Relazione progetto per esame di progettazione di sistemi embedded e multicore

Davide Belcastro,1962536

January 8, 2023

1 Introduzione

Il progetto riguarda lo string matching eseguito tramite MPI e OpenMP(progetto 1).
Per la ricerca di match ho usato il noto algortimo Knuth-Morris-Pratt il cui codice sequenziale l'ho preso in rete.
Di seguito illustrerò ad alto livello l'approccio usato per parallelizzare la ricerca di match; commenterò in maniera più dettagliata tutti gli script presenti; illustrerò le difficoltà incontrate in fase di programmazione per entrambi le architetture e concluderò commentando gli output di un esempio di simulazione illustrati tramite screenshot.

2 Struttura

Nella cartella "progetto" sono presenti due sotto cartelle omp e mpi le quali contengono i relativi sorgenti necessari per il funzionamento delle corrispondenti architetture.
E' presente inoltre una sotto cartella "file" che contiene due sotto cartelle, la prima ha 6 file di testo diversi (per contenuto e dimensioni) e un file pattern contenente n stringhe che vanno cercate all'interno dei 6 testi diversi.
Nella seconda cartella invece sono presenti 6 file pattern e un file testo,in questo caso sono i file pattern diversi nel contenuto e nelle dimensioni,e bisogna cercare le diverse stringhe ed eventuali match nello stesso file di testo.

3 Descrizione algoritmo

Dato un testo T, un insieme S di stringhe pattern e un numero di thread N, l'approccio usato è stato il seguente:
- funzione che calcola il numero di righe che ogni thread deve fare;
- ogni thread calcola (tramite il suo id) l'indice di inizio e di fine nel file di testo;
- ogni thread controlla quante volte fa match una stringa nel suo segmento e ritorna la somma in una sua variabile privata;
- sommo tutte le variabili private e ottengo il numero di match totali di una stringa in un file.

Essendo presenti più file di input, itero su ognuno di essi ripetendo il suddetto algortimo.
Per vedere le differenze tra i vari tempi di esecuzione al variare del numero di thread e delle dimensioni dell'input, ho deciso di creare uno script che eseguisse questo algortimo con 1,2,4,8,16,32 thread.
Ogni esecuzione stampa su output il tempo di esecuzione specificando il numero di thread e il file di input.
Essendo presenti due tipi di input: variazione nel file di testo e variazione nel file pattern,tutto questo viene ripetuto due volte.
Lo script, quindi, crea 4 tabelle: speedup al variare delle dimensioni del file di testo,speedup al variare delle dimensioni del file pattern,efficency al variare delle dimensioni del file di testo,efficency al variare delle dimensioni del file pattern.
Lo script, alla fine, esegue il controllo correttezza tra le esecuzioni sequenziali e parallele tramite il

comando "diff".
Il comando diff viene eseguito sui file match che crea ogni singola esecuzione con un determinato numero di thread. In questi file viene scritto il numero di volte che le stringhe fanno match nei file.
Sono presenti due script,uno che implementa l'architettura openMP e l'altro MPI.
Per automatizzare il tutto ho fatto un ulteriore script che esegue questi due script e infine effettua un controllo correttezza tra altri due file match di openmp e mpi che vengono creati dalle sole iterazioni sequenziali. Questi file scrivono l'indice di riga in cui trovano i match. Non ho potuto creare questi file anche per esecuzioni parallele in quanto, oltre a essere not-safety scrivere su un file in parallelo, l'esecuzione di thread è non deterministica quindi il thread x+1 potrebbe scrivere prima del thread x e il file risulterebbe diverso(per ordine di righe) dal file generato dall'esecuzione sequenziale.

4 Output

Le tabelle(file .csv) hanno le righe e le colonne che rappresentano il numero di thread e la dimensione dell'input,ad una determinata cella è possibile leggere lo speedup ed efficency con numero di thread e dimensione dell'input.
Il programma stampa su output il tempo di esecuzione impiegato da ogni singola esecuzione con n thread al file j-esimo.
Ricordo che:
Speedup(n,p) = tempo esecuzione sequenziale con input dim. = n/tempo esecuzione parallelo su input con dim = n e num thread = p
Efficency(n,p) = speedup(n,p) / (numero di thread = p)

5 Requisiti

Per eseguire il programma è necessario aver installato mpi e openmp nonchè il compilatore c, essendo l'intero programma scritto in c.
Poichè sono presenti script bash e systemcall di linux è necessario eseguire l'applicativo su sistema operativo linux.

