LM-40 - Scienze Computazionali

20410426 - IN480 - CALCOLO PARALLELO E DISTRIBUITO

CompactText – Codifica testi con identificatori univoci per parole ricorrenti

Studenti:

Sara Liñán Heredia

Antonio García Torres

Repository GitHub:

https://github.com/Tonips22/CompactText

Indice

1. Introduzione

Il presente lavoro è stato svolto da **Sara Liñán Heredia** e **Antonio García Torres** con l'obiettivo di implementare e confrontare **tre diverse versioni del programma**: una **sequenziale**, una parallela basata su **OpenMP** e una parallela basata su **MPI**, così da valutare **i benefici del parallelismo** e comprendere meglio le sfide legate alla gestione concorrente dei dati.

2. Obiettivi del progetto

Gli **obiettivi principali** che ci siamo prefissati sono stati:

- Realizzare un sistema che codifichi uno o più **file di testo** trasformando ogni **parola** in un codice **numerico univoco**.
- Mantenere intatta la struttura testuale, includendo **spazi**, **punteggiatura** e **caratteri speciali**, per garantire una **decompressione perfetta**.
- Implementare due operazioni fondamentali: **encoding** e **decoding**.
- **Parallelizzare** il processo per migliorare **l'efficienza** su **sistemi multicore**.
- Confrontare le **prestazioni** tra le versioni **sequenziale**, **OpenMP** e **MPI**.

3. Descrizione generale del sistema

Il sistema si basa su un **dizionario condiviso** che mappa ogni parola a un **identificatore intero univoco**. Durante la codifica, il testo viene letto parola per parola: se una parola non è presente nel dizionario viene aggiunta; in caso contrario, viene utilizzato l'identificatore esistente. Le parole vengono quindi sostituite dal relativo ID nel **file binario di output**, mentre i caratteri di punteggiatura e gli spazi sono codificati come valori negativi, per poterli riconoscere e ripristinare fedelmente nel file originale durante la decodifica.

4. Struttura e organizzazione del codice

Il progetto è organizzato in una struttura di directory chiara e modulare, pensata per facilitare la gestione del codice, dei dati di test e della documentazione. La seguente è una panoramica della gerarchia delle cartelle:

```
CompactText
├── data/                   # testi di prova
├── src/
│   ├── mpi/                # versione MPI
│   │   ├── mpi.c           # codice principale MPI
│   │   └── makefile        # makefile per compilare la versione MPI
│   ├── openmp/             # versione OpenMP
│   │   ├── omp.c           # codice principale OpenMP
│   │   └── makefile        # makefile per compilare la versione OpenMP
│   └── sequential/         # versione sequenziale
│       ├── seq.c           # codice principale sequenziale
│       └── makefile        # makefile per compilare la versione sequenziale
├── .gitignore
├── memory.docx             # Documento del progetto
└── README.md
```

5. Implementazioni

5.1 Versione Sequenziale

La versione sequenziale del programma esegue la codifica e la decodifica dei file **uno per uno**, **senza utilizzare il parallelismo**. Costituisce la base funzionale su cui sono state sviluppate le versioni parallele.

Codifica sequenziale (encode)

1. **Lettura del file di input**
   Il programma legge il contenuto **carattere per carattere**, formando parole e rilevando i separatori (spazi, segni di punteggiatura, ritorni a capo, ecc.).
2. **Gestione del dizionario**
   Per ogni parola letta:

- Se è già presente nel dizionario, viene scritto il suo ID.
- Se non è presente, viene aggiunta al dizionario e ne viene scritto il nuovo ID.

I separatori vengono scritti come **valori interi negativi** nel file binario.

3. **Output codificato**
   Il risultato è un **file binario** in cui le parole sono rappresentate dai loro ID e i separatori da interi negativi.
4. **Salvataggio del dizionario**
   Alla fine, il dizionario viene salvato in un file dict.txt per essere riutilizzato nella fase di decodifica.

Decodifica sequenziale (decode):

1. **Caricamento del dizionario**
   Il file dict.txt viene caricato, ricostruendo l'associazione tra ID e parole.
2. **Trasformazione del file binario in testo**
   Si legge il file codificato:

- Se il valore è positivo, viene scritta la parola corrispondente.
- Se è negativo, viene scritto il carattere separatore originale.

5.2 Versione OpenMP

Questa versione utilizza la libreria OpenMP per parallelizzare il processo di codifica e decodifica dei file. Questa implementazione crea un dizionario indipendente per ogni thread, eliminando la necessità di sincronizzazione e migliorando le prestazioni.

