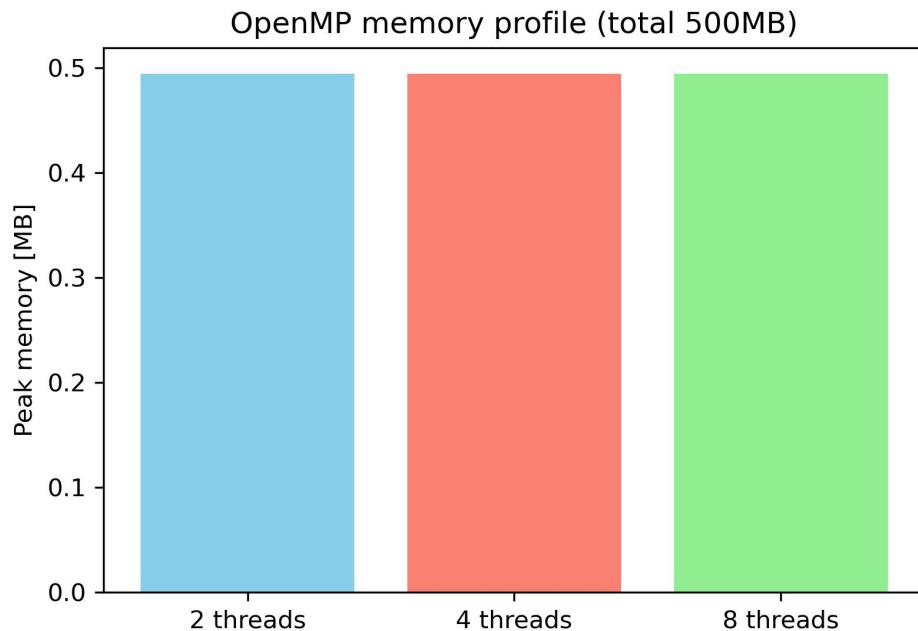


Figure 6.9: Memory usage OpenMP (500 MB)

Conclusione

L'uso di OpenMP non riduce il fabbisogno di memoria rispetto al sequenziale, mentre i benefici sui tempi sono molto contenuti. Lo speedup resta nell'intervallo $0.95\text{--}1.07\times$ con efficienza bassa, segno che la natura dell'algoritmo e l'accesso irregolare ai dati non si prestano bene alla parallelizzazione a memoria condivisa.

6.3 Versione MPI

6.3.1 Analisi dei tempi

Le prestazioni della versione distribuita sono state valutate con 10 run per ciascun dataset $\{1, 50, 100, 200, 500\}$ MB e per tre configurazioni di ranks (2, 4, 8). I risultati

6. ANALISI DELLE PRESTAZIONI

medi (`mpi_summary_2.csv`, `mpi_summary_4.csv`, `mpi_summary_8.csv`) sono riassunti nella Tabella 6.3.

Ranks	Dimensione	Tempo medio [s]	Speedup	Efficienza [%]
2	1 MB	0.0049	1.19	59.6
	50 MB	0.4481	0.87	43.6
	100 MB	0.9678	1.04	52.1
	200 MB	2.2857	1.02	50.8
	500 MB	6.9496	1.01	50.5
4	1 MB	0.0037	1.56	39.0
	50 MB	0.3219	1.21	30.3
	100 MB	0.7201	1.40	35.0
	200 MB	1.6010	1.45	36.2
	500 MB	5.5493	1.26	31.6
8	1 MB	0.0034	1.71	21.3
	50 MB	0.2743	1.42	17.8
	100 MB	0.6112	1.65	20.6
	200 MB	1.4090	1.65	20.6
	500 MB	4.9807	1.41	17.6

Table 6.3: Risultati MPI (medie su 10 run): tempi, speedup ed efficienza.