6 Istruzioni per eseguirlo

Per eseguire MPI o openMP è sufficiente eseguire lo script file.sh "m". Il file si trova nelle corrispettive directory.
"m" rappresenta il numero di volte che ripeto l'intero controllo match su uno stesso file.
Serve per evidenziare i possibili miglioramenti dovuti all'esecuzione parallela in quanto il tempo di esecuzione stesso viene calcolato alla fine dell'ultima esecuzione,e le stesse tabelle riportano la dimensione del file come numero di righe * m.
Esempio: m = 2 allora ogni esecuzione con un numero di thread specifico, per ogni file, effettua due volte il controllo.
Per eseguire sia omp che mpi invece bisogna eseguire il file.sh presente nella directory progetto,passando sempre m.
Con m = 5 la durata totale è di 15 minuti,si notano sostanziali differenze con thread differenti in termini di tempo di esecuzione,mentre con m = 2 la durata totale è di 4 minuti e le differenze sono presenti ma lievi.

7 File sorgenti

In questo capitolo descrivo tutti i file sorgenti necessari per il funzionamento,sia con architettura openmp che mpi.

7.1 Directory omp

In questa cartella abbiamo due file .c
Il file "find-match-omp-p.c" ha una funzione "run" che legge il file pattern e per ogni riga letta,chiama

la funzione "analizzastringa" che prende in input la riga corrente,il nome del file di testo,il numero di thread usati,ritorna il numero di match trovati per quella stringa,scrive sul file di match quante volte occorre quella stringa.
La funzione "analizzastringa" esegue in parallelo tramite la direttiva pragma omp parallel la ricerca nel testo di quella stringa,ogni thread prende un numero di righe equamente distribuito, e ogni thread chiama la funzione di KMP per la ricerca di eventuali match,la divisione del lavoro tra thread viene fatta dalla funzione settaRange che salva su variabili globali quante righe devono fare tutti i thread,poi in base all'indice del rank ricavo,per ogni thread,il suo inizio e la sua fine.
Uso la direttiva atomic per incrementare la variabile globale che indica il numero di match per una determinata stringa, necessito di gestire gli accessi in scrittura essendo openMP un architettura a memoria condivisa.
File "main.c": nella funzione main mi salvo il numero di cicli passato in input("m") e chiamo le due funzioni che creano le tabelle.
Le due funzioni ciclano su ogni file,per ogni iterazione prendono un file diverso(nella funzione per la costruzione della tabella al variare delle dimensioni del file di testo,mi prenderò quest'ultimo file che cambia ad ogni iterazione) nell'altra funzione invece vado a selezionare ogni volta il file pattern diverso,mentre fisso il file pattern nel primo caso e il file testo nel secondo.
Dentro il ciclo,una volta scelti i file, ciclo per i singoli thread e all'interno di questo ciclo itero (m volte) la funzione run passando in input(thread count(preso dal ciclo più interno),file testo,file pattern),mi salvo il tempo totale di esecuzione del programma con questo numero di thread e lo stampo su output insieme al nome del file. Inserisco nelle tabelle lo speedup e efficiency,se lancio con thread num = 1 allora mi salvo in una variabile il tempo di esecuzione in modo da poterlo riusare in seguito per il calcolo dello speedup.

