Università degli Studi di Padova

Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica

Relazione di CALCOLO PARALLELO

Algoritmo Knuth-Morris-Pratt

Autori

Raffaele Di Nardo Di Maio 1204879

Cristina Fabris 1205722

Anno Accademico 2018-2019

Indice

1

Descrizione del Problema

L'algoritmo analizzato si occupa di effettuare un pattern matching. Il problema può essere formalizzato nel seguente modo: definito un insieme Σ di simboli, detto *alfabeto*, si vuole comprendere se una sequenza di elementi dell'alfabeto

$$\{x_0\, x_1\, ...\, x_{n-1} \;\mid\; x_i \in \Sigma \;\; \forall\, 0 \leq i \leq n-1\}$$

detta *pattern*, sia contenuta o meno all'interno di una sequenza di dimensioni maggiori.
Nel nostro caso l'algoritmo riceve in input una stringa alfanumerica e un testo nel quale effettuare la ricerca e restituisce l'indice del carattere all'inizio della prima occorrenza del pattern nel testo.
Nel caso in cui non dovesse esserci alcuna occorrenza del pattern nel testo, comunica tale informazione come risultato dell'algoritmo.
Nel corso degli anni sono stati studiati ed implementati diversi algoritmi che si occupano di risolvere tale problema poiché quest'ultimo si presenta in campi di ricerca di vario genere. Per citare un esempio, nella bioinformatica l'algoritmo viene utilizzato nella ricerca di una determinata sequenza di basi all'interno del DNA.
La soluzione sequenziale "naive" al problema scandisce l'intero testo, svolgendo numerose iterazioni e ripetendo confronti già effettuati. Ad ogni iterazione vengono confrontati in maniera sequenziale gli elementi del testo e quelli della stringa per vedere se è presente un'occorrenza. Al primo confronto che restituisce esito negativo si passa all'iterazione successiva in cui si analizza nuovamente il pattern dal suo inizio.
Quest'algoritmo è poco efficiente, in quanto non tiene conto, in nessun modo, dei risultati ottenuti nelle iterazioni precedenti.
Un algoritmo più complesso ed efficiente è, invece, quello di Knuth Morris Pratt (KMP) (Sezione 2.1). Questo è stato sviluppato da Knuth e Pratt e indipendentemente da Morris nel 1975. In questo caso, l'algoritmo ha prestazioni migliori in quanto riduce il numero di confronti necessari alla ricerca. Si fa uso di una tabella che viene calcolata preliminarmente ed in essa vengono salvate informazioni riguardanti la posizione, all'interno del pattern, da cui ricominciare il confronto nell'iterazione successiva ad un confronto con esito negativo.
In questo modo, non tutti gli elementi del pattern precedentemente trovati nel testo devono essere riesaminati. Sfruttando tutte le informazioni acquisite, si riesce a diminuire il numero di confronti necessari nell'approccio classico.

L'algoritmo sviluppato all'interno della trattazione è una particolare implementazione parallela dell'algoritmo KMP che permette di lavorare in contemporanea su più sezioni del testo (Sezione 2.2).
Il protocollo di progammazione multiprocessore che è stato utilizzato da questa variante dell'algoritmo di Knuth, Morris e Pratt, è basato sullo scambio di messaggi e viene chiamato Message Passing Interface (MPI).
In seguito verranno analizzate le prestazioni e le misurazioni temporali effettuate al variare del numero di processori.

2

Descrizione dell'algoritmo

Il procedimento di pattern-matching sequenziale "naive" non sfrutta la possibile presenza di una serie di confronti con esito positivo (match), verificatasi prima di un esito negativo (mismatch) [1].
Una conseguenza di tale approccio è la dipendenza del tempo di computazione sia dalla grandezza del testo(n) che da quella del pattern alfanumerico (m), rendendo nel caso peggiore le prestazioni quadratiche rispetto ad n. Il numero di confronti effettuati, infatti, corrisponde a $\Theta((n-m+1)m)$.
L'algoritmo utilizzato divide il testo in sezioni e ne assegna una ad ogni processore disponibile. Ognuno di questi, in parallelo rispetto agli altri, esegue una ricerca del pattern voluto con l'algoritmo KMP e restituisce l'indice della prima occorrenza trovata. Nel caso in cui il pattern sia presente all'interno del testo, verrà restituito l'indice minore tra quelli individuati dai processi; altrimenti ne verrà notificata l'assenza all'utente.

