
Università degli Studi di Salerno

Dipartimento di Informatica

---

Corso di Laurea Magistrale in Informatica
Corso di Compressione Dati

# Improving Secure Compression Based on the Burrows–Wheeler Transform Using MPI and the DC3 Algorithm

**Studenti**
Andrea Sasso
Mattia Giuseppe Giella

**Docente**
Bruno Carpentieri

---

Anno Accademico 2025/2026

# Indice

**Abstract** **1**

**1 Introduzione** **2**
1.1 Prestazioni problematiche . . . . . 2
1.2 Pipeline proposta e modifiche . . . . . 3
1.2.1 Integrazione dell'algoritmo DC3 nella sBWT . . . . . 3
1.2.2 Euristica adattiva per la dimensione dei blocchi . . . . . 3
1.2.3 Parallelizzazione della fase RLE . . . . . 3
1.2.4 Ottimizzazione del Multiprocessing e gestione dei dati . . . . . 3
1.3 Parallelizzazione MPI . . . . . 4
1.4 Struttura della tesina . . . . . 4

**2 Algoritmi e Librerie utilizzati** **5**
2.1 Algoritmi di Compressione e Sicurezza . . . . . 5
2.1.1 Scrambled Burrows-Wheeler Transform (sBWT) . . . . . 5
2.1.2 Algoritmo DC3 per il Suffix Array . . . . . 6
2.1.3 Blocky Move-To-Front (bMTF) . . . . . 6
2.1.4 Run-Length Encoding (RLE) . . . . . 7
2.1.5 Codificatori a lunghezza variabile . . . . . 7
2.2 Linguaggio e Librerie Software . . . . . 8
2.2.1 Multiprocessing (Standard Library) . . . . . 8
2.2.2 MPI . . . . . 8
2.2.3 mpi4py (MPI for Python) . . . . . 8

**3 Implementazione** **9**
3.1 sBWT . . . . . 9
3.1.1 Euristica per la determinazione dinamica dei blocchi . . . . . 9
3.1.2 Considerazioni sul Multiprocessing . . . . . 10
3.1.3 Algoritmo DC3 per la costruzione lineare del Suffix Array . . . . . 11
3.1.4 Counting Sort per l'ordinamento lineare . . . . . 12
3.2 bMTF . . . . . 14
3.3 RLE . . . . . 14
3.3.1 Ottimizzazione della memoria tramite Shared Array . . . . . 14

3.3.2 Parallelizzazione della RLE . . . . . 14
3.4 Variante MPI . . . . . 15
3.4.1 Distribuzione dei dati e gestione dei blocchi . . . . . 16
3.4.2 Raccolta e merging dei risultati . . . . . 16
3.4.3 Coerenza e sicurezza della pipeline . . . . . 16
3.4.4 Vantaggi della variante MPI . . . . . 16
3.5 Ulteriori Ottimizzazioni . . . . . 17

**4 Testing e Risultati** **18**
4.1 Ambiente di esecuzione . . . . . 18
4.2 Analisi dei risultati . . . . . 18
4.2.1 Miglioramenti Ottenuti . . . . . 18
4.2.2 Confronto con Bzip2 . . . . . 19
4.2.3 Rapporto di Compressione . . . . . 19
4.2.4 Anomalie Riscontrate . . . . . 19

**5 Conclusioni** **20**
5.1 Considerazioni critiche e limitazioni . . . . . 21
5.2 Sviluppi futuri . . . . . 21

# Abstract

Il progetto presentato ha l'obiettivo di estendere e ottimizzare un sistema di compressione sicura basato sulla Burrows-Wheeler Transform (BWT). La pipeline di riferimento integra la trasformata con gli algoritmi Move-To-Front (MTF), Run-Length Encoding (RLE) e codifiche a lunghezza variabile. L'architettura preesistente implementa un layer di sicurezza alla pipeline classica, attraverso la Scrambled BWT (sBWT), basata sulla randomizzazione dell'alfabeto, e una variante della MTF suddivisa in blocchi concatenati tramite funzioni di hash. Il contributo principale di questo progetto risiede in una revisione algoritmica volta a migliorare la scalabilità e l'efficienza computazionale del sistema. In primo luogo, è stato integrato l'algoritmo DC3 per la costruzione del Suffix Array, riducendo la complessità temporale da $\mathcal{O}(n \log^2 n)$ a $\mathcal{O}(n)$. In secondo luogo, è stata introdotta un'euristica adattiva per la determinazione dinamica della dimensione dei blocchi, progettata per ottimizzare l'utilizzo della gerarchia delle memorie cache (L2/L3) e migliorare sensibilmente il rateo di compressione della fase RLE, precedentemente penalizzato da una segmentazione statica. Il lavoro affronta inoltre la parallelizzazione dell'intera pipeline, estendendo il supporto multiprocessore alla fase di RLE tramite l'uso di Shared Arrays per minimizzare l'overhead di comunicazione. Oltre all'implementazione in ambiente multiprocessing Python, è stata sviluppata una versione basata sullo standard Message Passing Interface (MPI), consentendo una distribuzione del carico più efficiente e granulare. I risultati ottenuti mostrano una significativa riduzione dei tempi di esecuzione e un miglioramento dell'efficienza di compressione, senza compromettere la robustezza crittografica della pipeline sicura.

# Capitolo 1

# Introduzione

## 1.1 Prestazioni problematiche

L'implementazione di riferimento, basata sul lavoro di Zeng et al. [1], introduce un'implementazione sicura per la compressione tramite l'integrazione della *Scrambled Burrows-Wheeler Transform* (sBWT) e della *Blocky Move-to-Front* (bMTF). Un'analisi approfondita delle prestazioni ha evidenziato alcune criticità che limitano l'applicabilità dell'algoritmo in scenari reali e su dataset di grandi dimensioni. Le principali problematiche riscontrate riguardano:

- **Inefficienza della sBWT:** La versione originale utilizza un algoritmo di costruzione del *Suffix Array* con complessità temporale $O(n \log^2 n)$, che rappresenta un collo di bottiglia computazionale all'aumentare della dimensione dell'input. Per garantire la scalabilità del sistema, è stato integrato l'algoritmo **DC3**, riducendo la complessità a $O(n)$.

- **Gestione statica dei blocchi:** Il progetto preesistente presentava criticità nella gestione dei blocchi sia per la sBWT che per la bMTF. La parallelizzazione della sBWT, basata su blocchi statici da 300 KB, non si adattava alla grandezza dell'input, portando a un utilizzo inefficiente delle risorse computazionali. Parallelamente, la bMTF utilizzava una segmentazione fissa di 1024 byte, tale granularità garantisce sicurezza crittografica attraverso un rimescolamento frequente del dizionario, ma penalizza sia il rateo di compressione e che il tempo di esecuzione. Come soluzione è stata introdotta un'euristica che determina la dimensione dei blocchi in modo dinamico.

