Relazione Progetto Laboratorio di Programmazione Parallela

Domenico Commisso - Matricola: 605011
Laboratorio di Programmazione Parallela - A.A. 2025/2026

15 dicembre 2025

Indice

1 Introduzione

Descrizione del problema.

- L'algoritmo DEFLATE è intrinsecamente sequenziale: la compressione di un byte dipende dallo stato storico dei dati precedenti (sliding window e tabelle di Huffman dinamiche). Per parallelizzare il processo, è necessario dividere il file di input in blocchi indipendenti, che vengono compressi separatamente da thread diversi.

  Questo approccio, però, introduce una problematica strutturale in fase di decompressione. Ogni blocco compresso (usando il flag di flush `Z_FINISH`) diventa indipendente e termina con un marcatore di fine stream (`Z_STREAM_END`).

  Trattando il file risultante come un flusso unico, una decompressione standard, incontrando il marcatore di fine flusso del primo blocco, interpreterebbe quel segnale come la fine dell'intero file, ignorando tutti i blocchi successivi. Di conseguenza, il compressore parallelo non può limitarsi a concatenare i dati, ma deve strutturare il file (aggiungendo metadati) in modo che il decompressore continui a processare tutti i chunk successivi fino al completamento reale della decompressione.

2 Istruzioni per Compilazione ed Esecuzione

In questa sezione vengono fornite le istruzioni necessarie per compilare ed eseguire le tre versioni del progetto (Sequenziale, C++ Native e OpenMP) sui nodi del cluster.

2.1 Compilazione

Il codice è stato suddiviso in tre eseguibili distinti per separare l'implementazione sequenziale, quella basata su `std::thread` (C++20) e quella quella basata su OpenMP.

```
$ make all
# Genera tre eseguibili: miniz_seq, miniz_par e miniz_omp
```

Listing 1: Compilazione tramite Makefile

2.2 Esecuzione e Parametri

La sintassi a riga di comando è identica per tutti gli eseguibili. Il programma accetta una lista di file o directory in input e una serie di flag di configurazione.

```
./miniz_<version> [opzioni] <input_1> [input_2] ...
```

Listing 2: Sintassi generale

2.2.1 Descrizione dei Flag

- I flag sono uguali a quelli presenti nella versione sequenziale già fornita, con l'aggiunta di un nuovo flag.
- `-t <bytes>`: **Soglia dimensione (Threshold)**. Specifica la dimensione in byte che discrimina tra file "piccoli" e "grandi".
  - *File size* $<$ *Threshold:* Processati sequenzialmente (parallelismo a livello di file).
  - *File size* $\geq$ *Threshold:* Processati a blocchi paralleli (parallelismo interno al file).
  - *Default:* 16 MB (16777216 byte).

2.3 Esempi di Utilizzo

Esempio 1: Compressione standard (C++ Native)
Comprime ricorsivamente la cartella `dataset`, mantenendo i file originali e usando la soglia di default (16MB).

```
$ ./miniz_par -r 1 -C 0 dataset
```

Esempio 2: Decompressione con OpenMP
Decomprime i file nella cartella `dataset`, rimuovendo i file `.defl` originali.

```
$ ./miniz_omp -r 1 -D 1 dataset
```

Esempio 3: Tuning della Soglia
Esegue la compressione impostando una soglia molto bassa (1MB) per forzare il parallelismo a blocchi anche su file di medie dimensioni.

```
# Soglia impostata a 1 MB (1048576 bytes)
$ ./miniz_omp -C 0 -t 1048576 file_medio.bin
```

3 Dettagli Implementativi

In questa sezione vengono descritte le scelte architetturali comuni e le specificità delle due versioni.