7.2 Directory mpi

A differenza di openMP, MPI è un sistema a memoria distribuita,per scambiare informazioni tra i vari thread ho dovuto fare MESSAGE PASSING.
Il file "run.c" calcola le dimensioni di tutti i file(in termini di numero di righe) e crea le prime righe delle tabelle,dopo di chè si occupa di compilare e lanciare (tramite system call di linux) il programma "run-mpi-s.c" per l'esecuzione con un thread e poi, per tutti gli altri thread, "run-mpi-p.c".
Il file "run-mpi-s.c" ha due funzioni che creano prima le tabelle al variare delle dimensioni del file di testo, e poi le tabelle al variare delle dimensioni dei pattern,in entrambi i casi eseguono un ciclo che itera su tutti i file, ad ogni passo eseguono per m volte la funzione che controlla,dato un file pattern e un file di testo le occorrenze,dopo di che scrive su file i tempi di esecuzione,speedup,etc..
La funzione che controlla i pattern è la stessa usata nell'architettura omp,soltanto che usa un thread solo quindi non ha parallelizzazione.
Il file "run-mpi-p.c", invece prende in ingresso il numero di thread con cui viene lanciato e, se sono il thread 0 avvio le solite funzioni di creazione delle tabelle,altrimenti eseguo, per tutte le volte che è necessario eseguire, delle recive e send. Quest'ultime sono necessarie per scambiare i dati che servono per fare i controlli relativi alla porzione del thread in esame. Successivamente i thread != 0 inviano tramite send le occorrenze trovate.
Il numero di volte che esegue le recv e send è calcolato cosi: ho un ciclo di due iterazioni perchè ripeto per la tabella al variare delle dimensioni pattern e al variare del file di testo, all'interno ho un ciclo perchè ho 6 file in entrambi i casi,ancora all'interno ho un ciclo che viene eseguito m volte, questo è il numero di iterazioni totali che mi servono per rispondere ad ogni send che mi invia il thread 0.
Le funzioni di creazione della tabella sono molto simili,l'unica differenza sta nel fatto che in mpi ho invertito le righe e le colonne rispetto a openmp,questo perchè in mpi,lanciando ogni volta il programma con un numero di thread definito,mi veniva più comodo inserire una linea che avesse tutti i tempi di esecuzione di uno stesso thread con file di input diversi,quindi necessitavo di avere i thread nelle righe e i file nelle colonne,in modo da aggiungere una riga j -esima che faceva riferimento al thread j-esimo.

8 Difficoltà incontrate

Le prime difficoltà incontrate le ho avute nel rendere efficiente il programma parallelizzando l'algoritmo KMP,avevo parallelizzato sia la costruzione della tabella(cioè la prima parte dell'algortimo) che la

seconda parte,in entrambi i casi ottenevo un leggero miglioramento(quasi nullo) quando avevo una stringa pattern enorme,altrimenti avevo un peggioramento anche abbastanza grande,di conseguenza ho cambiato strategia e ho parallelizzato la ricerca del pattern nel file,dividendo le righe di testo.
Scelta la soluzione,con openMP non ho avuto molte difficoltà in quanto si presta per sua natura ottimo per l'algortimo ideato da me,necessitavo di un programma che fosse globalmente sequenziale e localmente parallelo(il programma parte sequenziale ma,alcune parti di codice vengono eseguite in parallelo effettuando fork e join), openMP possiede tutte queste caratteristiche e implementare un programma del genere è stato abbastanza facile,discorso diverso per MPI, che è GPLS,questo implica che il programma di per sè nasce per essere parallelo.
Per questo motivo ho dovuto scrivere un file che attraverso la system call di linux "system" lanciasse il programma specificando il numero di thread per ogni esecuzione.
Infine ho cercato di simulare la stessa soluzione fatta per openMP facendo "if thread != 0 attendi tutte le informazioni utili per fare la tua parte di testo".

9 Esempio di output con test.sh 5

Premessa: file1 indica il file di dimensione maggiore,file6 il più "leggero".
Analizzando le tabelle e i tempi di esecuzione del programma si può osservare come con uno stesso file in input,il progrmma con 1 thread impiega molto di più rispetto a quando viene lanciato con 2 thread o 4(ovviamente se il file ha dimensione molto grandi,altrimenti il vantaggio in termini di tempo di esecuzione non si nota), aumentando ancora il numero di thread ad 8 il tempo di esecuzione migliora di poco se non per niente, con 16 e 32 i tempi sono anche più alti rispetto ad un thread e questo è in linea con quanto visto durante l'intero corso e con la legge di Ambdhal,la velocità di esecuzione di un algortimo non dipende dal numero di thread con cui viene eseguito ma dall'algortimo stesso, il tempo si riduce fino ad un certo punto,dopo di che o rimane stabile o addirittura,aumentando di tanto il numero di thread, si rischia di avere un peggioramento.
Inoltre possiamo vedere come passando da un file ad un altro(con uno stesso numero di thread) i tempi si riducono, questo è dovuto al fatto che hanno dimensioni più piccole,solo con il file pattern6 abbiamo un aumento poichè,anche essendo il più piccolo, ha delle stringhe molto presenti nel file,e questo comporta un rallentamento nell'algortimo KMP in quanto trova spesso un match che poi va a scrivere nel file match,accedendo molto volte in scrittura,causando un aumento di tempo.