- **RLE sequenziale:** Nonostante la parallelizzazione della sBWT, la *Run-Length Encoding* (RLE) rimaneva sequenziale. La RLE può essere parallelizzata per ridurre ulteriormente il tempo totale di esecuzione.

- **Limiti del Multiprocessing in Python:** L'implementazione basata sulla libreria `multiprocessing` di Python introduce un overhead considerevole nella creazione e nella comunicazione tra processi. Si è proposta quindi un'implementazione basata su **MPI** (*Message Passing Interface*), standard per il calcolo parallelo distribuito.

L'obiettivo del lavoro proposto è quindi il superamento di tali limiti attraverso una revisione algoritmica e l'introduzione di nuovi paradigmi di parallelizzazione, al fine di migliorare le prestazioni della pipeline sicura, senza sacrificarne le proprietà crittografiche.

## 1.2 Pipeline proposta e modifiche

La soluzione proposta non altera la pipeline in maniera sostanziale, ma introduce miglioramenti al parallelismo e all'efficienza algoritmica.

Le principali innovazioni introdotte sono descritte di seguito:

### 1.2.1 Integrazione dell'algoritmo DC3 nella sBWT

Per superare il limite computazionale della sBWT, è stato implementato l'algoritmo **DC3** per la costruzione del *Suffix Array*. A differenza degli approcci basati su ordinamento standard, DC3 garantisce una complessità $O(n)$. L'integrazione di DC3 all'interno della *Scrambled BWT* ha richiesto un adattamento del layer di sicurezza all'algoritmo, tramite l'ordinamento dei suffissi secondo l'ordinamento segreto generato dalla chiave $K$ e dal seed $r$.

### 1.2.2 Euristica adattiva per la dimensione dei blocchi

Sono state rimosse le limitazioni derivanti dall'uso di blocchi statici, introducendo euristiche per il calcolo dinamico della loro dimensione. L'obiettivo è distribuire equamente il carico di lavoro tra i core disponibili attraverso una suddivisione scalabile dell'input. Tale approccio permette di bilanciare la dimensione dei blocchi per evitare che un'eccessiva frammentazione penalizzi il rateo di compressione o che, al contrario, blocchi troppo estesi portino alla saturazione delle risorse di memoria.

### 1.2.3 Parallelizzazione della fase RLE

Nella pipeline originale, la fase di Run-Length Encoding (RLE) costituiva un passaggio sequenziale successivo alla trasformazione. Nella nuova implementazione, tale fase è stata parallelizzata. In particolare, l'output dell'algoritmo Blocky Move-To-Front (bMTF) viene suddiviso in blocchi indipendenti, consentendo l'esecuzione parallela della compressione, seguita da un'operazione di merge dei risultati. Durante la decompressione, la stringa prodotta dal decoder viene suddivisa in maniera controllata, evitando la separazione dei simboli dai rispettivi contatori di ripetizione. Per file di grandi dimensioni, questa strategia consente una riduzione del tempo di esecuzione, migliorando l'efficienza complessiva dell'algoritmo grazie alla parallelizzazione.

### 1.2.4 Ottimizzazione del Multiprocessing e gestione dei dati

Oltre alle migliorie algoritmiche, è stata ottimizzata la gestione multiprocessing tramite l'introduzione di strutture dati migliori per la gestione della comunicazione e della memoria condivisa.

## 1.3 Parallelizzazione MPI

Oltre all'implementazione basata su `multiprocessing`, è stata sviluppata una versione alternativa che utilizza lo standard **MPI** (*Message Passing Interface*). In questa versione, l'esecuzione è distribuita fra i nodi in maniera più granulare, MPI permette una gestione esplicita della comunicazione (*scatter/gather*), eliminando l'overhead introdotto dall'interprete Python e riducendo il costo di serializzazione e sincronizzazione dei processi.

## 1.4 Struttura della tesina

La tesina è organizzata come segue:

- Il Capitolo 1 introduce le problematiche riscontrate nel progetto precedente e le migliorie apportate;
- Il Capitolo 2 illustra gli algoritmi e le librerie utilizzate, motivando le scelte effettuate e i vantaggi che ne derivano;
- Il Capitolo 3 analizza le implementazioni realizzate, confrontandole con quelle precedenti;
- Il Capitolo 4 descrive il dataset utilizzato e i risultati ottenuti, confrontandoli con la precedente implementazione e discutendone vantaggi e svantaggi;
- Il Capitolo 5 conclude la tesina, riepilogando le criticità e le migliorie apportate e proponendo possibili sviluppi futuri.

# Capitolo 2

# Algoritmi e Librerie utilizzati

In questo capitolo vengono descritti gli algoritmi fondamentali che compongono la pipeline di compressione sicura e le librerie software utilizzate per l'implementazione e la parallelizzazione del sistema.

## 2.1 Algoritmi di Compressione e Sicurezza

### 2.1.1 Scrambled Burrows-Wheeler Transform (sBWT)

La sBWT è una variante crittografica della trasformata di Burrows–Wheeler (BWT) classica. Nella BWT tradizionale i suffissi della stringa di input vengono ordinati in modo lessicografico, mentre nella sBWT l'ordinamento avviene rispetto a un alfabeto segreto, generato a partire da una chiave $K$ e da un seed $r$. In questo modo l'output risulta incomprensibile senza conoscere la chiave, ma mantiene comunque vicini i caratteri uguali o simili, migliorando l'efficacia delle fasi successive di compressione.

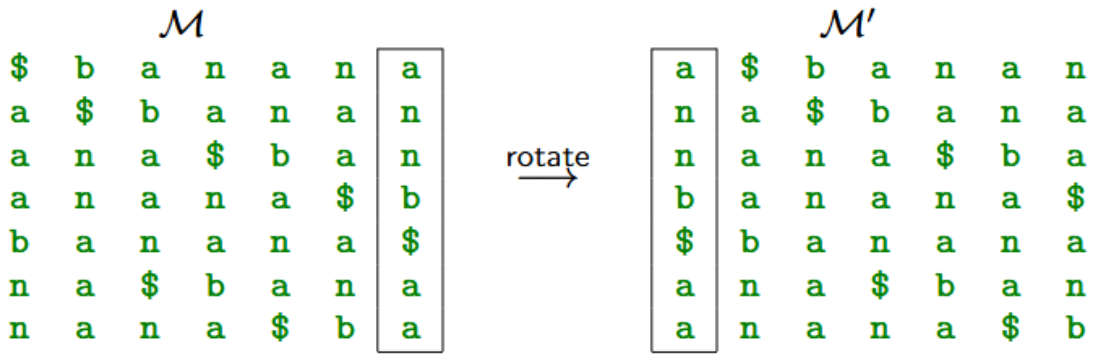


Figura 2.1: Esempio di funzionamento della BWT: si generano tutte le rotazioni cicliche della stringa di input, si ordinano lessicograficamente e si prende l'ultima colonna come output.