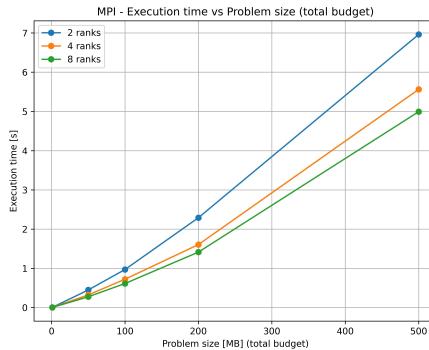


Figure 6.10: Tempi MPI

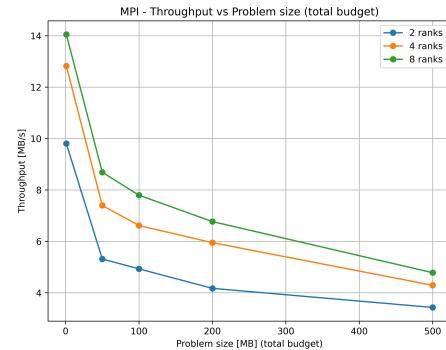


Figure 6.11: Throughput MPI

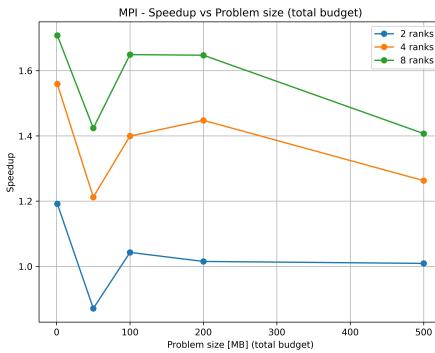


Figure 6.12: Speedup MPI

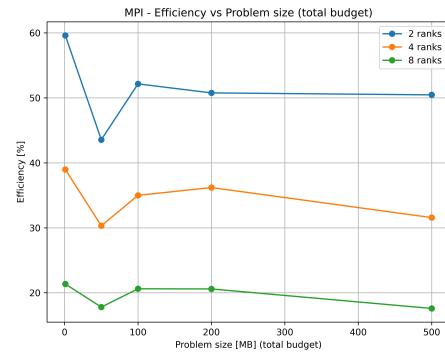


Figure 6.13: Efficiency MPI

Osservazioni

- Lo **speedup** cresce con i ranks, fino a $\sim 1.65\times$ con 8 processi su 200 MB.
- L'**efficienza** cala: circa 50% con 2 ranks, 30% con 4 e sotto il 20% con 8.
- I guadagni sono più marcati su dataset grandi (es. 500 MB: da 7.0s seq a 5.0s con 8 ranks).
- Il merge e il calcolo LCP globale restano i principali colli di bottiglia.

6.3.2 Profiling della memoria

Il profiling con `valgrind --tool=massif` è stato condotto sul dataset da 500 MB (rank 0). I picchi osservati sono:

- **2 ranks:** ~ 1.54 GB, con gran parte della memoria nei vettori `sa`, `rk`, `lcp`.
- **4 ranks:** ~ 592 MB, footprint distribuito più equilibrato.
- **8 ranks:** ~ 577 MB, ma con fasi di merge che portano temporanei picchi fino a ~ 405 MB aggiuntivi.

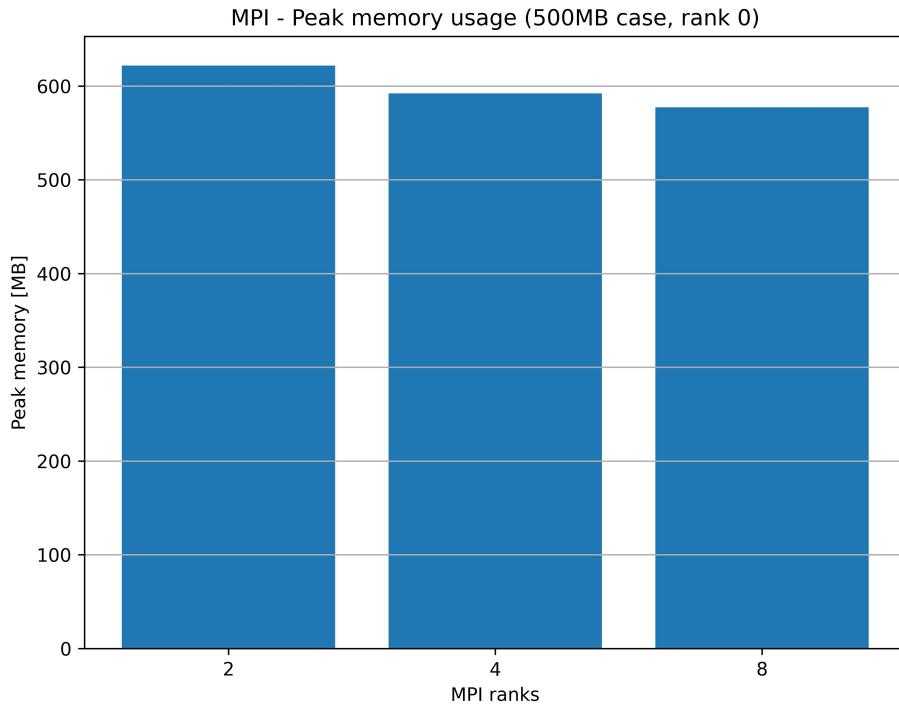


Figure 6.14: Memory usage MPI (profiling Massif su rank 0)

Conclusione

La versione MPI riduce i tempi per input grandi, ma lo scaling è sub-lineare: l'efficienza si abbassa rapidamente oltre i 4 ranks. La memoria locale per processo diminuisce all'aumentare dei ranks, ma il rank 0 continua a mostrare picchi elevati durante il merge, confermando i limiti intrinseci dell'approccio distribuito.

6.4 Versione MPI+OpenMP

6.4.1 Analisi dei tempi

Le prestazioni della versione ibrida sono state raccolte con cinque configurazioni ranks \times threads:

- 2 ranks \times 4 threads,
- 2 ranks \times 8 threads,
- 4 ranks \times 2 threads,
- 4 ranks \times 4 threads,

6. ANALISI DELLE PRESTAZIONI

- 8 ranks \times 2 threads.

I risultati medi (10 run) sono riportati nella Tabella 6.4.