Codifica parallela (encode)

1. **Dizionari locali per thread**
   Viene riservato un dizionario per ciascun thread, evitando conflitti nella scrittura di nuove parole durante la codifica.
2. **Parallelizzazione con OpenMP**
   Ogni thread elabora uno o più file utilizzando il proprio dizionario. Questo viene realizzato con un ciclo `#pragma omp for` con pianificazione dinamica per bilanciare il carico tra i thread.
3. **Fusione finale dei dizionari**
   Una volta terminati tutti i thread, il **thread master** (`#pragma omp single`) fonde tutti i dizionari locali nel `global_dict`, evitando duplicati.

   Questa fase è **sequenziale**, ma molto veloce, in quanto opera solo sul contenuto già accumulato. Alla fine, il dizionario globale viene salvato in `dict.txt`.

4. **Liberazione della memoria**
   Dopo la fusione, la memoria dei dizionari locali viene liberata per evitare perdite.

Decodifica parallela (decode):

1. Il dizionario globale viene caricato dal disco (dict.txt).
2. I file binari vengono elaborati in parallelo: ogni thread trasforma il proprio file in testo utilizzando il dizionario.
3. Poiché i file vengono elaborati in modo indipendente, non ci sono conflitti né necessità di sincronizzazione.

5.3 Versione MPI.

Questa versione utilizza **MPI (Message Passing Interface)** per distribuire la codifica e la decodifica tra più processi. Ogni processo lavora **in modo indipendente** su un sottoinsieme dei file, e i dizionari locali vengono successivamente **fusi** dal processo master.

Codifica parallela (encode)

1. **Distribuzione dei file (round-robin):**
   I file vengono distribuiti equamente tra i processi. Ogni processo li codifica in parallelo, senza bisogno di sincronizzazione.
2. **Codifica locale e misurazione dei tempi:**
   - Le parole vengono lette dai file.

    - Si consultano e aggiornano i dizionari locali.
    - Le parole sono codificate come interi (ID), i separatori come interi negativi.
    - Si misura il tempo di esecuzione.
3. **Fusione dei dizionari (solo processo 0):**
    - I processi non-master inviano i propri dizionari al **rank 0**.
    - Il processo master li riceve e li fonde evitando duplicati.
    - Il dizionario globale viene salvato in dict.txt.
    - Questo passaggio è sequenziale ma veloce.

Decodifica parallela (decode)

1. **Caricamento e broadcast del dizionario:**
   Il **processo 0** carica dict.txt e lo invia agli altri processi tramite MPI_Bcast.
2. **Decodifica indipendente:**
    - Ogni processo decodifica i propri file binari in parallelo.
    - I token vengono interpretati: positivi come parole, negativi come separatori.
    - Si misura il tempo di esecuzione per ogni processo.

6. Problemi incontrate e soluzioni adottate

6.1 Generale

Problemi

- **Conversione da C++ a C**: Il codice originale era in C++ ed è stato riscritto completamente in C per evitare dipendenze esterne e semplificare l'uso con compilatori come mpicc.
- Errori iniziali nel programma:
    - A capo.
    - Caratteri speciali e separatori.
    - Parole contigue senza spazio (per esempio: "parola,altra").
- **Problemi di lettura e scrittura binaria**: Ci sono stati errori nella lettura/scrittura degli ID e dei separatori come interi, specialmente in architetture diverse (endianess).

Soluzioni

- Riscrittura completa in C per miglior compatibilità e controllo su strutture e memoria.
- Debug dell'analizzatore lessicale per garantire la corretta gestione di caratteri speciali, a capo e separatori.

6.2 OpenMP

Problemi

- Progettazione iniziale **poco efficiente**: All'inizio, tutti i thread condividevano un unico dizionario globale protetto da lock, causando alta contenzione e rallentamenti significativi (la versione parallela era fino a 4 volte più lenta della sequenziale).
- **Collo di bottiglia** nella sezione critica: L'accesso concorrente al dizionario impediva la scalabilità.
- **Parallelizzazione eccessiva** (file + all'interno dei file): Si è tentato di parallelizzare sia l'elaborazione dei file sia delle parole all'interno di ciascun file, risultando controproducente per l'eccesso di sincronizzazione.
- **Prestazioni scarse** in generale: Anche con #pragma omp parallel for, l'uso di lock e strutture condivise riduceva il guadagno in velocità.

Soluzioni

- Ogni thread usa il proprio dizionario locale, eliminando i lock.
- Fusione dei dizionari locali alla fine della codifica, eliminando la contenzione.
- Uso di #pragma omp for schedule(dynamic) per un miglior bilanciamento del carico tra i thread.