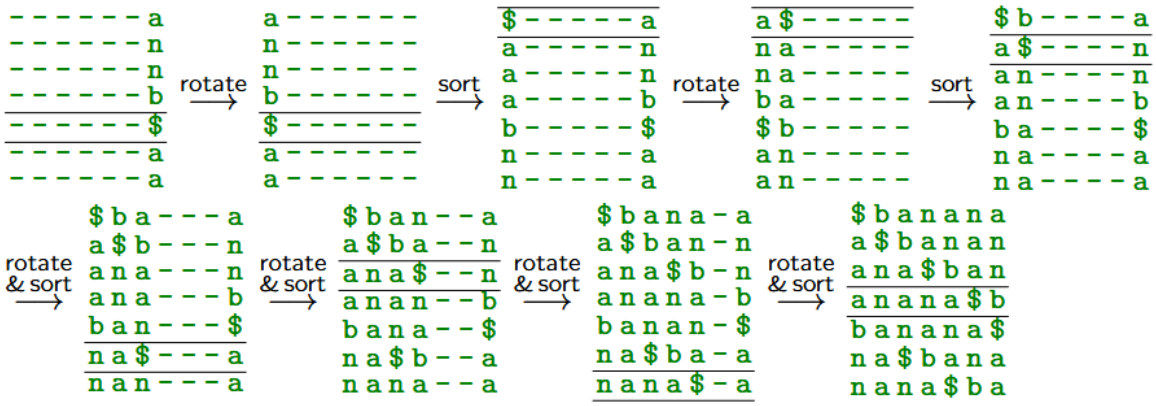


Figura 2.2: Operazione inversa della BWT, dalla stringa si può riottenere la tabella con operazioni di rotazione e ordinamento

### 2.1.2 Algoritmo DC3 per il Suffix Array

Per l'implementazione efficiente della sBWT è stato adottato l'algoritmo **DC3**, che permette di costruire il Suffix Array in tempo lineare $O(n)$ tramite una strategia divide-et-impera [2]. Il procedimento si articola nelle seguenti fasi:

1. campionamento dei suffissi a partire dalle posizioni $i$ tale che $i \bmod 3 \neq 0$;
2. ordinamento ricorsivo dei suffissi campionati;
3. ordinamento dei suffissi rimanenti (con $i \bmod 3 = 0$) sfruttando i risultati del passo precedente;
4. merge finale delle due sequenze di suffissi ordinate.

L'uso dell'algoritmo **DC3** ha comportato una riduzione significativa dei tempi di esecuzione rispetto alla versione originale, caratterizzata da complessità $O(n \log^2 n)$. Questo miglioramento è dovuto principalmente all'impiego del *Counting Sort*, che consente l'ordinamento in tempo lineare in presenza di un alfabeto di dimensione limitata. Gli algoritmi DC3 e Counting Sort saranno descritti in dettaglio nel capitolo dedicato all'implementazione.

### 2.1.3 Blocky Move-To-Front (bMTF)

La bMTF è una variante dell'algoritmo Move-to-Front che introduce un ulteriore livello di sicurezza. L'input viene suddiviso in blocchi e, per ciascun blocco, le posizioni dei simboli nell'alfabeto vengono ricalcolate applicando una nuova permutazione segreta derivata dall'hash del blocco precedente concatenato con la chiave segreta $K$. Questa strategia rende più difficile condurre attacchi statistici basati sulla frequenza dei caratteri, poiché la distribuzione delle posizioni varia in modo dipendente sia dai dati sia dalla chiave.

Illustration for "panama".
List initially contains English alphabets in order.
We one by one characters of input to front.

| input_str chars | output_arr | list |
|---|---|---|
| p | 15 | abcdefghijklmnopqrstuvwxyz |
| a | 15 1 | pabcdefghijklmnoqrstuvwxyz |
| n | 15 1 14 | apbcdefghijklmnoqrstuvwxyz |
| a | 15 1 14 1 | napbcdefghijklmoqrstuvwxyz |
| m | 15 1 14 1 14 | anpbcdefghijklmoqrstuvwxyz |
| a | 15 1 14 1 14 | manpbcdefghijkloqrstuvwxyz |

Figura 2.3: Esempio di funzionamento della Move-to-Front (MTF): partendo da un alfabeto inizialmente ordinato in modo lessicografico, per ogni carattere in input si salva in un'array l'indice corrispondente nella lista e si sposta il carattere in testa all'alfabeto. L'array verrà infine dato in output [3]

.

### 2.1.4 Run-Length Encoding (RLE)

La Run-Length Encoding (RLE) è il primo algoritmo di compressione utilizzato nella pipeline. Essa rappresenta sequenze di caratteri ripetuti tramite coppie (carattere, contatore), risultando facilmente invertibile. Il suo impiego è molto efficace sull'output delle trasformazioni precedenti, che presenta lunghe sequenze consecutive di zeri (grazie alla MTF), permettendo una riduzione significativa della lunghezza della stringa compressa.

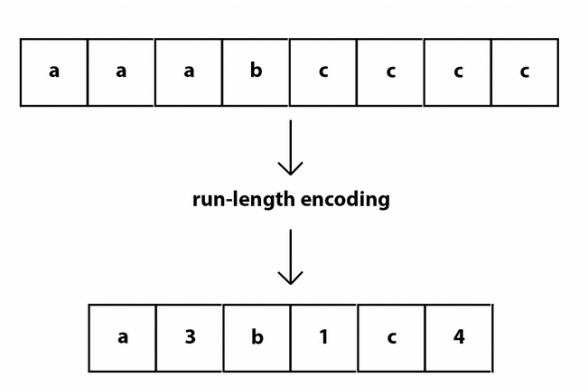


Figura 2.4: Risultato della RLE

### 2.1.5 Codificatori a lunghezza variabile

Per la fase finale della pipeline, sono stati forniti diversi algoritmi della famiglia *Variable Length Prefix Code*:

- **Huffman Coding:** Utilizza un albero binario per assegnare codici più brevi ai simboli più frequenti.
- **LZW (Lempel-Ziv-Welch):** Algoritmo basato su dizionario che sostituisce sequenze di simboli con codici numerici.