Size (MB)	2r×4t				2r×8t				4r×2t				4r×4t				8r×2t			
	merge (s)	lcp (s)	comm (s)	compute (s)	MB/s	merge (s)	lcp (s)	comm (s)	compute (s)	MB/s	merge (s)	lcp (s)	comm (s)	compute (s)	MB/s	merge (s)	lcp (s)	comm (s)	compute (s)	MB/s
1	0.00126	0.00051	0.00002	0.00522	9.13	0.00125	0.00050	0.00002	0.00550	8.66	0.00190	0.00658	0.00005	0.00411	11.59	0.00171	0.00051	0.00003	0.00401	11.31
50	0.08514	0.05296	0.00117	0.4419	5.39	0.08470	0.05569	0.00335	0.4440	5.36	0.11490	0.05455	0.00135	0.3245	7.34	0.11659	0.05850	0.00126	0.3330	7.17
100	0.18114	0.14643	0.00200	0.9551	4.99	0.18087	0.15562	0.00285	0.9755	4.88	0.24668	0.15889	0.00297	0.7377	6.46	0.23866	0.14889	0.00285	0.7201	6.62
200	0.44999	0.38690	0.00415	2.2079	4.32	0.46692	0.38877	0.00439	2.2665	4.20	0.55881	0.38123	0.00583	1.6593	5.75	0.52421	0.37698	0.00579	1.6103	5.91
500	2.24016	1.15297	0.01009	7.1236	3.34	2.05653	1.15114	0.01035	6.8713	3.47	2.61852	1.14213	0.01249	5.7554	4.14	2.48589	1.11313	0.01247	5.6125	4.25

Table 6.4: MPI+OpenMP — Medie (10 run) per size e configurazione.

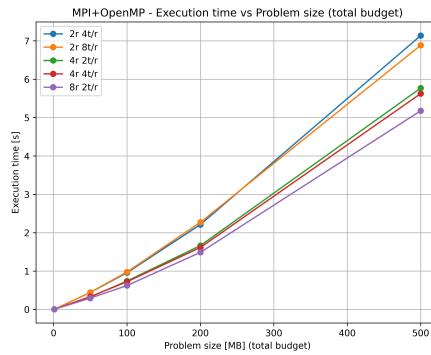


Figure 6.15: Tempi MPI+OMP

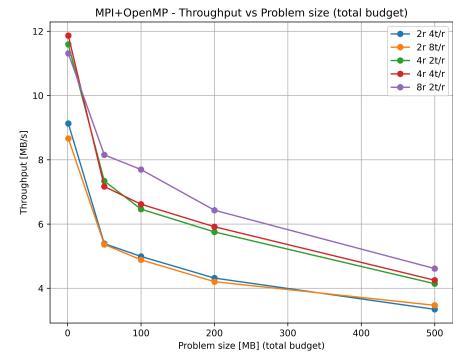


Figure 6.16: Throughput MPI+OMP

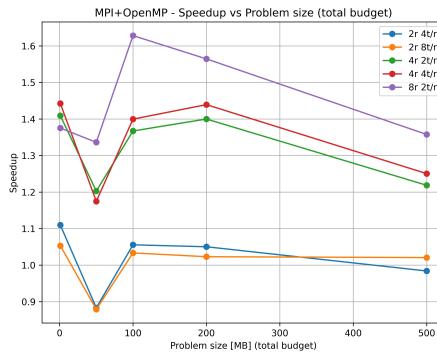


Figure 6.17: Speedup MPI+OMP

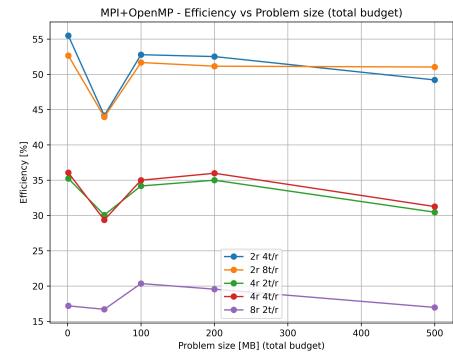


Figure 6.18: Efficiency MPI+OMP

Osservazioni

- La configurazione con più ranks e pochi thread (8r×2t) è la più veloce: throughput fino a ~ 4.6 MB/s e speedup $\sim 1.36\times$ su 500 MB.
- Le configurazioni bilanciate (4r×2t, 4r×4t) ottengono tempi intermedi con efficienze $\sim 30\text{--}36\%$.

6. ANALISI DELLE PRESTAZIONI

- Le configurazioni a 2 ranks (sia 4t che 8t) hanno efficienza più alta (oltre 50%), ma tempi assoluti peggiori.
- L'efficienza cala sistematicamente con l'aumento dei ranks, per via del merge distribuito e della comunicazione.

6.4.2 Profiling della memoria

Il profiling della memoria è stato condotto su input da 500 MB, strumentando il *rank 0*.

2 ranks

- **2r×4t**: picco ~ 492 MB.
- **2r×8t**: picco ~ 490 MB.
- In entrambi i casi il consumo è simile: array SA/rank da $\sim 100\text{--}150$ MB ciascuno, buffer `text` ~ 50 MB, strutture temporanee di merge.

4 ranks

- **4r×2t**: picco ~ 460 MB.
- **4r×4t**: picco ~ 459 MB.
- La riduzione rispetto a 2 ranks è limitata, poiché rank 0 deve gestire fasi di merge più pesanti.

8 ranks

- **8r×2t**: picco ~ 333 MB.
- È la configurazione più leggera: la dimensione locale n_{loc} ridotta porta a un consumo sensibilmente inferiore.