3.1 Gestione dei File e I/O

La scelta di utilizzare `mmap` nella maggior parte dei casi, rispetto ai tradizionali stream C++ (`std::ostream`) è dettata dalla necessità di minimizzare l'overhead di I/O. Mentre `ostream` si basa su system call che richiedono il trasferimento dei dati dallo User Space al Kernel Space, con costosi cambi di contesto e copie di memoria ridondanti, `mmap` proietta il file direttamente nello spazio di indirizzamento virtuale del processo. Questo approccio (*zero-copy*) permette all'applicazione di trattare il file come un array in memoria, riducendo la latenza e consentendo un random access immediato, essenziale per la scrittura parallela non sequenziale dei blocchi compressi.
Per sfruttare il mapping uso due classi, una per i file di sola lettura(input) e una per i file di sola scrittura(output):

- `ReadOnlyFileMap`: Implementa il pattern RAII per l'input. Mappa l'intero file in memoria in modalità sola lettura (`PROT_READ`), permettendo ai thread di accedere ai dati tramite un puntatore diretto senza effettuare copie intermedie.
- `WriteOnlyFileMap`: Gestisce l'output mappando in memoria un'area pre-allocata su disco (`MAP_SHARED`). Include il metodo `commit(size)`, necessario per troncare il file alla dimensione esatta dei dati compressi una volta terminate le scritture parallele, eliminando lo spazio in eccesso.

La soglia(Threshold) per distinguere i file "piccoli" da quelli "grandi" viene impostata di default a 4MB(per motivi che vedremo nell'analisi del threshold), viene salvata all'interno della struct "config".

- I file sotto la soglia vengono processati sequenzialmente ma in parallelo tra loro.
- I file sopra la soglia vengono processati in parallelo ma sequenzialmente tra loro.

3.2 Versione sequenziale

La versione sequenziale non ha subito grosse modifiche, se non l'introduzione del mapping del file in ingresso e il controllo per la presenza del footer nella decompressione. Il controllo sul footer, permette di ignorarne la presenza così da evitare errori di decompressione che possono portare alla creazione di un file diverso da quello originale.

3.3 Versione Parallela - C++ Standard (Native)

Per la prima versione parallela del progetto, ho utilizzato la classe `threadPool` fornita durante il corso per la gestione efficiente dei thread worker.

3.3.1 Compressione

La funzione inizia definendo la dimensione dei *chunk* che sarà tra 32kB e 4MB e calcolando il numero totale di task necessari per coprire l'intero file in base alla dimensione dei *chunk*. Viene preparato un vettore di strutture `CompressionJob`, dove ogni elemento descrive un singolo lavoro, con: Id del task, offset di partenza nel file mappato e la dimensione esatta dei dati da processare (in byte).

Vengono allocate le seguenti risorse:

- `compressed_files`: un vettore di `vector<unsigned char>` che fungerà da buffer temporaneo per memorizzare l'output compresso di ogni blocco.
- `max_dest_len`: la dimensione massima teorica di un blocco compresso, calcolata tramite la funzione `compressBound(CHUNK)` fornita da miniz.
- `ok`: un `atomic<bool>` inizializzato a `true`, utilizzato per segnalare eventuali errori tra i thread.
- `ready`: uno `unique_ptr` a un array di oggetti `PaddedFlag`. Ogni flag è un `atomic<bool>` allineato alla cache line (per evitare false sharing) e serve a segnalare al *writer* che un blocco specifico è pronto per essere scritto.

Abbiamo quindi due ruoli distinti:

- `Producer`: Viene lanciata una lambda function per ogni blocco (task). Ogni producer legge una porzione di dati dal file mappato in memoria e la comprime indipendentemente. Il risultato viene salvato nella posizione corrispondente del vettore `compressed_files`. Al termine della compressione, il producer imposta a `true` il flag `ready[i]` e notifica il writer (tramite `notify_one`), segnalando la disponibilità dei dati.
- `Writer`: Un singolo task *consumer* viene lanciato in parallelo ai producer. Deve garantire l'ordine sequenziale dei dati nel file di output. Il writer itera sugli indici dei blocchi da 0 a N-1 e, per ogni blocco `i`, attende attivamente (tramite `wait`) che il flag `ready[i]` diventi `true`. Una volta disponibile, copia i dati compressi dal buffer temporaneo alla memoria mappata di output e libera immediatamente la memoria del buffer per ridurre l'occupazione della RAM. Durante questo processo, memorizza le dimensioni di ogni blocco compresso, necessarie per costruire il footer.