6.3 MPI

Problemi

- Non si è riusciti a **parallelizzare completamente** a livello intra-file:
  I file vengono solo distribuiti tra i processi, ma non si effettua parallelizzazione interna a ciascun file.
- **Fusione dei dizionari** nel master (rank 0): Anche se funzionale, questa parte è sequenziale e può diventare un collo di bottiglia con molti processi o molte parole.
- **Ridondanza di parole tra processi**: Dato che ogni processo crea il proprio dizionario, è comune trovare parole duplicate che devono poi essere filtrate nella fusione finale.
- **Limitazione** dovuta all'ambiente di esecuzione: A causa delle restrizioni del computer disponibile, non è stato possibile eseguire la versione MPI con un numero elevato di processi (per esempio, 8 o più).

Soluzioni

- Distribuzione dei file con strategia round-robin tra i processi (rank).
- Ogni processo codifica i propri file e genera il proprio dizionario.
- Il processo rank 0 riceve e fonde i dizionari di tutti i processi, eliminando i duplicati.
- Uso di MPI_Bcast per condividere il dizionario all'avvio della decodifica.

7. Analisi delle prestazioni

7.1 Obiettivo dei test

L'obiettivo di questa analisi è confrontare le tre versioni implementate del programma, sequenziale, OpenMP e MPI, in termini di prestazioni, considerando diversi scenari di dimensione dei file di input e diversi livelli di parallelismo.

7.2 Scenari di test

I test sono stati effettuati su tre scenari distinti:

- **Scenario S (Small):** 10 file di testo di circa 1 MB ciascuno (10 MB totali).
- **Scenario M (Medium):** 6 file di circa 5 MB (30 MB totali).
- **Scenario L (Large):** un singolo file da 50 MB.

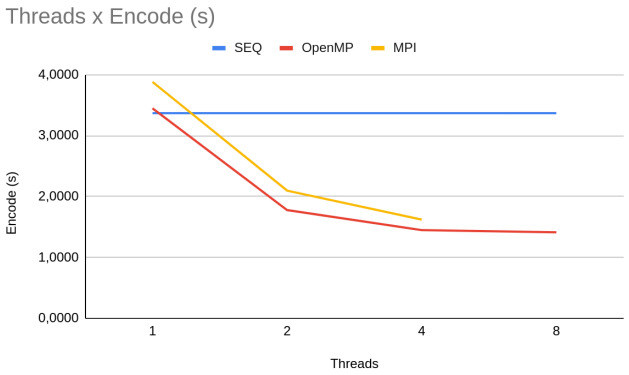
A causa di limitazioni hardware, non è stato possibile utilizzare **8 processi MPI**.

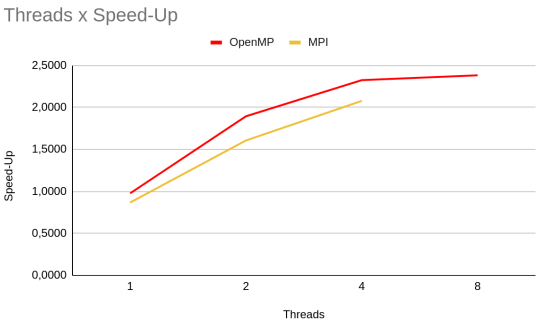
7.3 Metodologia

Per ogni combinazione di scenario, versione del programma e numero di thread/processi, sono state effettuate 3 ripetizioni, misurando il tempo totale di codifica (encode) e decodifica (decode). I tempi sono stati raccolti in secondi e poi mediati.

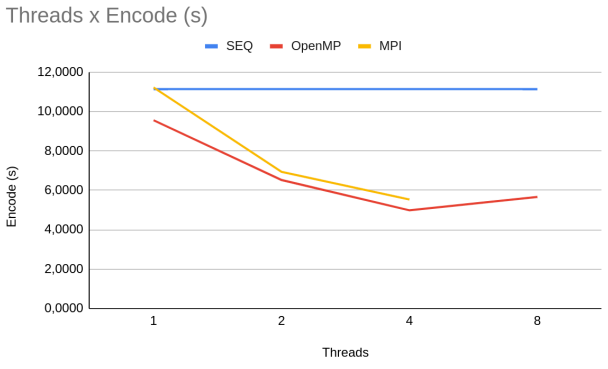
7.4 Risultati

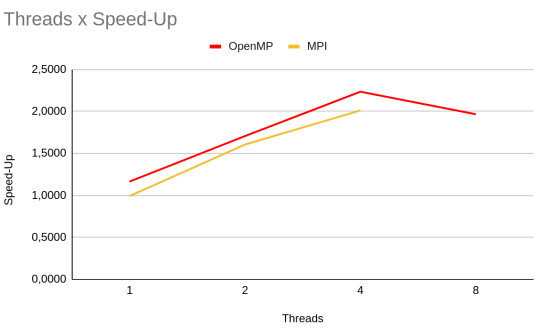
Scenario S (Small)