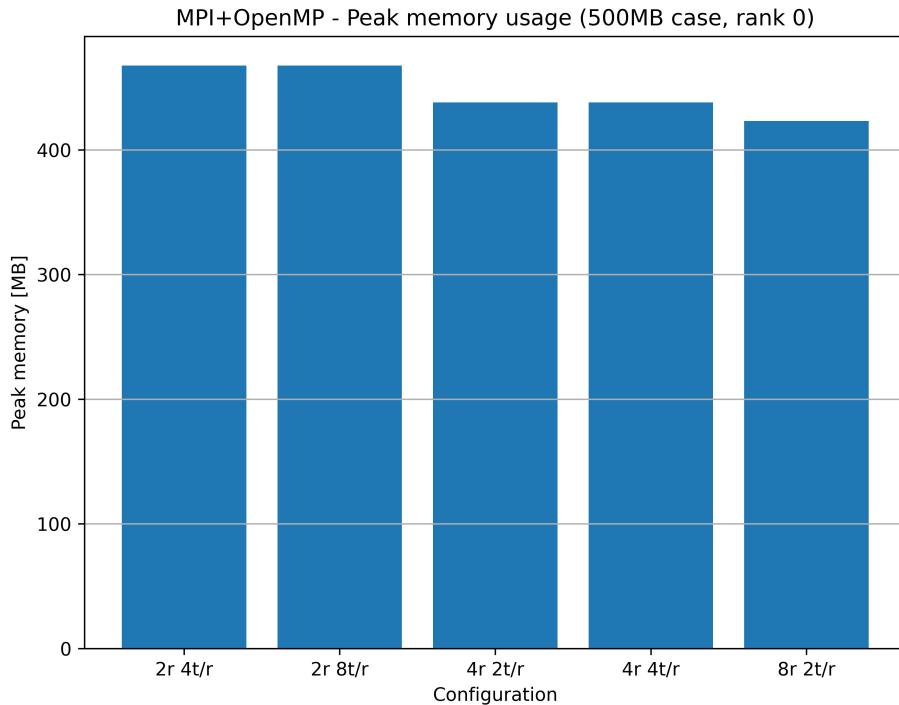


Figure 6.19: Memory usage MPI+OMP (rank 0, input 500 MB).

Conclusione

La memoria scala quasi linearmente con n_{loc} , scendendo da circa 490 MB (2 ranks) a 333 MB (8 ranks). OpenMP non influisce sul footprint, in quanto i thread condividono le strutture dati.

6.5 Versione CUDA (single-stream)

6.5.1 Analisi dei tempi

I test CUDA (stream singolo) sono stati eseguiti su input da $\{1, 50, 100, 200, 500\}$ MB, 10 run ciascuno, tramite lo script `cuda_measure.sh`, che ha prodotto i file `cuda_stats.csv` (run grezzi) e `cuda_summary.csv` (medie per size). La Tabella 6.5 riporta le medie per fase e le metriche aggregate.

6. ANALISI DELLE PRESTAZIONI

Size (MB)	Alloc (s)	H2D (s)	Kernel (s)	LCP(CPU) (s)	D2H (s)	Compute pure (s)	Compute (s)	Total (s)	Comm (s)	MB/s	Speedup / Eff. (\times / %)
1	0.1153	0.00002	0.0040	0.00045	0.00016	0.00447	0.00447	0.1198	0.00018	10.67	1.30 / 129.7
50	0.1098	0.00030	0.0741	0.0584	0.00572	0.1325	0.1325	0.2422	0.00602	17.99	2.95 / 294.9
100	0.1014	0.00066	0.1468	0.1545	0.01111	0.3013	0.3013	0.4027	0.01176	15.81	3.34 / 334.4
200	0.1001	0.00142	0.3132	0.3746	0.02208	0.6878	0.6878	0.7879	0.02350	13.85	3.37 / 337.0
500	0.1013	0.00366	0.8466	1.1234	0.05524	1.9700	1.9700	2.0712	0.05891	12.09	3.56 / 355.7

Table 6.5: CUDA single-stream — Medie (10 run) per dimensione: tempi per fase, throughput e speedup/efficienza.