Struttura del Footer Al termine della scrittura dei dati compressi, il writer appende al file un blocco finale di metadati, essenziale per consentire la decompressione parallela. Il footer contiene:

1. L'array delle dimensioni (in byte) di tutti i blocchi compressi (`block_sizes`).
2. Il numero totale di blocchi (`ntask`), come `size_t`.
3. Un identificatore univoco a 64 bit (`0xB045A3`), che serve come firma per affermare che il file sia stato generato dal compressore parallelo.

Infine, viene chiamato il metodo `commit()` sulla mappa di output per troncare il file alla dimensione esatta (dati + footer), eliminando lo spazio in eccesso pre-allocato.

3.3.2 Decompressione

La procedura inizia con la lettura del footer in coda al file per recuperare la struttura dei dati. Dopo aver validato l'identifier, il programma legge l'array delle dimensioni dei blocchi compressi e ricostruisce il vettore dei `CompressionJob`, calcolando gli offset necessari per accedere ai singoli chunk.

Il parallelismo segue lo stesso pattern Producer/Consumer della compressione:

- `Producer`: Ogni task inizializza uno stato `z_stream` per la decompressione (`inflate`). A differenza della compressione, la dimensione dell'output non è nota a priori; pertanto, il producer decomprime i dati in un buffer che viene espanso dinamicamente. Al termine dell'operazione, il buffer viene spostato nel vettore globale `decompressed_blocks` e viene attivato il flag `ready[i]` corrispondente.
- `Writer`: Il consumer itera sequenzialmente sugli indici dei blocchi, attendendo che il blocco i-esimo sia pronto tramite attesa attiva sul flag atomico. Una volta scritti i dati sullo stream di output, il writer esegue una deallocazione aggressiva della memoria (`clear` e `shrink_to_fit`) sul buffer appena consumato. Questo passaggio è fondamentale per evitare l'esaurimento della RAM (OOM) quando si elaborano file di grandi dimensioni.

3.3.3 walkDir e doWork

Per la funzione walkDir, facciamo (l'analisi ricorsiva o non) della cartella e, per ogni file, chiamiamo una lambda function che decide se il file è valido e se deve stare nel vettore dei file "grandi" o in quello dei file "piccoli". Il vettore dei file piccoli verrà elaborato in parallelo, quello dei file grandi sarà elaborato sequenzialmente perchè poi sarà la doWork a chiamare su di essi l'elaborazione parallela. La doWork controlla il file, ne crea il mapping in lettura e crea il file di output. In base alla dimensione del file, decide se elaborarlo in sequenziale o in parallelo.

3.4 Versione Parallela - OpenMP

Questa implementazione sostituisce la gestione esplicita dei thread con le direttive di compilazione OpenMP, delegando al runtime la gestione del pool di risorse e dello scheduling.

3.4.1 Compressione

La strategia che ho adottato per la compressione è: *data-parallel*. La dimensione del *chunk* viene calcolata dinamicamente (tra 32KB e 4MB) per bilanciare granularità e overhead. Viene allocato un unico grande buffer contiguo (`arena`) dimensionato per contenere l'output di tutti i task nel caso peggiore. Il processo si divide in due fasi parallele distinte:

1. **Fase di Calcolo:** Un ciclo `#pragma omp parallel for` distribuisce i blocchi di input ai thread. Ogni thread scrive il risultato in una porzione pre-calcolata del buffer `arena` e salva la dimensione compressa. L'accesso in scrittura è sicuro poiché ogni thread lavora su aree di memoria distinte.
2. **Fase di Scrittura:** Dopo aver calcolato gli offset in modo sequenziale(farlo in parallelo porterebbe troppo overhead), un secondo ciclo parallelo esegue delle `memcpy` concorrenti per trasferire i dati dall'`arena` alla mappa di output su disco.