- **Arithmetic Coding:** Codifica l'intero messaggio in un unico numero decimale compreso tra 0 e 1, offrendo spesso rapporti di compressione superiori a Huffman.

## 2.2 Linguaggio e Librerie Software

Per l'implementazione si è scelto di utilizzare Python, in quanto il progetto precedente adottava già questo linguaggio di programmazione, apprezzato per l'ampia disponibilità di librerie e per la sua flessibilità nella gestione di strutture dati complesse.

### 2.2.1 Multiprocessing (Standard Library)

Per lo sfruttamento delle architetture multi-core locali è stata utilizzata la libreria multiprocessing della standard library di Python. Essa permette di istanziare processi indipendenti, ciascuno con il proprio interprete Python e il proprio spazio di memoria. In questo modo è possibile eseguire in parallelo la sBWT e la RLE sui diversi blocchi dell'input, riducendo i tempi complessivi di elaborazione.

### 2.2.2 MPI

La **Message Passing Interface (MPI)** è uno standard per la programmazione parallela basata sul *message passing*, cioè sullo scambio esplicito di messaggi tra processi indipendenti. MPI non è un linguaggio, ma una libreria che permette di gestire comunicazione, sincronizzazione e coordinamento tra processi su sistemi a memoria distribuita [4].

In MPI, un programma parallelo è costituito da più processi che comunicato attraverso un *Communicator*, ciascuno con il proprio spazio di indirizzamento e identificato da un *rank*. La comunicazione può essere:

- **Point-to-point:** scambio diretto tra due processi (`MPI_Send`, `MPI_Recv`);
- **Collective:** operazioni su gruppi di processi (`MPI_Broadcast`, `MPI_Scatter`, `MPI_Gather`, `MPI_Reduce`).

MPI inoltre offre primitive avanzate per la gestione dei *communicators* e comunicazioni non bloccanti per sovrapporre calcolo e trasferimento dati.

### 2.2.3 mpi4py (MPI for Python)

Per la versione distribuita è stata utilizzata la libreria open source `mpi4py`, che fornisce un'interfaccia Python allo standard MPI appoggiandosi a implementazioni native come `OpenMPI` o `MPICH`. Attraverso mpi4py è stato possibile utilizzare primitive di comunicazione collettive, tra cui:

- **Scatter**: per distribuire i blocchi di testo dal processo root ai processi worker;
- **Gather**: per raccogliere e riassemblare i blocchi compressi e codificati;
- **Barrier**: per la sincronizzazione dei processi nelle fasi critiche della pipeline.

# Capitolo 3

# Implementazione

In questo capitolo verranno affrontati i principali miglioramenti effettuati e cosa questi hanno comportato all'interno dell'esecuzione.

## 3.1 sBWT

La prima fase della pipeline è la *Scrambled Burrows-Wheeler Transform* (sBWT). Sebbene l'implementazione crittografica sia rimasta invariata, sono stati introdotti diversi miglioramenti prestazionali riguardanti:

- euristica dinamica per la dimensione dei blocchi,
- ottimizzazioni della parallelizzazione `multiprocessing`,
- algoritmo DC3 per la costruzione lineare del Suffix Array.

### 3.1.1 Euristica per la determinazione dinamica dei blocchi

Una delle principali innovazioni rispetto all'implementazione originale è la gestione dinamica della granularità del parallelismo. La scelta statica di blocchi da 300 KB è stata sostituita da un'euristica che adatta la dimensione in base al numero di processori logici disponibili (`multiprocessing.cpu_count()`) e alla dimensione del file di input.

**Euristica per la sBWT e ottimizzazione della Cache** Per la fase di sBWT, implementata nel modulo `compression.py`, la dimensione del blocco ($L_{BWT}$) è calcolata per bilanciare l'occupazione della memoria e sfruttare la gerarchia delle cache di sistema. Sono state definite due costanti: MIN_BLOCK (256 KB), scelta per essere compatibile con la dimensione tipica delle cache L2 per core, e MAX_BLOCK (2 MB), che rappresenta una frazione della cache L3 [5], evitando un degrado della località. Il calcolo della lunghezza del blocco avviene secondo la formula:

$$L_{BWT} = \max\left(L_{min}, \min\left(\frac{S_{total}}{N_{proc}}, L_{max}\right)\right)$$

dove $S_{total}$ è la dimensione del file e $N_{proc}$ il numero di core rilevati. Questa logica garantisce che:

- Su file di grandi dimensioni, i blocchi non devono superare i 2 MB per evitare un'eccessiva pressione sulla memoria e massimizzare il *cache-reuse* durante la costruzione del Suffix Array.
- Su file piccoli, venga preservata una soglia minima (256 KB) sotto la quale la suddivisione in blocchi impatterebbe troppo sul rateo di compressione.

Come verrà mostrato nella sezione dedicata all'analisi dei risultati, sebbene la parallelizzazione e la suddivisione dei dati in blocchi producano un miglioramento significativo dei tempi di compressione, il rateo di compressione risente della suddivisione in blocchi.

### 3.1.2 Considerazioni sul Multiprocessing

L'efficienza della parallelizzazione in Python è spesso limitata dall'overhead della comunicazione inter-processuale e dal costo della serializzazione dei dati (*pickling*)[6]. Per mitigare tali criticità, l'implementazione proposta ottimizza l'uso della classe *multiprocessing.Pool* attraverso una gestione migliore della distribuzione dei task e della raccolta dei risultati, evitando l'uso di strutture condivise basate su *multiprocessing.Manager*, che introducono un ulteriore livello di sincronizzazione e serializzazione.

Le principali scelte tecniche adottate nel modulo `compression.py` sono descritte di seguito:

**Utilizzo di imap_unordered e riordinamento indicizzato** Per la distribuzione dei task è stato utilizzato `pool.imap_unordered`. Questa scelta permette ai processi *worker* di restituire i risultati non appena pronti, senza attendere l'ordine sequenziale dei blocchi. Per garantire la coerenza dell'input originale, è stata implementata una strategia di **riordinamento asincrono**:

- Ogni task viene inviato come una tupla contenente l'indice del blocco (`index`).
- Il *worker* restituisce una coppia `(index, result)`.
- Il processo *master* inserisce il risultato direttamente nella posizione corretta di una lista pre-allocata: `output[index] = res`.

Questo approccio riduce i tempi di attesa del processo principale, permettendo di iniziare la ricostruzione del flusso di dati mentre gli altri core stanno ancora elaborando i blocchi rimanenti.