2.1 Algoritmo Knuth Morris Pratt

L'algoritmo KMP esegue pattern-matching in tempo lineare rispetto alla sola dimensione del testo. Sfrutta i risultati ottenuti in precedenza per evitare di dover effettuare nuovamente confronti su caratteri già analizzati. Per rispettare queste prestazioni viene eseguita inizialmente una fase di pre-processamento del pattern, utilizzata per evitare di dover compiere maggiori confronti come nel procedimento "naive"[1].
Per comprendere in che modo venga effettuata questa fase, è conveniente introdurre i concetti di prefisso e suffisso di una stringa. Con il termine *prefisso* di lunghezza k di una stringa si intendono i primi k caratteri che la compongono, mentre con *suffisso* di lunghezza k, gli ultimi k. Per ogni prefisso del pattern, viene ricercato il suo prefisso di lunghezza maggiore, che ne costituisca anche un suffisso. Il pre-processamento restituisce un vettore di numeri interi della stessa lunghezza del pattern, chiamato vettore di fallimento o fail vector. Questo contiene, in ogni cella di indice j, la lunghezza del prefisso ricercato, come specificato in precedenza, nella sottostringa del pattern (che si estende dal primo carattere al j-esimo) (Figura 2.1). In questo modo, accedendo ai valori salvati nel vettore, nel caso in cui si verifichi un mismatch, si potrà sapere quali

[1] esito positivo = uguaglianza di un carattere nel testo e un carattere nel pattern

caratteri confrontare successivamente all'interno del testo e del pattern. Una

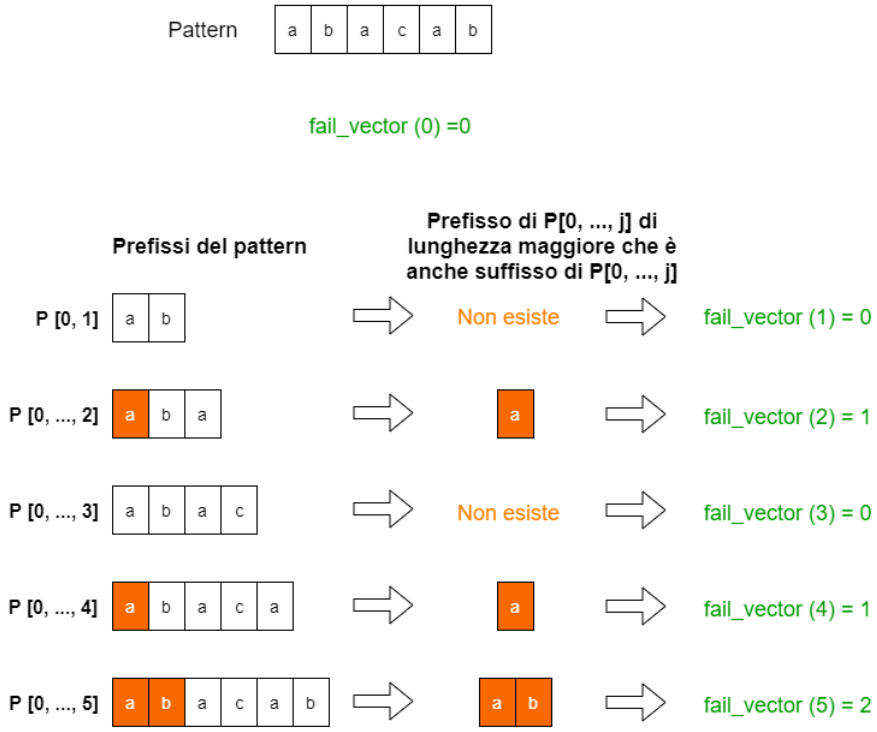

Figura 2.1: Esempio di computazione del fail vector per il pattern "abacab".