3.4.2 Decompressione

Per la decompressione ho applicato un approccio basato sul **Task Parallelism**, poiché la dimensione dell'output non è nota e l'ordine di scrittura deve essere preservato. L'implementazione utilizza il costrutto `#pragma omp task` all'interno di una regione `single`, creando un grafo di dipendenze gestito dal runtime. Ogni task avrà un id i `firstprivate`, quindi inizializzato al valore i corrente del ciclo for

ma privato e gli altri valori `shared`, quindi condivisi con le altre task. Per evitare l'esaurimento della memoria (OOM) su file di grandi dimensioni, i blocchi vengono processati in `Batch` di 64 elementi.

- **Task Producer:** Decomprime un blocco usando `inflate` con buffer dinamico. La clausola `depend(out: buf[i])` segnala al runtime quando il dato è pronto.
- **Task Consumer:** Si occupa della scrittura su file. Utilizza la clausola `depend(in: buf[i])` per attendere automaticamente il completamento del producer relativo a quel blocco, e `depend(inout: order_token)` per garantire che le scritture avvengano in ordine sequenziale ($0 \to 1 \to \cdots \to N$).

Questa gestione implicita delle dipendenze semplifica la sincronizzazione.

3.4.3 WalkDir e doWork

La funzioni `walkDir` e `doWork` hanno la stessa logica delle implementazioni presenti nella versione parallela senza openMP. Il doWork, dopo aver riempito i due array, elabora i file piccoli tramite un `parallel for` con scheduling dinamico. Per controllarne il successo, effettua una riduzione con operazione `and` su una variabile booleana `global_ok` che viene modificata a false al verificarsi di un errore. I file grandi vengono eseguiti sequenzialmente.

4 Analisi delle Prestazioni

I test sono stati eseguiti su un singolo nodo del cluster con i file salvati sulla directory scratch del nodo. Le metriche considerate sono lo **Strong Speedup** e la **Weak Scalability**.

4.1 Specifiche hardware

CPU: Intel Xeon dual socket, 20 core fisici totali e 2 thread per core. *RAM:* 125GB totali divisi in due nodi NUMA associati ai rispettivi socket.

4.2 Strong Speedup

Lo Strong Speedup è calcolato come $S_N = \frac{T_1}{T_N}$, dove T_1 è il tempo con 1 thread e T_N con N thread, mantenendo fissa la dimensione del problema.

4.2.1 Compressione File Grandi

Dal grafico in Figura 1, possiamo osservare l'andamento dello Strong Speedup comprimendo due file di grandi dimensioni (oltre la soglia di 4MB). Le prestazioni in compressione arrivano ad un massimo di circa 9x e in decompressione a circa 3x, rispetto alla versione sequenziale. Gli andamenti di entrambe le versioni parallele sono equiparabili. Le prestazioni hanno un picco quando si usano 32 core.