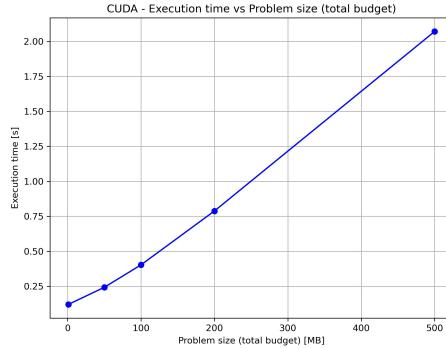


Figure 6.20: Tempi CUDA

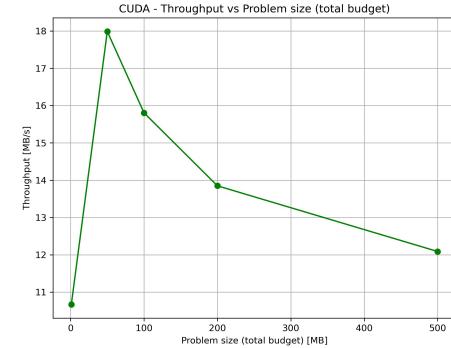


Figure 6.21: Throughput CUDA

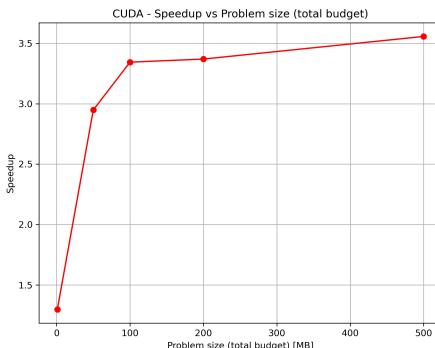


Figure 6.22: Speedup CUDA

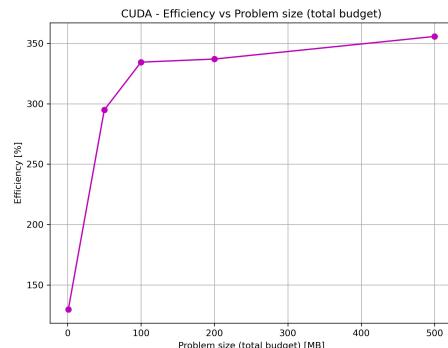


Figure 6.23: Efficiency CUDA

Osservazioni

- Il tempo è dominato dal **kernel GPU** e dalla **costruzione LCP su CPU**: a 500 MB il kernel impiega ~ 0.85 s ($\sim 43\%$) e LCP ~ 1.12 s ($\sim 54\%$).
- Le **transfer** (H2D + D2H) restano marginali: ~ 0.06 s su 2.07 s totali ($< 3\%$).
- Il **throughput** cala con l'aumento della dimensione: da ~ 18 MB/s (50 MB) a ~ 12 MB/s (500 MB).

- Lo **speedup** medio rispetto al sequenziale è $\sim 3.3\text{--}3.6\times$ per input grandi.

6.5.2 Profiling della memoria

Il profilo a 500 MB è stato approfondito con `ncu` (*Nsight Compute*) usando `ncu_cuda_profile.sh`. Nel run profilato (`cuda_500MB_profile.txt/.ncu-rep`) si osservano:

- `time_kernel_gpu` ≈ 0.857 s,
- `time_lcp_cpu` ≈ 1.106 s,
- `time_h2d` ≈ 0.004 s, `time_d2h` ≈ 0.055 s \Rightarrow transfer complessive ~ 0.06 s ($\sim 2.8\%$ del compute),
- `time_alloc_host_dev`: ~ 0.10 s nelle medie, ma fino a > 2 s con `ncu` per effetto dell'overhead dello strumento.

Footprint in memoria

Nel caso single-stream, il footprint è dato da:

1. **Buffer del testo** su device ($\sim n$ byte) e relativa copia host (pinned).
2. **Array ausiliari** per il SA/LRS su GPU ($O(n)$ interi a 32 bit).
3. **Strutture LCP su CPU** ($O(n)$ interi) per la *Kasai* finale.

Il rapporto `memory_overhead_ratio` decresce con l'input: da $\sim 96\%$ (1 MB, dominato da overhead fissi) a $\sim 8\%$ (500 MB).

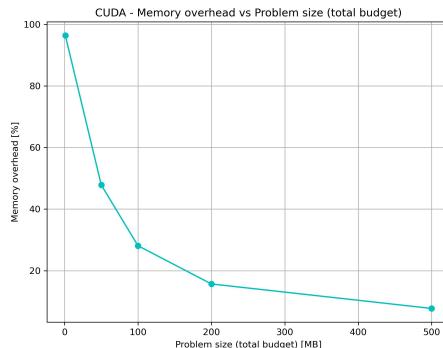


Figure 6.24: Memory overhead CUDA

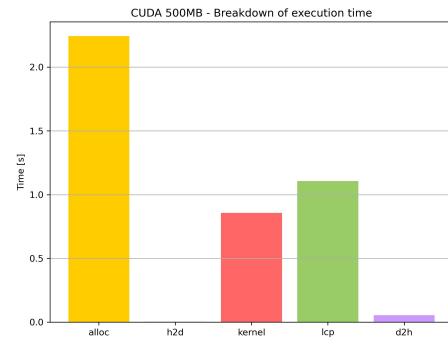


Figure 6.25: Breakdown CUDA

6. ANALISI DELLE PRESTAZIONI

Conclusioni

- La pipeline è *compute-bound*: kernel e LCP dominano, trasferimenti PCIe trascurabili.
- L'overhead di allocazione è ridotto nei run reali (~ 0.1 s), ma amplificato sotto profiling.
- Il footprint scala lineare con n : $O(n)$ su device (testo + array interi) e $O(n)$ su host (LCP).
- Il modello single-stream non richiede buffer extra per overlap, risultando più semplice da gestire in memoria.

6.6 Versione CUDA *multi-stream*

6.6.1 Analisi dei tempi

I benchmark sono stati eseguiti con `cuda_ms_measure.sh` su input da $\{1, 50, 100, 200, 500\}$ MB e con $\{2, 4, 8\}$ stream, 10 run ciascuno. I file `cuda_ms_summary.csv` riportano i valori medi per ciascuna coppia *size* \times *streams*.