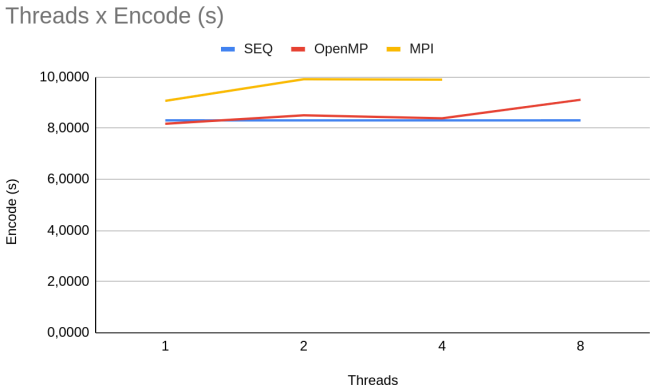


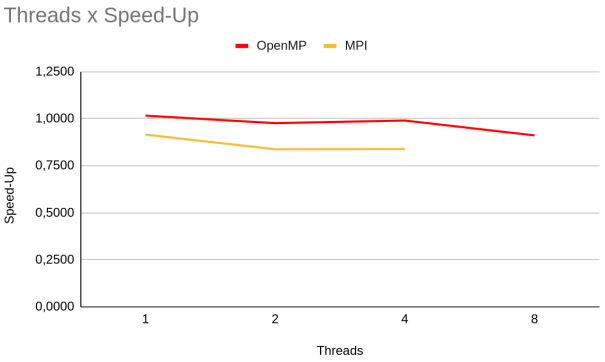
Scenario M (Medium)





Scenario L (Large)





7.5 Analisi dei risultati

Scenario S:

- La **versione sequenziale** è già veloce. I file sono così piccoli che il guadagno della parallelizzazione è **limitato**.
- **OMP migliora notevolmente** tra 1 e 4 thread, ma con 8 thread **non ottiene quasi più benefici**. Questo suggerisce che **l'overhead del parallelismo comincia a superare il guadagno**.
- **MPI ha prestazioni peggiori rispetto a OMP in tutto lo scenario**. Questo è prevedibile, poiché il costo della comunicazione e della sincronizzazione tra processi MPI è troppo alto per compiti così leggeri.

Scenario M:

- Già con 1 thread, **OMP migliora leggermente la versione sequenziale**.
- Lo **speed-up di OMP è molto buono con 2 e 4 thread**: si nota che il lavoro è abbastanza grande da ammortizzare il costo del parallelismo.
- A partire da 4 thread, OMP **inizia a peggiorare**: con 8 thread **impiega più tempo rispetto a 4**.
- **MPI migliora sensibilmente a partire da 2 processi**, ma non supera mai OMP.

Scenario L:

- Essendo **un unico file molto grande**, le versioni parallele **non riescono a dividerlo efficientemente senza frammentare artificialmente il lavoro**.
- **L'I/O (lettura e scrittura)** diventa il collo di bottiglia. Se non è parallelizzato, **non c'è un guadagno reale.**

Per vedere tutti i test: **CompactText Tests**

8. Conclusioni

Il progetto CompactText ha permesso di esplorare in modo approfondito l'elaborazione efficiente dei testi mediante la codifica delle parole con identificatori univoci. Attraverso lo sviluppo di tre diverse versioni del sistema è stato possibile valutare in maniera comparativa i benefici del parallelismo su architetture multicore e distribuite.

I risultati hanno mostrato che, sebbene la versione sequenziale costituisca una base solida, le versioni parallele (in particolare quella basata su OpenMP) offrono miglioramenti significativi in termini di efficienza nell'elaborazione di più file. Tuttavia, sono emerse anche sfide rilevanti legate alla sincronizzazione, alla progettazione efficiente di strutture dati condivise e alla scalabilità, soprattutto nel caso dell'implementazione con MPI.

In sintesi, CompactText ha raggiunto in modo soddisfacente gli obiettivi prefissati. Oltre ai risultati tecnici, il progetto ha rappresentato un'importante occasione di apprendimento, permettendoci di approfondire i concetti di parallelismo, distribuzione dei compiti e le difficoltà pratiche che emergono nell'implementazione di soluzioni efficienti in ambienti reali.