**Calcolo dinamico del chunksize** Il parametro `chunksize` determina il numero di task inviati a un worker in un singolo scambio di messaggi. Sia nella parallelizzazione della BWT che di RLE, il valore viene calcolato dinamicamente per massimizzare il *throughput*:

$$\text{chunksize} = \min\left(3, max(\frac{N_{blocks}}{N_{proc}}, 1)\right)$$

Questa formula punta a suddividere i blocchi in micro-batch per ogni core, che variano in un range da 1 a 3 in base al numero di blocchi. Questa euristica permette di ridurre il numero di operazioni di I/O inter-processuale e evitando idling.

**Shared Array e inizializzazione dei Worker** Nella fase RLE, è stata introdotta un'ottimizzazione tramite l'uso di **memoria condivisa** con `multiprocessing.RawArray`. Invece di passare liste di interi (l'output della bMTF) ai *worker*, il sistema opera come segue:

1. Viene allocato un `RawArray` di tipo intero (`'i'`) accessibile globalmente.
2. Tramite il parametro `initializer=init worker`, ogni processo figlio riceve il riferimento al buffer condiviso all'atto della creazione.
3. I task inviati ai *worker* contengono solo gli indici di *start* ed *end*.

**Shared Array e inizializzazione dei Worker** Nella fase RLE, è stata introdotta un'ottimizzazione tramite l'uso di **memoria condivisa** con `multiprocessing.RawArray`. Invece di passare liste di interi (l'output della bMTF) ai *worker*, il sistema opera come segue:

1. Viene allocato un `RawArray` di tipo intero (`'i'`) accessibile globalmente.
2. Tramite il parametro `initializer=init worker`, ogni processo figlio riceve il riferimento al buffer condiviso all'atto della creazione.
3. I task inviati ai *worker* contengono solo gli indici di *start* ed *end*.

Questo meccanismo evita la duplicazione dell'input in memoria e la serializzazione dei dati, consentendo alla pipeline di processare file di grandi dimensioni senza saturare le risorse di sistema.

L'utilizzo della memoria condivisa è stato limitato alla fase di RLE, in quanto i dati elaborati sono rappresentabili come interi e possono essere facilmente mappati su strutture condivise. Nella sBWT, invece, un array condiviso sarebbe stato controproducente: i dati stringa richiedono meccanismi di sincronizzazione tra compressione e decompressione, con overhead superiore ai benefici.

### 3.1.3 Algoritmo DC3 per la costruzione lineare del Suffix Array

L'ottimizzazione più significativa introdotta nella pipeline riguarda la transizione verso un algoritmo di costruzione del *Suffix Array* a complessità lineare $O(n)$. L'implementazione si basa sull'algoritmo **DC3** (*Difference Cover modulo 3*).

L'implementazione sviluppata nel modulo `suffix.py` si articola in due componenti principali: un *wrapper* di pre-processing e il nucleo ricorsivo dell'algoritmo.

**Pre-processing e Normalizzazione dei Ranghi** Poiché l'algoritmo DC3 richiede in input una sequenza di interi positivi, è stata progettata la funzione `buildSuffixArrayDC3` che agisce come interfaccia tra il layer di sicurezza e la logica algoritmica. Il processo di normalizzazione avviene come segue:

1. Viene generata una mappatura segreta tramite `customSort.getSecretSort`, che associa a ogni carattere dell'alfabeto un peso numerico di tipo *float* dipendente dalla chiave $K$.

2. Per permettere l'utilizzo del *Counting Sort* (che richiede indici interi), i pesi *float* vengono "compressi" in ranghi interi contigui $(1, 2, \ldots, \Sigma)$.

3. La stringa originale viene trasformata in un vettore di interi $s$, dove ogni elemento rappresenta il rango segreto del carattere corrispondente.

**Tripletta e Ricorsione** Il cuore dell'algoritmo `dc3(s, n, max)` implementa la strategia *divide-et-impera* basata sulla partizione degli indici in base al modulo 3:

- **Campionamento S12:** Vengono estratti gli indici $i$ tali che $i \pmod 3 \neq 0$. Questi vengono ordinati in tempo lineare considerando le triplette di caratteri $(s[i], s[i+1], s[i+2])$. L'ordinamento è garantito da tre passate consecutive di *Counting Sort* (una per ogni posizione della tripletta).

- **Naming e Ricorsione:** A ogni tripletta viene assegnato un "nome" (rango). Se i nomi non sono univoci, l'algoritmo viene invocato ricorsivamente sulla stringa dei nomi per risolvere le ambiguità dei suffissi con $i \pmod 3 \neq 0$.

- **Ordinamento S0:** Gli indici $i \pmod 3 = 0$ vengono ordinati in tempo $O(n)$ sfruttando l'ordine già calcolato per il gruppo S12. Un suffisso in $i \equiv 0$ è infatti univocamente determinato dalla coppia $(s[i], \text{rango}(i+1))$.

**Fase di Merging Segreto** L'unione dei due gruppi (`s12_sorted` e `s0`) avviene tramite un *merging* lineare. La complessità della comparazione tra un suffisso $p_{12} \in S_{12}$ e uno $p_0 \in S_0$ è ridotta a tempo costante $O(1)$ grazie alle informazioni pre-calcolate:

- Se $p_{12} \equiv 1 \pmod 3$, la comparazione utilizza la coppia $(s[p_{12}], \text{names}[p_{12}+1])$.

- Se $p_{12} \equiv 2 \pmod 3$, si utilizza la tripla $(s[p_{12}], s[p_{12}+1], \text{names}[p_{12}+2])$.

Tutte le operazioni di comparazione e ordinamento all'interno di `suffix.py` rispettano l'ordinamento imposto dalla chiave segreta. L'adozione del DC3, unita all'efficienza del Counting Sort, ha permesso di trasformare la costruzione del Suffix Array in un'operazione scalabile, rendendo la sBWT più efficiente su dataset di grandi dimensioni.

### 3.1.4 Counting Sort per l'ordinamento lineare

Il `Counting Sort` è un algoritmo di ordinamento non basato su confronti che opera in tempo lineare. A differenza di algoritmi come il Quick Sort o il Merge Sort, esso sfrutta le proprietà intrinseche dei dati (numeri interi) e richiede la conoscenza preventiva del range di valori in cui gli elementi dell'array sono distribuiti. Il principio fondamentale consiste nel calcolare, per ciascun elemento $x$, il numero di elementi nell'array di input che hanno valore uguale a $x$, per poi posizionare $x$ direttamente nella sua posizione finale [7]. La procedura si articola nelle seguenti fasi:

1. **Inizializzazione:** Si individuano i valori estremi del vettore $A$, ovvero $max(A)$ e $min(A)$. Viene quindi allocato un vettore ausiliario di supporto $C$ (detto array di conteggio) di dimensione pari alla differenza fra $max(A)$ e $min(A)$, inizializzato a zero.