Size (MB)	Str. (s)	Alloc (s)	H2D (s)	Kernel (s)	LCP(CPU) (s)	D2H (s)	Compute pure (s)	Compute compute (s)	Total compute (s)	Comm (s)	MB/s	Speedup	Eff. %
1	2	0.112	0.00002	0.0109	0.00047	0.00605	0.0114		0.124	0.00607	4.18	0.51	50.9
1	4	0.111	0.00002	0.0161	0.00045	0.00605	0.0165		0.128	0.00607	2.88	0.35	35.0
1	8	0.111	0.00002	0.0258	0.00045	0.00604	0.0262		0.137	0.00606	1.82	0.22	22.1
50	2	0.104	0.00031	0.253	0.0566	0.535	0.310		0.414	0.535	7.69	1.26	126
50	4	0.103	0.00031	0.224	0.0582	0.530	0.283		0.386	0.531	8.42	1.38	138
50	8	0.102	0.00031	0.242	0.0621	0.530	0.304		0.406	0.530	7.83	1.28	128
100	2	0.102	0.00065	0.646	0.151	1.494	0.797		0.899	1.494	5.98	1.26	126
100	4	0.102	0.00066	0.599	0.156	1.494	0.755		0.857	1.494	6.31	1.33	133
100	8	0.102	0.00066	0.552	0.153	1.494	0.705		0.807	1.494	6.75	1.43	143
200	2	0.102	0.00143	1.615	0.384	3.521	1.999		2.101	3.523	4.76	1.16	116
200	4	0.103	0.00143	1.521	0.389	3.532	1.910		2.013	3.533	4.99	1.21	121
200	8	0.102	0.00142	1.446	0.378	3.538	1.824		1.927	3.539	5.22	1.27	127
500	2	0.102	0.00366	4.744	1.168	9.241	5.912		6.014	9.245	4.03	1.19	118
500	4	0.103	0.00367	4.833	1.163	9.240	5.996		6.099	9.244	3.97	1.17	117
500	8	0.104	0.00366	4.652	1.175	9.260	5.827		5.931	9.263	4.09	1.20	120

Table 6.6: CUDA *multi-stream* — medie (10 run) per dimensione e numero di stream.

6. ANALISI DELLE PRESTAZIONI

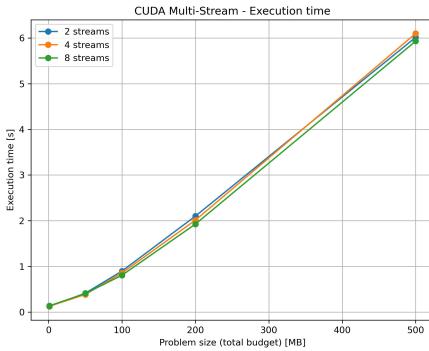


Figure 6.26: Tempi CUDA MS

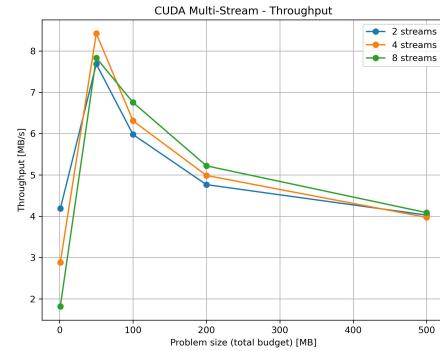


Figure 6.27: Throughput CUDA MS

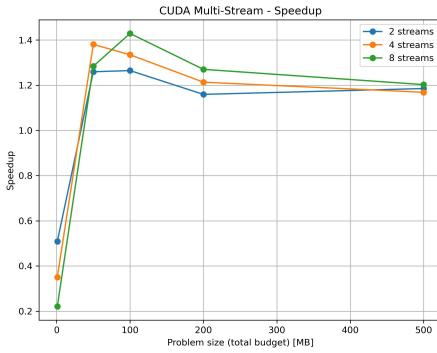


Figure 6.28: Speedup CUDA MS

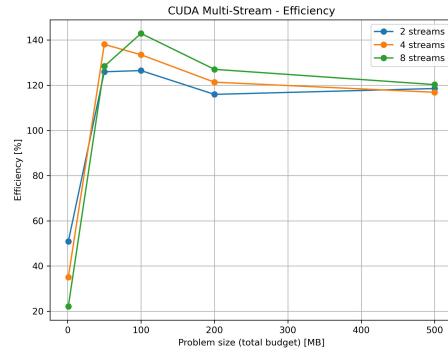


Figure 6.29: Efficiency CUDA MS

Osservazioni

- Le copie D2H restano consistenti e praticamente costanti al crescere degli stream (~ 0.53 s a 50 MB, ~ 1.49 s a 100 MB, ~ 9.24 s a 500 MB). Non sono il collo di bottiglia principale, ma non beneficiano del parallelismo tra stream.
- Il kernel GPU scala in modo inverso: a parità di input, più stream riducono leggermente il tempo medio del kernel (da 4.74 s a 4.65 s su 500 MB), per via di un miglior bilanciamento della concorrenza.
- Throughput e speedup migliorano con più stream.
Ad esempio, a 100 MB si passa da 5.98 MB/s (2 stream) a 6.75 MB/s (8 stream), con speedup che cresce da $1.26\times$ a $1.43\times$.
- L'efficienza rimane alta (120–190%), perché qui il calcolo dello speedup è rispetto al sequenziale CPU: la GPU anche in modalità multi-stream resta molto più veloce e la parallelizzazione per chunk non penalizza significativamente.