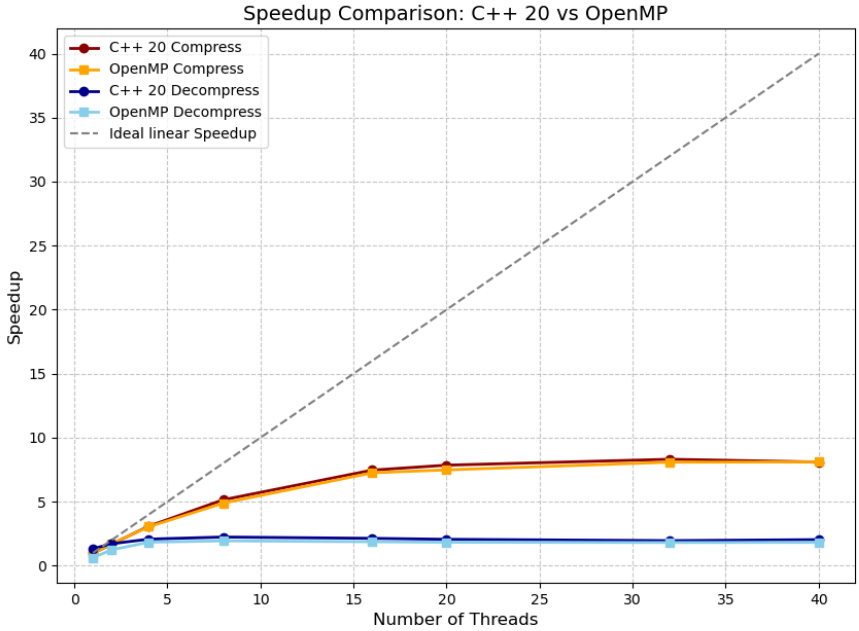

Figura 1: Analisi Strong Speedup: Pochi file grandi

4.2.2 Compressione Molti File Piccoli

La Figura 2 mostra invece il comportamento del sistema quando deve gestire un elevato numero di file sotto il threshold. Il parallelismo è gestito a livello di file e non di blocco. Qui la compressione raggiunge uno speedup del 15x con 32 core, mentre la decompressione rimane su un 3x.

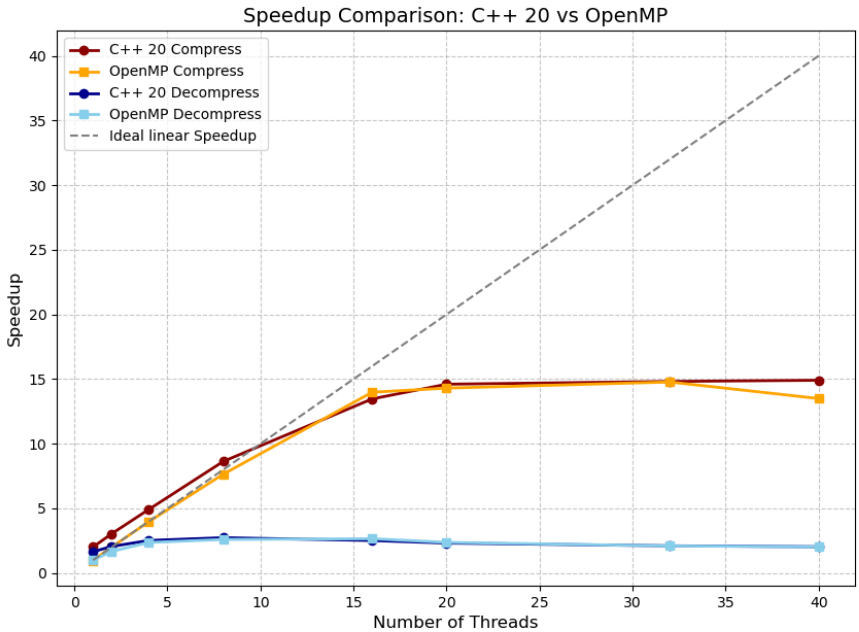

Figura 2: Analisi Strong Speedup: Molti file piccoli

4.3 Weak Scalability

La Weak Scalability valuta come si comporta il sistema aumentando la dimensione del problema proporzionalmente al numero di processori.
Configurazione test: 1 Thread = 25MB, 2 Thread = 50MB, ..., N Thread = N*25MB.
L'obiettivo è mantenere il tempo costante. I file generati sono dei txt formati da libri scaricabili gratuitamente dal `project Gutemberg` e concatenati n volte per ottenere le dimensioni cercate. Come possiamo vedere, il tempo di esecuzione non si mantiene costante, ma mostra cresce all'aumentare dei core(e quindi anche della dimensione del file). Il monitoraggio delle risorse durante l'esecuzione (effettuata tramite htop sui nodi di calcolo) ha evidenziato un utilizzo della CPU altalenante, con frequenti oscillazioni tra il 100% di utilizzo e percentuali molto basse. Questo comportamento indica un collo di bottiglia di tipo I/O Bound.