2. **Calcolo delle frequenze:** Si effettua una scansione del vettore di input $A$. Per ogni valore incontrato, si incrementa il contatore corrispondente nell'array $C$.

3. **Ricostruzione del vettore:** Si scorre l'array $C$ sequenzialmente. Per ogni indice $i$ tale che $C[i] > 0$, il valore viene riscritto nel vettore output tante volte quante indicate dalla frequenza memorizzata, garantendo così l'ordinamento crescente.

**Esempio Applicativo** Si consideri il vettore $A = [2, 5, 2, 7, 8, 8, 2, 3, 6]$.

- **Analisi del range:** Il valore minimo è 2 e il massimo è 8. L'intervallo di riferimento è $[2, 8]$.

- **Frequenze (Array C):** Dopo la scansione, l'array dei conteggi risulterà:

$$C = [3, 1, 0, 1, 1, 1, 2]$$

  Dove $C[0]$ indica le occorrenze di 2, $C[1]$ quelle di 3, e così via. Ad esempio, il valore 4 ha frequenza 0 ($C[2] = 0$).

- **Output finale:** Ripercorrendo $C$, si ottiene la sequenza ordinata:

$$A_{ordinato} = [2, 2, 2, 3, 5, 6, 7, 8, 8]$$

**Analisi della Complessità** L'efficienza del Counting Sort è strettamente legata alla natura dei dati in input:

- **Tempo:** L'algoritmo richiede $O(n)$ per la scansione iniziale e il conteggio, e $O(k)$ per la ricostruzione del vettore, dove $k$ è l'ampiezza dell'intervallo. La complessità asintotica totale è dunque $O(n + k)$.

- **Spazio:** È richiesta memoria aggiuntiva $O(k)$ per l'array ausiliario.

L'algoritmo risulta estremamente performante (superiore agli algoritmi basati su confronti che sono vincolati a $\Omega(n \log n)$) quando l'intervallo dei valori $k$ è dello stesso ordine di grandezza della dimensione dell'input $n$. Qualora $k \gg n$, l'eccessivo consumo di memoria e il tempo di scansione di $C$ rendono l'algoritmo meno efficiente rispetto a soluzioni alternative. Nel nostro caso $k$ è la grandezza del dizionario quindi avrà sempre dimensioni molto inferiori a $n$.

## 3.2 bMTF

Il sistema è passato da una gestione statica a un'euristica dinamica per definire la dimensione dei blocchi nella bMTF. Originariamente, l'uso di un blocco fisso da 1024 byte, nonostante fosse molto sicuro dal punto di vista crittografico (grazie ai frequenti rimescolamenti del dizionario), frammentava eccessivamente i dati, riducendo la ridondanza e le sequenze di simboli uguali. Questo impattava sulla capacità della codifica RLE di individuare lunghe sequenze di zeri, riducendo di conseguenza il tasso di compressione.

Per risolvere il problema, è stata introdotta la seguente euristica:

$$Block\_size = \max\left(10\text{ KB}, \min\left(\frac{L}{N}, 8\text{ MB}\right)\right) \tag{3.1}$$

dove $L$ è la lunghezza del file e $N$ la dimensione dell'alfabeto. I limiti di 10 KB e 8 MB sono stati scelti in modo da bilanciare l'efficienza della memoria (similmente a quanto avviene nel calcolo dei blocchi nella sBWT) e la sicurezza crittografica.

## 3.3 RLE

Esaminando l'algoritmo RLE, abbiamo osservato che su file di grandi dimensioni i tempi di esecuzione risultavano molto elevati. Un'analisi dell'implementazione precedente ha evidenziato la possibilità di sviluppare una versione parallelizzabile dell'algoritmo, introducendo alcuni accorgimenti per la suddivisione e la successiva raccolta dei dati, sia in fase di compressione sia di decompressione.

### 3.3.1 Ottimizzazione della memoria tramite Shared Array

In un'architettura parallela basata su Python, uno dei colli di bottiglia più critici è rappresentato dal trasferimento di grandi moli di dati tra *Master* e *Workers*. Per ottimizzare la fase di codifica RLE nel modulo `compression.py`, è stata implementata una gestione della memoria basata su `Shared Arrays`, superando i limiti della comunicazione inter-processuale (*IPC*) standard. L'integrazione degli *Shared Arrays* trasforma la gestione della memoria da un sistema basato su scambi di messaggi a un'architettura a memoria condivisa, riducendo l'overhead di questa fase della pipeline per file molto grandi.

### 3.3.2 Parallelizzazione della RLE

Nella pipeline originale, la codifica *Run-Length Encoding* (RLE) veniva eseguita in modo sequenziale dopo la raccolta di tutti i dati trasformati. Tale approccio limitava il parallelismo e costringeva il sistema a trasferire grandi volumi di dati non compressi tra i processi. La soluzione proposta sposta la codifica RLE all'interno dei singoli task paralleli, adottando una suddivisione in blocchi coerente con l'euristica utilizzata per la sBWT 3.1.1. In questo modo, la compressione avviene localmente in ciascun processo, riducendo sia l'overhead di comunicazione sia la quantità di dati scambiati.

**Esecuzione distribuita** Ogni processo *worker* esegue la funzione `multi_rle_encode`. Grazie all'uso dei *Shared Array*, il worker non riceve una copia dei dati, ma accede direttamente ai valori della bMTF tramite indici di *offset*. Localmente, ogni processo produce una stringa RLE parziale che rappresenta la compressione del proprio blocco. Al termine dell'elaborazione, il processo *master* esegue un'operazione di concatenazione.

**Strategia di merging e gestione dei confini** Una volta completata la fase di compressione parallela, il processo principale deve combinare i risultati prodotti dai singoli blocchi. Ogni blocco genera una stringa RLE composta da una sequenza di coppie *conteggio–valore*, dove:

- il formato `count-value` viene utilizzato quando il conteggio è maggiore di uno;
- il solo `value` rappresenta implicitamente un conteggio pari a uno;
- le coppie sono separate da una virgola.

Una semplice concatenazione delle stringhe prodotte dai blocchi risulterebbe sub-ottimale, in quanto non consentirebbe di gestire correttamente le ripetizioni che si estendono a cavallo dei confini tra blocchi adiacenti. La funzione `rle_merge` implementata risolve questa criticità seguendo i passaggi riportati di seguito:

1. I risultati parziali vengono inizialmente riordinati in base all'indice originale del blocco, così da ripristinare l'ordine corretto del flusso di dati.
2. Durante la fase di concatenazione, l'algoritmo confronta l'ultima coppia RLE del blocco $i$ con la prima coppia del blocco $i + 1$, analizzandone il valore simbolico.
3. Qualora i valori coincidano, i rispettivi conteggi vengono combinati in un unico record RLE, sommando i conteggi associati e producendo una singola coppia *conteggio–valore*.