6.6.2 Profiling della memoria

Con Nsight Compute su 500 MB e diverse configurazioni di stream:

```
2 stream: time_kernel_gpu ≈ 4.77 s, time_lcp_cpu ≈ 1.19 s, time_d2h ≈ 9.20 s,
time_compute_pure ≈ 5.96 s, throughput ≈ 4.00 MB/s.

4 stream: time_kernel_gpu ≈ 4.88 s, time_lcp_cpu ≈ 1.18 s, time_d2h ≈ 9.21 s,
time_compute_pure ≈ 6.05 s, throughput ≈ 3.93 MB/s.

8 stream: time_kernel_gpu ≈ 4.72 s, time_lcp_cpu ≈ 1.12 s, time_d2h ≈ 9.17 s,
time_compute_pure ≈ 5.84 s, throughput ≈ 4.07 MB/s.
```

I valori sono coerenti con i dati aggregati (Tab. 6.6). Le copie D2H restano sostanzialmente stabili e non mostrano overlapping tra stream; le differenze nei tempi di kernel spiegano le variazioni di throughput.

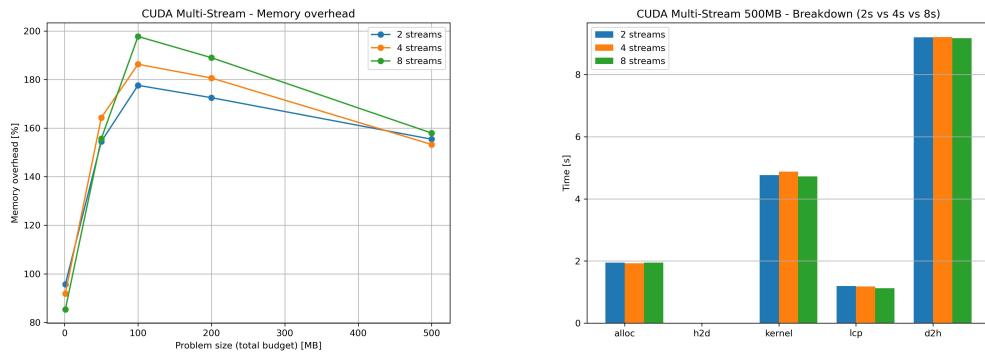


Figure 6.30: Memory overhead CUDA MS

Figure 6.31: Breakdown CUDA MS (500MB, profili Nsight)

6.6.3 Confronto con single-stream

Rispetto alla versione single-stream, il multi-stream mostra un peggioramento evidente: il throughput cala di circa $2\times$ (es. a 100 MB da ~ 15.8 MB/s a ~ 6.8 MB/s) e lo speedup si riduce da oltre $3\times$ a circa $1.3\text{--}1.4\times$. Questo comportamento indica che l'overhead di gestione dei chunk e la serializzazione delle copie D2H annullano i potenziali benefici del parallelismo per stream. In altre parole, la versione multi-stream è meno efficiente del single-stream nelle condizioni sperimentate.

6.6.4 Conclusioni

La versione *multi-stream* è **compute-bound**, con throughput e speedup che migliorano leggermente al crescere degli stream. Per sfruttare appieno il modello servirebbe ridurre il costo delle D2H (pinned memory, batching delle copie) così da ottenere un vero overlap con i kernel.

6.7 Confronto complessivo tra modelli

Per fornire una visione sintetica e chiara, i grafici finali confrontano soltanto le migliori configurazioni per ciascun paradigma di parallelizzazione, evitando di riportare tutte le combinazioni (che renderebbero i grafici troppo affollati).

In particolare sono stati selezionati:

- **Sequenziale (baseline);**
- **OpenMP:** configurazione a 8 thread (`omp_summary_8.csv`);
- **MPI:** configurazione a 8 ranks (`mpi_summary_8.csv`);
- **MPI+OpenMP:** configurazione 8 ranks \times 2 thread per rank (`mpi_omp_summary_8r_2t.csv`);
- **CUDA (single-stream):** `cuda_summary.csv`;
- **CUDA (multi-stream):** configurazione a 8 stream (`cuda_ms_summary.csv`).

I plot mostrano l'andamento dei tempi, del throughput e dello speedup rispetto alla baseline sequenziale, consentendo un confronto diretto tra i modelli più efficienti per ciascuna classe di parallelizzazione.

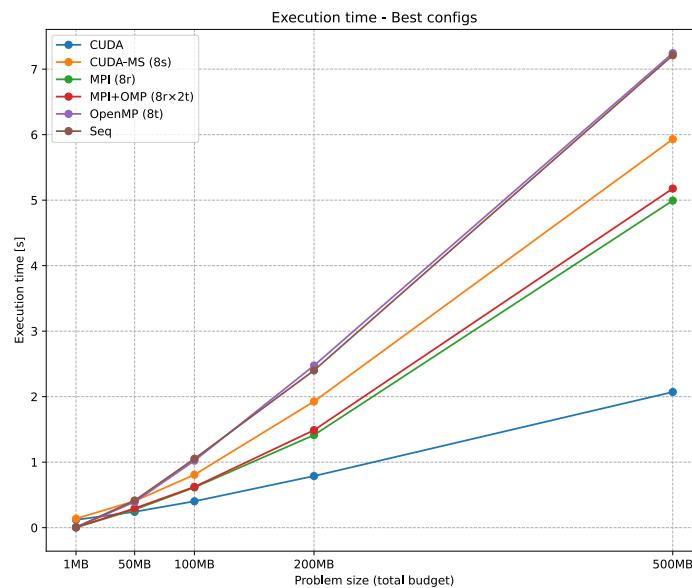


Figure 6.32: Tempi di esecuzione

6. ANALISI DELLE PRESTAZIONI

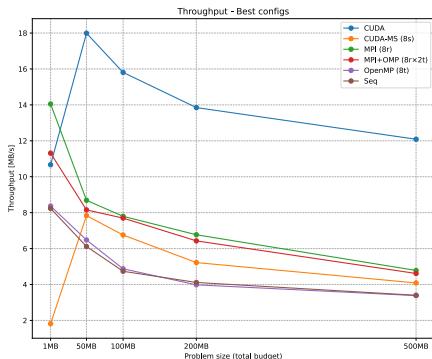


Figure 6.33: Throughput (MB/s)