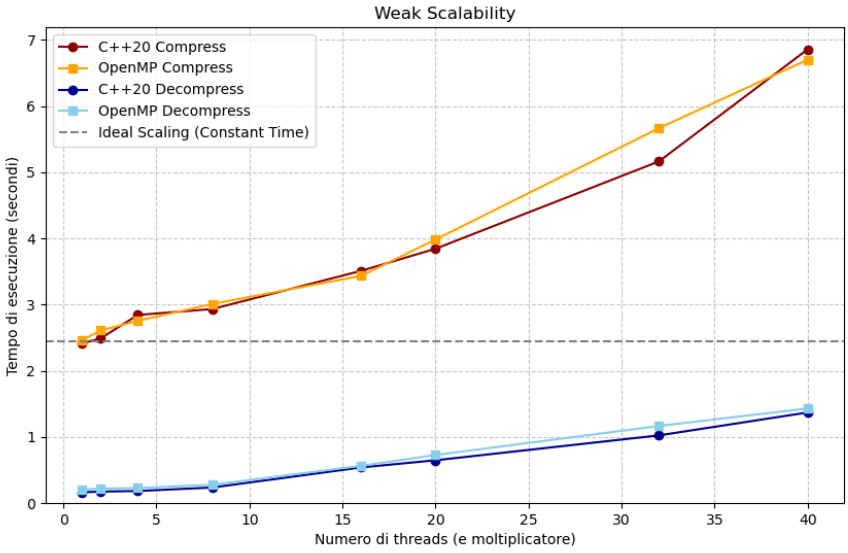

Figura 3: Grafico della Weak Scalability

4.4 Analisi Threshold

La figura 4 mostra l'analisi dell'impatto sullo speedup di diversi valori threshold.
Configurazione test: Il test viene effettuato su una cartella contenente file misti(da pochi Kb a 1GB) e utilizzando tutti i core a disposizione.

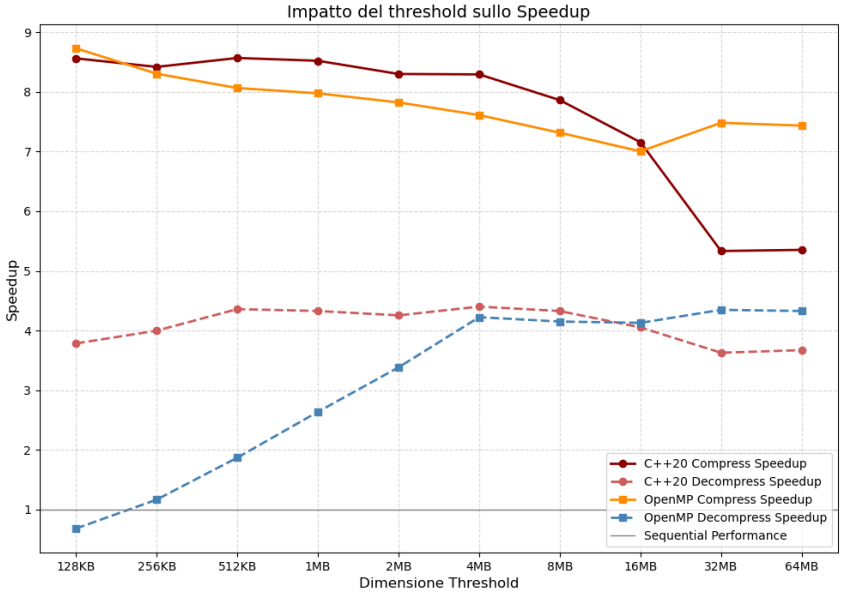

Figura 4: Grafico analisi threshold

5 Criticità e Soluzioni Adottate

Descrizione dei problemi incontrati durante lo sviluppo.