9.1 openMP

Vediamo nel dettaglio le tabelle e i tempi di esecuzione del programma che usa l'architettura openMP che prende in input i file presenti nella cartella.
Vedere Foto 1, 2, 3 e 4.
Esempio, il programma con input "file1.txt" come file di testo impiega 17 secondi,per vedere se i valori nelle tabelle funzionano,vediamo che il programma con 2 thread e file1 impiega 10 secondi,ma allora lo speedup con input file 1(il più grande) e numero di thread = 2 dovrebbe essere(tempo esecuzione sequenziale con file 1 / tempo esecuzione con num thread = 2 e file 1) = 1,7, se guardiamo la foto 3 possiamo vedere come nella cella che rappresenta lo speedup con due thread e file1(cioè il più in alto nella tabella) è proprio 1,7. Mentre nella seconda tabella alla foto 4, nella stessa cella dovremmo avere (speedup / numero di thread) = 0,85, che è il valore riportato.

9.2 MPI

Vediamo nel dettaglio le tabelle e i tempi di esecuzione del programma che usa l'architettura MPI che prende in input i file presenti nella cartella.
Vedere Foto 5, 6, 7, 8, 9, 10, 11 e 12.
Esempio, il programma con input "file1.txt" come file di testo impiega 19 secondi,per vedere se i valori nelle tabelle funzionano,vediamo che il programma con 2 thread e file1 impiega 10 secondi,ma allora lo speedup con input file 1 e numero di thread = 2 dovrebbe essere(tempo esecuzione sequenziale con file 1 / tempo esecuzione con num thread = 2 e file 1) = 1,9, se guardiamo la foto 11 possiamo vedere come nella cella che rappresenta lo speedup con due thread e file1(cioè la colonna più a sinistra nella

```
num_thread = 1 e tempo = 17.000000 e file 1
num_thread = 2 e tempo = 10.000000 e file 1
num_thread = 4 e tempo = 8.000000 e file 1
num_thread = 8 e tempo = 11.000000 e file 1
num_thread = 16 e tempo = 11.000000 e file 1
num_thread = 32 e tempo = 16.000000 e file 1
num_thread = 1 e tempo = 18.000000 e file 2
num_thread = 2 e tempo = 11.000000 e file 2
num_thread = 4 e tempo = 8.000000 e file 2
num_thread = 8 e tempo = 12.000000 e file 2
num_thread = 16 e tempo = 9.000000 e file 2
num_thread = 32 e tempo = 15.000000 e file 2
num_thread = 1 e tempo = 16.000000 e file 3
num_thread = 2 e tempo = 10.000000 e file 3
num_thread = 4 e tempo = 7.000000 e file 3
num_thread = 8 e tempo = 11.000000 e file 3
num_thread = 16 e tempo = 9.000000 e file 3
num_thread = 32 e tempo = 12.000000 e file 3
num_thread = 1 e tempo = 14.000000 e file 4
num_thread = 2 e tempo = 9.000000 e file 4
num_thread = 4 e tempo = 6.000000 e file 4
num_thread = 8 e tempo = 9.000000 e file 4
num_thread = 16 e tempo = 8.000000 e file 4
num_thread = 32 e tempo = 10.000000 e file 4
num_thread = 1 e tempo = 10.000000 e file 5
num_thread = 2 e tempo = 6.000000 e file 5
num_thread = 4 e tempo = 5.000000 e file 5
num_thread = 8 e tempo = 7.000000 e file 5
num_thread = 16 e tempo = 6.000000 e file 5
num_thread = 32 e tempo = 8.000000 e file 5
num_thread = 1 e tempo = 6.000000 e file 6
num_thread = 2 e tempo = 4.000000 e file 6
num_thread = 4 e tempo = 3.000000 e file 6
num_thread = 8 e tempo = 6.000000 e file 6
num_thread = 16 e tempo = 4.000000 e file 6
num_thread = 32 e tempo = 5.000000 e file 6
num_thread = 1 e tempo = 9.000000 e file 1
num_thread = 2 e tempo = 5.000000 e file 1
num_thread = 4 e tempo = 4.000000 e file 1
num_thread = 8 e tempo = 5.000000 e file 1
num_thread = 16 e tempo = 4.000000 e file 1
num_thread = 32 e tempo = 5.000000 e file 1
num_thread = 1 e tempo = 3.000000 e file 2
num_thread = 2 e tempo = 1.000000 e file 2
num_thread = 4 e tempo = 1.000000 e file 2
num_thread = 8 e tempo = 3.000000 e file 2
num_thread = 16 e tempo = 2.000000 e file 2
num_thread = 32 e tempo = 2.000000 e file 2
num_thread = 1 e tempo = 1.000000 e file 3
num_thread = 2 e tempo = 1.000000 e file 3
num_thread = 4 e tempo = 1.000000 e file 3
num_thread = 8 e tempo = 2.000000 e file 3
num_thread = 16 e tempo = 0.100000 e file 3
num_thread = 32 e tempo = 2.000000 e file 3
```