Questo approccio consente di preservare la correttezza semantica della codifica RLE anche in presenza di una suddivisione in blocchi, evitando frammentazioni delle sequenze ripetute e garantendo un risultato equivalente a quello di una compressione sequenziale.

**Algoritmo di split** Per creare blocchi indipendenti e sintatticamente corretti, l'algoritmo individua un punto di taglio approssimativo e lo perfeziona spostandosi verso la coppia RLE completa più vicina, delimitata dalla virgola, preservando così l'integrità delle coppie. Ogni segmento assegnato ai processi worker contiene solo coppie complete (count-value o value) e può essere decodificato autonomamente.

## 3.4 Variante MPI

Oltre all'implementazione basata sulla libreria `multiprocessing`, è stata sviluppata una variante che utilizza lo standard **MPI** tramite il pacchetto `mpi4py`. La variante implementata con MPI altera solo i meccanismi di parallelizzazione, preservando i principali algoritmi della pipeline originale.

### 3.4.1 Distribuzione dei dati e gestione dei blocchi

Nel modello `multiprocessing`, i dati vengono distribuiti tramite `Pool`:

- **Fase sBWT:** slicing della stringa di input e passaggio ai worker tramite `pickle`, con copia dei dati.
- **Fase RLE:** utilizzo di `RawArray` per evitare duplicazioni, consentendo accessi tramite indici.

Nella variante MPI, ogni processo lavora in memoria isolata: Il *Master* (Rank 0) calcola gli offset e le dimensioni dei blocchi, suddividendo l'input in segmenti da distribuire tra i processi tramite `comm.Scatterv`, consentendo l'assegnazione di blocchi di dimensioni variabili. Ogni processo, incluso il Master, suddivide ulteriormente il proprio segmento in sottoblocchi larghi 2MB, per evitare la saturazione della memoria e crash dovuti a blocchi eccessivamente grandi. Per garantire la coerenza tra compressione e decompressione, ogni processo registra la dimensione dei blocchi in un file di testo identificato dal proprio rank; durante la decompressione, ogni processo legge il file corrispondente per ricostruire correttamente le dimensioni dei blocchi da elaborare.

### 3.4.2 Raccolta e merging dei risultati

Nella versione `multiprocessing`, i risultati dei *Worker* vengono raccolti tramite `imap_unordered` e riordinati manualmente tramite un indice restituito da ogni processo.

In MPI, la primitiva `comm.Gather` assicura che i blocchi compressi vengano raccolti dal Master mantenendo l'ordine dei Rank, semplificando il merge finale.

### 3.4.3 Coerenza e sicurezza della pipeline

Per garantire che la Scrambled BWT sia invertibile, tutti i processi utilizzano la stessa permutazione dell'alfabeto segreta. Il seed casuale $r$ generato dal Master viene propagato a tutti i worker tramite `comm.bcast`, assicurando consistenza nelle elaborazioni parallele.

### 3.4.4 Vantaggi della variante MPI

Grazie alla gestione esplicita dei messaggi tra processi indipendenti, la variante MPI supera le limitazioni imposte dal Global Interpreter Lock (GIL) di Python e riduce l'overhead associato alla creazione e al monitoraggio dei processi, garantendo un parallelismo effettivo e più efficiente. Tale architettura permette di ottimizzare l'uso della memoria e ridurre i tempi complessivi di compressione e decompressione, semplificando al contempo il mantenimento della coerenza dei dati tra i nodi.

## 3.5 Ulteriori Ottimizzazioni

I tempi di esecuzione sono stati ulteriormente ridotti attraverso ottimizzazioni del codice relative alla gestione della memoria. In alcune fasi della pipeline, l'approccio inizialmente adottato era il seguente:

```
out_string = ""

while condizione:
    out_string += struttura_dati[indice]

return out_string
```

Questa modalità di costruzione della stringa di output risulta estremamente onerosa in termini di memoria e computazione, specialmente quando il file da elaborare presenta dimensioni significative. Per ovviare a questo problema, la stringa `out_string` è stata sostituita con una lista:

```
out_list = []

while condizione:
    out_list.append(struttura_dati[indice])

return "".join(out_list)
```

Tale modifica permette un uso più efficiente delle risorse di sistema pur mantenendo inalterato il formato di output originale.

# Capitolo 4

# Testing e Risultati

La fase di testing è stata condotta su tre dataset di dimensioni crescenti, rispettivamente di circa 100, 200 e 300 MB. I seguenti script eseguono compressione e decompressione, facendo un controllo sulla correttezza lossless e dando in output tempi di esecuzione e percentule di compressione:

- `tester.py` e `tester_mpi.py`: eseguiti fornendo in input il file sorgente, la chiave e la codifica;
- `tester_bzip.py`: utilizzato come termine di paragone, basato su un'implementazione built-in dell'algoritmo *bzip2*, tramite la libreria `bz2` che rappresenta lo stato dell'arte per la compressione basato su BWT.

## 4.1 Ambiente di esecuzione

- CPU: Ryzen 7 3750H
- RAM: 16GB DDR4
- Sistema Operativo: Windows 11 Pro
- Python: 3.13.5

## 4.2 Analisi dei risultati

### 4.2.1 Miglioramenti Ottenuti

Il passaggio dall'implementazione precedente (`Legacy`) alle due versioni dell'implementazione proposta, `MPI` e `Multiprocessing`, mostra un miglioramento prestazionale drastico. Nel caso del file da 300 MB, il tempo di compressione si riduce da circa 8193 secondi a 1012 secondi utilizzando MPI, performando leggermente meglio della libreria `multiprocessing`, dovuto ad una gestione più efficiente dell'overhead di comunicazione tra i processi.

### 4.2.2 Confronto con Bzip2

Nonostante le varie ottimizzazioni, **Bzip2** risulta avere comunque prestazioni molto migliori, sia in termini di tempi che rapporto di compressione. Il modulo `bz2` funge da wrapper per la libreria di basso livello `libbzip2`, scritta in C ed estremamente ottimizzata. Inoltre il layer di sicurezza introdotto comporta tempi di calcolo maggiori ed una riduzione nella percentuale di compressione, dovuto alla suddivisione in blocchi.

### 4.2.3 Rapporto di Compressione

Le implemetazioni proposte apportano un migliormento nel rapporto di compressione, dovuto alla migliore gestione dei blocchi in sBWT e bMTF, adattandosi all'input permettono di creare blocchi non troppo piccoli, e quindi mantenere abbastanza alto il grado di ridondanza e quindi compressione.