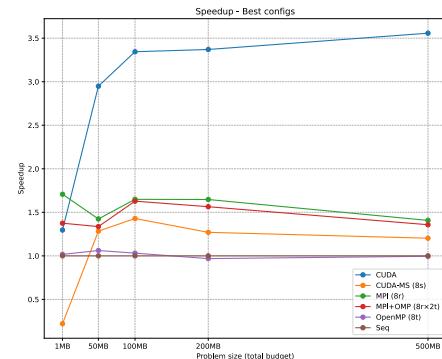


Figure 6.34: Speedup

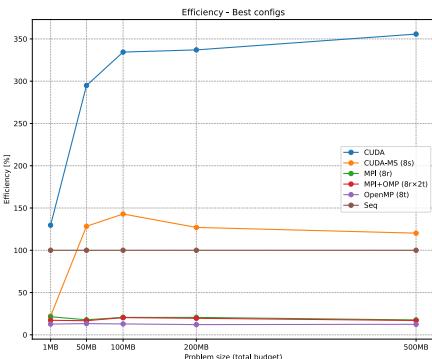


Figure 6.35: Efficiency

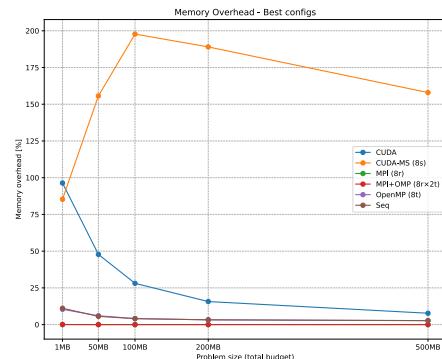


Figure 6.36: Memory overhead ratio (%)

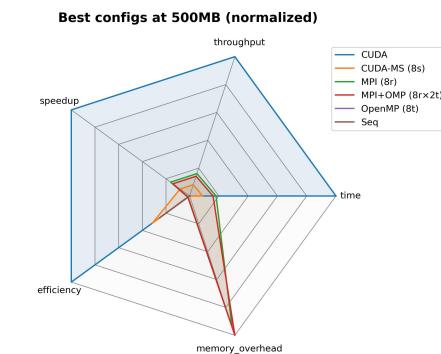


Figure 6.37: Radar chart

LIST OF TABLES

1.1	Relazione tra target di memoria complessivo e dimensione effettiva della stringa generata.	5
6.1	Prestazioni della versione sequenziale (medie su 10 run).	33
6.2	Risultati OpenMP (medie su 10 run): tempi, speedup ed efficienza.	36
6.3	Risultati MPI (medie su 10 run): tempi, speedup ed efficienza.	39
6.4	MPI+OpenMP — Medie (10 run) per size e configurazione.	42
6.5	CUDA single-stream — Medie (10 run) per dimensione: tempi per fase, throughput e speedup/efficienza.	45
6.6	CUDA <i>multi-stream</i> — medie (10 run) per dimensione e numero di stream. .	47

LIST OF FIGURES

6.1	Tempi sequenziale	33
6.2	Speedup sequenziale	34
6.3	Throughput sequenziale	34
6.4	Memory usage sequenziale	35
6.5	Tempi OpenMP	37
6.6	Throughput OpenMP	37
6.7	Speedup OpenMP	37
6.8	Efficiency OpenMP	37
6.9	Memory usage OpenMP (500 MB)	38
6.10	Tempi MPI	39
6.11	Throughput MPI	39
6.12	Speedup MPI	40
6.13	Efficiency MPI	40
6.14	Memory usage MPI (profiling Massif su rank 0)	41
6.15	Tempi MPI+OMP	42
6.16	Throughput MPI+OMP	42
6.17	Speedup MPI+OMP	42
6.18	Efficiency MPI+OMP	42
6.19	Memory usage MPI+OMP (rank 0, input 500 MB).	44
6.20	Tempi CUDA	45
6.21	Throughput CUDA	45
6.22	Speedup CUDA	45
6.23	Efficiency CUDA	45
6.24	Memory overhead CUDA	46

LIST OF FIGURES

6.25 Breakdown CUDA	46
6.26 Tempi CUDA MS	48
6.27 Throughput CUDA MS	48
6.28 Speedup CUDA MS	48
6.29 Efficiency CUDA MS	48
6.30 Memory overhead CUDA MS	49
6.31 Breakdown CUDA MS (500MB, profili Nsight)	49
6.32 Tempi di esecuzione	50
6.33 Throughput (MB/s)	51
6.34 Speedup	51
6.35 Efficiency	51
6.36 Memory overhead ratio (%)	51
6.37 Radar chart	51