Figure 1: Output del programma con architettura openmp

tabella, con dimensione maggiore) è proprio 1,9. Mentre nella seconda tabella alla foto 12, nella stessa cella dovremmo avere (speedup / numero di thread) = 0,95, che è il valore riportato.

```
num_thread = 1 e tempo = 2.000000 e file 4
num_thread = 2 e tempo = 2.000000 e file 4
num_thread = 4 e tempo = 1.000000 e file 4
num_thread = 8 e tempo = 1.000000 e file 4
num_thread = 16 e tempo = 2.000000 e file 4
num_thread = 32 e tempo = 1.000000 e file 4
num_thread = 1 e tempo = 1.000000 e file 5
num_thread = 2 e tempo = 0.100000 e file 5
num_thread = 4 e tempo = 1.000000 e file 5
num_thread = 8 e tempo = 0.100000 e file 5
num_thread = 16 e tempo = 1.000000 e file 5
num_thread = 32 e tempo = 0.100000 e file 5
num_thread = 1 e tempo = 19.000000 e file 6
num_thread = 2 e tempo = 13.000000 e file 6
num_thread = 4 e tempo = 7.000000 e file 6
num_thread = 8 e tempo = 7.000000 e file 6
num_thread = 16 e tempo = 7.000000 e file 6
num_thread = 32 e tempo = 6.000000 e file 6
Analisi correttezza tra result_parallelo_1.txt e result_parallelo_2.txt:ok
Analisi correttezza tra result_parallelo_1.txt e result_parallelo_4.txt:ok
Analisi correttezza tra result_parallelo_1.txt e result_parallelo_8.txt:ok
Analisi correttezza tra result_parallelo_1.txt e result_parallelo_16.txt:ok
Analisi correttezza tra result_parallelo_1.txt e result_parallelo_32.txt:ok
```

Figure 2: Output del programma con architettura openmp

Tabella speedup variazione dimensione file di testo						
size/num_thread	1	2	4	8	16	32
386040	1.000000	1.700000	2.125000	1.545455	1.545455	1.062500
334010	1.000000	1.636364	2.250000	1.500000	2.000000	1.200000
299570	1.000000	1.600000	2.285714	1.454545	1.777778	1.333333
253890	1.000000	1.555556	2.333333	1.555556	1.750000	1.400000
180550	1.000000	1.666667	2.000000	1.428571	1.666667	1.250000
118650	1.000000	1.500000	2.000000	1.000000	1.500000	1.200000

Figure 3: Tabella speedup con openmp

Tabella efficiency variazione dimensione file di testo						
size/num_thread	1	2	4	8	16	32
386040	1.000000	0.850000	0.531250	0.193182	0.096591	0.033203
334010	1.000000	0.818182	0.562500	0.187500	0.125000	0.037500
299570	1.000000	0.800000	0.571429	0.181818	0.111111	0.041667
253890	1.000000	0.777778	0.583333	0.194444	0.109375	0.043750
180550	1.000000	0.833333	0.500000	0.178571	0.104167	0.039062
118650	1.000000	0.750000	0.500000	0.125000	0.093750	0.037500