### 4.2.4 Anomalie Riscontrate

L'analisi dei tempi relativi al file `Test_200` mostra un incremento dei tempi di esecuzione ed un rapporto di compressione più alto, questo è probabilmente dovuto ad un grado di ridondanza elevato.

Tabella 4.1: Confronto prestazioni tra Bzip2, Legacy, MPI e Multiprocessing

| File | Metodo | T. Compressione (s) | T. Decompressione (s) | Compressione (%) |
|---|---|---|---|---|
| Test_100 | Bzip2 | 14.06 | 12.32 | 72.90% |
| | Legacy | 2651.68 | 3413.05 | 49.37% |
| | MPI | 519.99 | 251.83 | 56.72% |
| | Multiprocessing | 558.55 | 263.34 | 56.86% |
| Test_200 | Bzip2 | 25.02 | 15.98 | 73.22% |
| | Legacy | 5925.26 | 19283.88 | 52.93% |
| | MPI | 1090.65 | 639.22 | 58.41% |
| | Multiprocessing | 1132.86 | 666.69 | 58.46% |
| Test_300 | Bzip2 | 33.13 | 22.51 | 73.71% |
| | Legacy | 8193.44 | 16041.27 | 54.17% |
| | MPI | 1012.25 | 570.62 | 54.31% |
| | Multiprocessing | 1073.36 | 592.41 | 54.34% |

# Capitolo 5

# Conclusioni

Il presente lavoro di tesi ha avuto come obiettivo l'analisi e la mitigazione delle criticità prestazionali di un sistema di compressione sicura basato sulla *Scrambled Burrows-Wheeler Transform* (sBWT) e sulla *Blocky Move-to-Front* (bMTF). Partendo da un'implementazione di riferimento caratterizzata da colli di bottiglia algoritmici e limitazioni architetturali, il progetto ha portato allo sviluppo di una soluzione maggiormente scalabile e orientata al calcolo parallelo.
Le ottimizzazioni introdotte hanno interessato tutte le principali fasi della pipeline di compressione, consentendo miglioramenti significativi in termini di prestazioni ed efficienza complessiva.

- **Efficienza algoritmica:** La sostituzione dell'algoritmo di costruzione del Suffix Array, avente complessità $O(n \log^2 n)$, con l'algoritmo lineare **DC3** ha eliminato il principale limite alla scalabilità del sistema, rendendo possibile l'elaborazione di dataset di dimensioni dell'ordine delle centinaia di megabyte entro tempi computazionali accettabili.
- **Parallelismo e gestione delle risorse:** L'introduzione di strategie euristiche per la suddivisione dinamica dei blocchi ha consentito una migliore distribuzione del carico di lavoro sui core disponibili. Inoltre, la parallelizzazione della fase di *Run-Length Encoding* (RLE) ha contribuito a migliorare i tempi di esecuzione.
- **Confronto tra MPI e multiprocessing:** La realizzazione di due differenti varianti parallele ha permesso di evidenziare i vantaggi dell'approccio a scambio di messaggi basato su **MPI**. Questo modello garantisce prestazioni superiori e una gestione della memoria più efficiente rispetto al modulo `multiprocessing`.

I risultati ottenuti confermano l'efficacia delle migliorie proposte. In particolare, i tempi di compressione e decompressione sono stati ridotti sostanzialmente. Parallelamente, è stato osservato anche un lieve miglioramento del rapporto di compressione, attribuibile a una gestione più efficiente della frammentazione dei dati.

## 5.1 Considerazioni critiche e limitazioni

Nonostante i rilevanti progressi ottenuti, il confronto con **Bzip2** evidenzia la presenza di un divario prestazionale ancora significativo. Tale differenza è principalmente riconducibile a due fattori fondamentali:

1. **Overhead introdotto dai meccanismi di sicurezza:** Le operazioni di *scrambling* dell'alfabeto e la gestione delle permutazioni segrete impiegate nella sBWT e nella bMTF comportano un costo computazionale aggiuntivo.

2. **Limitazioni del linguaggio di implementazione:** Nonostante la parallelizzazione e le varie ottimizzazioni proposte, Python introduce un overhead dovuto all'interpretazione del codice e alla gestione degli oggetti, risultando meno efficiente rispetto all'implementazione nativa in C adottata dal modulo `bz2`.

## 5.2 Sviluppi futuri

I risultati ottenuti suggeriscono diverse possibili direzioni di miglioramento e approfondimento:

- **Porting in C/C++:** La riscrittura dei moduli computazionalmente più onerosi, in particolare l'algoritmo DC3 e la bMTF, in un linguaggio compilato consentirebbe di eliminare l'overhead dell'interprete Python, migliorando ulteriormente le prestazioni del sistema.

- **Esecuzione su cluster:** L'integrazione di MPI apre alla possibilità di valutare la scalabilità del sistema su infrastrutture di calcolo distribuito. Testing su cluster permetterebbero di analizzare le prestazioni su dataset di dimensioni molto maggiori.

In conclusione, il lavoro svolto introduce significative ottimizzazioni rispetto all'implementazione di riferimento, riducendo i tempi di esecuzione e garantendo la scalabilità su dataset di grandi dimensioni, senza compromettere la robustezza delle garanzie crittografiche.

# Bibliografia


[1] Gongxian Zeng et al. "Secure Compression and Pattern Matching Based on Burrows-Wheeler Transform". In: *2018 16th Annual Conference on Privacy, Security and Trust (PST)*. 2018, pp. 1–10. DOI: 10.1109/PST.2018.8514159.

[2] Juha Kärkkäinen, Peter Sanders e Stefan Burkhardt. "Linear work suffix array construction". In: *Journal of the ACM (JACM)* 53.6 (2006), pp. 918–936.

[3] *Move To Front Data Transform Algorithm*. https://www.geeksforgeeks.org/dsa/move-front-data-transform-algorithm/.

[4] Message P Forum. *MPI: A message-passing interface standard*. 1994.

[5] Mike Anderson. *The Cache Clash: L1, L2, and L3 in CPUs*. Accessed: 2026-02-07. Mag. 2025. URL: https://medium.com/@mike.anderson007/the-cache-clash-l1-l2-and-l3-in-cpus-2a21d61a0c6b (visitato il giorno 07/02/2026).

[6] parallelize.readthedocs.io. *Considerations with using multiprocessing*. Discusses overhead due to pickling and inter-process communication in Python's multiprocessing module. 2025. URL: https://parallelize.readthedocs.io/en/latest/considerations.html (visitato il giorno 02/02/2026).

[7] *Counting Sort*. https://it.wikipedia.org/wiki/Counting$_s ort$.