- **Divisione in blocchi del file:** Dovendo dividere i file di grandi dimensioni in blocchi per comprimerli in parallelo, ogni blocco è diventato indipendente dagli altri. Ogni task di compressione invoca la funzione `deflate` utilizzando il flag di flush `Z_FINISH`. Parametro fondamentale perché segnala che l'input corrente conclude il file, forzando la scrittura di tutti i dati pendenti e l'aggiunta del marcatore di fine stream (`Z_STREAM_END`). Ogni blocco diventa un piccolo file autonomo, permettendo ai thread di decomprimerli successivamente in parallelo senza dipendenze di contesto. L'implementazione memorizza la dimensione compressa di ogni blocco, così da inserire, alla fine del file, un *footer* contenente la tabella degli offset e il numero totale di blocchi. Infine, viene aggiunto un identificatore univoco che verrà cercato dal decompressore per distinguere i file compressi in parallelo da quelli sequenziali.

- **Scrittura file compresso in parallelo:** Il file compresso, deve essere scritto sul file di output in ordine corretto. Pertanto, avendo salvato la dimensione dei singoli blocchi compressi, posso calcolare la dimensione finale del file e la posizione in cui ogni blocco dovrà essere scritto. Questo mi permette di rendere parallela anche la scrittura del file compresso.

- **Gestione file piccoli:** Parallelizzare la compressione di file piccoli porterebbe solo costi superiori dovuti a overhead per la gestione dei thread. Pertanto, i file piccoli vengono processati in modo sequenziale. Il threshold viene impostato di base a 4MB perchè, nell'analisi(figura 4) è risultato il valore che permette di avere quasi il valore ottimo di speedup in decompressione e di non perdere troppo speedup in compressione.

6 Conclusioni

Il progetto ha dimostrato come l'applicazione di tecniche di programmazione parallela all'algoritmo di compressione DEFLATE possa portare a miglioramenti prestazionali, nonostante la dipendenza storica

dei dati.
Dall'analisi dello Strong Speedup, emerge che le due implementazioni(C++20 e openMP) offrono prestazioni pressoché identiche.
OpenMP: Si è rivelata la soluzione più rapida da implementare e più compatta a livello di codice, delegando efficacemente al runtime la gestione del bilanciamento del carico.
C++ Standard: Sebbene richieda una gestione più verbosa di thread e sincronizzazione, offre un controllo più granulare sulla memoria e sulla gestione delle risorse.
In termini di efficienza pura, nessuna delle due prevale nettamente sull'altra, suggerendo che il collo di bottiglia principale non risieda nella gestione dei thread, ma nelle operazioni di I/O e calcolo.

6.1 Scalabilità e Limiti

Strong Speedup: Il sistema scala bene fino a circa 32 thread.
Nella compressione, otteniamo uno speedup massimo di circa 9x per file grandi e 15x per molti file piccoli. Quest'ultimo caso beneficia maggiormente del parallelismo perché elimina le dipendenze tra i blocchi, riducendo l'overhead di sincronizzazione.
La decompressione si ferma a uno speedup di circa 3x. Questo è dovuto alla natura del decompressore che, dovendo scrivere un output di dimensione ignota a priori, richiede allocazioni dinamiche e una sincronizzazione più forte per mantenere l'ordine sequenziale, limitando i benefici del calcolo parallelo.

Weak Scalability: Il tempo di esecuzione non rimane costante all'aumentare proporzionale di carico e core. Questo fenomeno evidenzia che l'applicazione è parzialmente Memory/IO Bound: l'aumento dei core satura la banda passante verso la memoria o verso il disco (specialmente in scrittura), impedendo una scalabilità lineare ideale.

*Threshold:*L'analisi del Threshold ha confermato che non esiste una strategia unica perfetta. L'introduzione di una soglia dinamica (fissata di base a 4MB) è utile per massimizzare le prestazioni in base al contesto. Processare file piccoli interamente in parallelo evita l'overhead eccessivo della divisione in blocchi, mentre la suddivisione interna è essenziale per sfruttare i core su file di grandi dimensioni.