Figure 4: Tabella efficiency con openmp

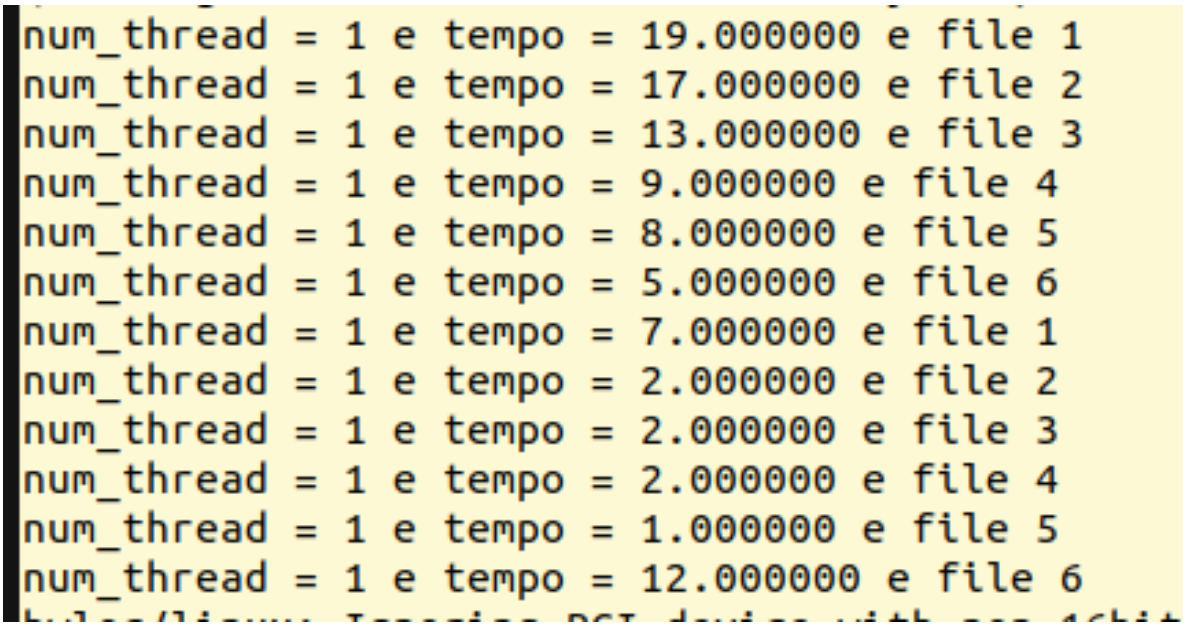

Figure 5: Output del programma con architettura mpi

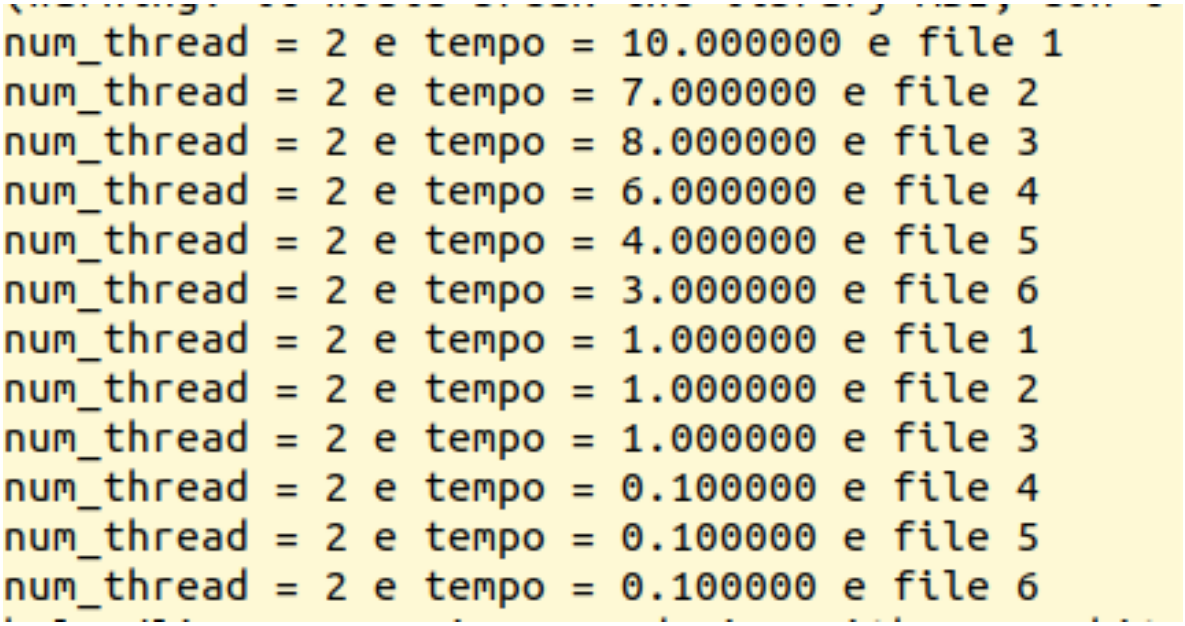

Figure 6: Output del programma con architettura mpi

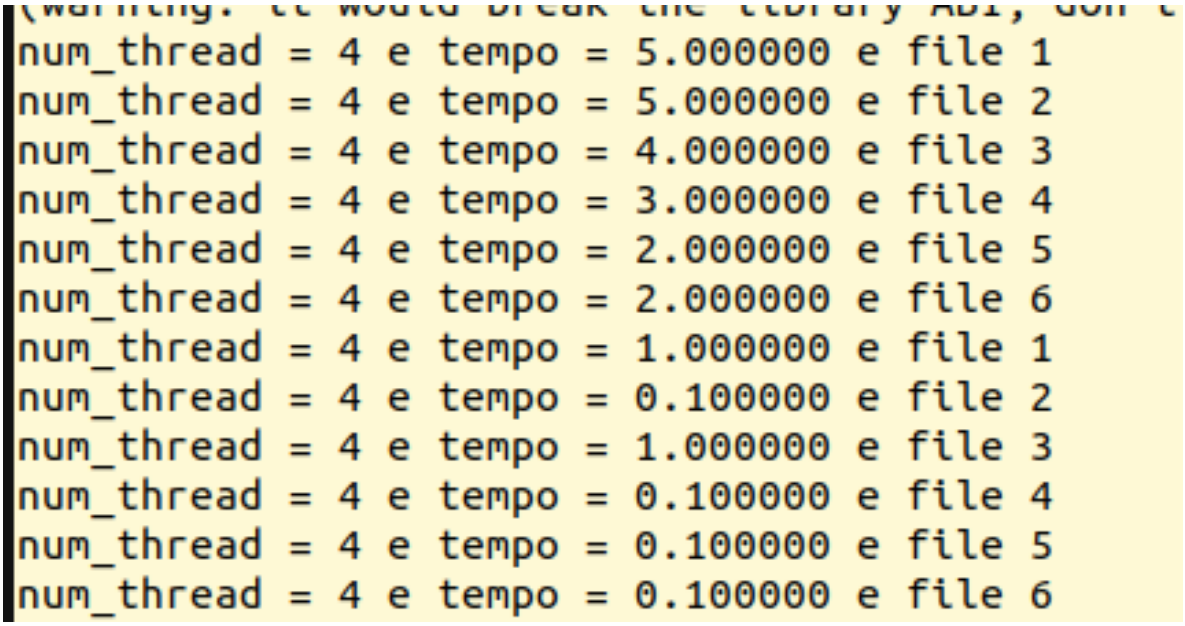

Figure 7: Output del programma con architettura mpi

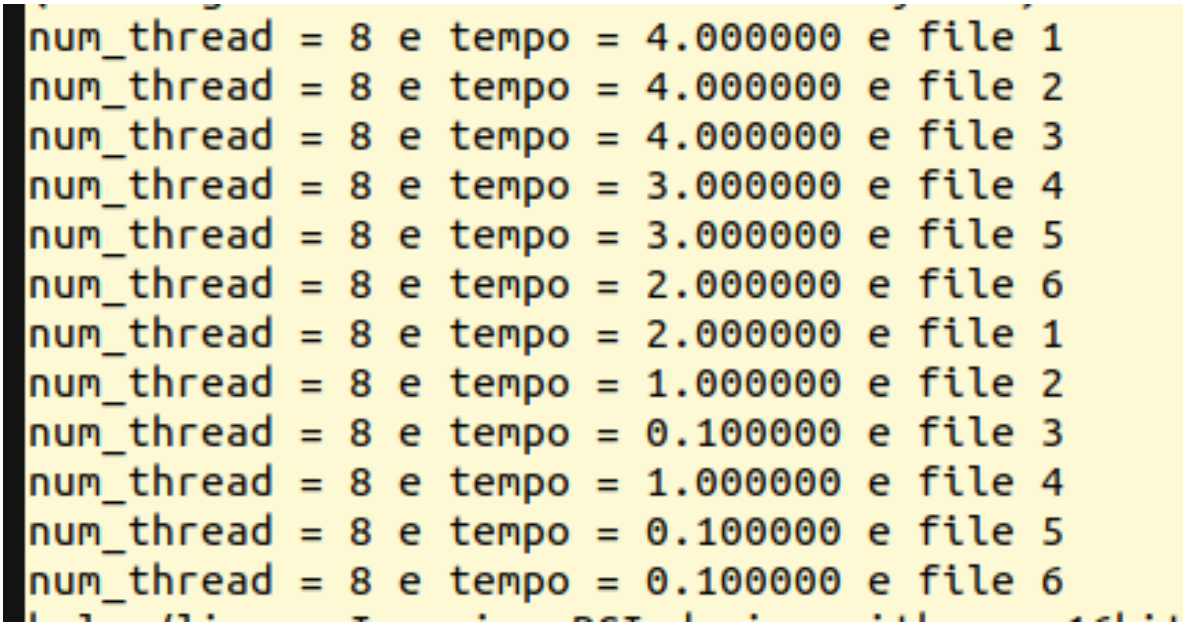

Figure 8: Output del programma con architettura mpi

```
num_thread = 16 e tempo = 10.000000 e file 1
num_thread = 16 e tempo = 8.000000 e file 2
num_thread = 16 e tempo = 8.000000 e file 3
num_thread = 16 e tempo = 6.000000 e file 4
num_thread = 16 e tempo = 5.000000 e file 5
num_thread = 16 e tempo = 4.000000 e file 6
num_thread = 16 e tempo = 2.000000 e file 1
num_thread = 16 e tempo = 1.000000 e file 2
num_thread = 16 e tempo = 1.000000 e file 3
num_thread = 16 e tempo = 1.000000 e file 4
num_thread = 16 e tempo = 0.100000 e file 5
num_thread = 16 e tempo = 0.100000 e file 6
```

Figure 9: Output del programma con architettura mpi

```
num_thread = 32 e tempo = 15.000000 e file 1
num_thread = 32 e tempo = 14.000000 e file 2
num_thread = 32 e tempo = 12.000000 e file 3
num_thread = 32 e tempo = 10.000000 e file 4
num_thread = 32 e tempo = 8.000000 e file 5
num_thread = 32 e tempo = 6.000000 e file 6
num_thread = 32 e tempo = 3.000000 e file 1
num_thread = 32 e tempo = 2.000000 e file 2
num_thread = 32 e tempo = 2.000000 e file 3
num_thread = 32 e tempo = 1.000000 e file 4
num_thread = 32 e tempo = 1.000000 e file 5
num_thread = 32 e tempo = 0.100000 e file 6
Analisi correttezza tra result_parallelo_1.txt e result_parallelo_2.txt:ok
Analisi correttezza tra result_parallelo_1.txt e result_parallelo_4.txt:ok
Analisi correttezza tra result_parallelo_1.txt e result_parallelo_8.txt:ok
Analisi correttezza tra result_parallelo_1.txt e result_parallelo_16.txt:ok
Analisi correttezza tra result_parallelo_1.txt e result_parallelo_32.txt:ok
```

Figure 10: Output del programma con architettura mpi

Tabella speedup variazione dimensione file di testo						
num_thread/size	386040	334010	299570	253890	180550	118650
1	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000
2	1.900000	2.428571	1.625000	1.500000	2.000000	1.666667
4	3.800000	3.400000	3.250000	3.000000	4.000000	2.500000
8	4.750000	4.250000	3.250000	3.000000	2.666667	2.500000
16	1.900000	2.125000	1.625000	1.500000	1.600000	1.250000
32	1.266667	1.214286	1.083333	0.900000	1.000000	0.833333

Figure 11: Tabella speedup con mpi

Tabella efficiency variazione dimensione file di testo						
num_thread/size	386040	334010	299570	253890	180550	118650
1	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000
2	0.950000	1.214286	0.812500	0.750000	1.000000	0.833333
4	0.950000	0.850000	0.812500	0.750000	1.000000	0.625000
8	0.593750	0.531250	0.406250	0.375000	0.333333	0.312500
16	0.118750	0.132812	0.101562	0.093750	0.100000	0.078125
32	0.039583	0.037946	0.033854	0.028125	0.031250	0.026042

Figure 12: Tabella efficiency con mpi