volta terminata la computazione del vettore, l'algoritmo inizia a confrontare gli elementi del testo con quelli che formano il pattern cercato. Nel caso in cui si verifichi un mismatch tra l'elemento i-esimo del testo e il j-esimo del pattern, il successivo confronto avviene tra l'elemento t-esimo del testo e l'elemento k-esimo del pattern. Gli indici t e k vengono calcolati con la seguente regola:

$$\begin{cases} k = fail\_vector[j-1] & t = i & se\ j \neq 0 \\ k = 0 & t = i+1 & se\ j = 0 \end{cases}$$

Una volta che i confronti hanno esito positivo per ogni carattere del pattern viene restituito l'indice iniziale dell'occorrenza del pattern nel testo.

2.2 KMP parallelo

Sfruttando più processori è possibile effettuare la ricerca del pattern in più porzioni del testo contemporaneamente [2]. In fase di esecuzione, il programma riceve in input da linea di comando il nome del file, contenente il testo, e il pattern da cercare.

Per prima cosa viene selezionato un processore che funga da "master" e si occupi di elaborare tutte le informazioni necessarie per l'esecuzione dell'algoritmo KMP. Nel nostro caso questo processore "privilegiato" è quello di rank 0 e i suoi compiti principali sono:

1. **Pre-processamento del pattern**
   Calcola il fail vector applicando al pattern di partenza la funzione di fallimento

2. **Elaborazione del testo**
   Legge in input il nome del file in cui cercare il pattern e si occupa di aprire lo stream verso di esso. Legge il contenuto del testo ed in seguito chiude lo stream verso di esso. Elabora le stringhe necessarie agli altri processori per applicare il KMP attraverso due fasi (Figura 2.2):

   - **Ciclo 1**
     Una volta salvato l'intero testo all'interno di una stringa, rimuove $m-1$ elementi da essa (m = lunghezza del pattern) e la divide in un numero di sezioni pari al numero di processi disponibili.
     Se la stringa risultante dalla rimozione degli ultimi caratteri non dovesse essere un multiplo del numero di processori utilizzati, viene svolta una suddivisione intera della stringa in sottostringhe (una per ciascun processore) e gli elementi non inclusi vengono inviati al processore di $rank = numero\_processori - 1$. La cosa più importante è che le stringhe debbano essere inviate rispettando l'ordine crescente dei rank e associandole in questo modo ai vari processori.
   - **Ciclo 2**
     Nel caso in cui siano presenti occorrenze del pattern a cavallo tra due sezioni del testo, calcolate nel **Ciclo 1**, non verranno rilevate. Per risolvere tale problema, il processore 0, elabora anche le sottostringhe necessarie a ciascun processore per verificare la presenza di possibili occorrenze in quelle zone.
     Le sequenze elaborate in questa fase sono costituite dagli ultimi $m-1$ caratteri di una sezione del **Ciclo 1** e gli $m-1$ caratteri della sezione successiva. Da questa elaborazione del testo, deriva la nostra scelta di escludere alcuni caratteri nel primo ciclo.

3. **Distribuzione dei dati**
   Distribuisce a ogni processore le sottostringhe elaborate nel **Ciclo 1** e nel **Ciclo 2** ed il fail vector calcolato nella prima fase. Inoltre invia ad ogni processore il valore intero necessario per stabilire quale sia l'indice di un'eventuale occorrenza nelle stringhe del **Ciclo 2**, all'interno del testo originario.

4. **Gestione dell'output**
   Verifica quale sia l'indice minore tra quelli restituiti da ogni processore e lo stampa su standard output. Nel caso in cui non sia presente il pattern, notifica ciò all'utente.

Tutti i processi, compreso il "master", in seguito alla distribuzione dei dati da parte del processore di rank 0, applicano l'algoritmo KMP sulle stringhe del **Ciclo 1**. Nel caso in cui nessuno di questi trovasse un'occorrenza, verrebbe

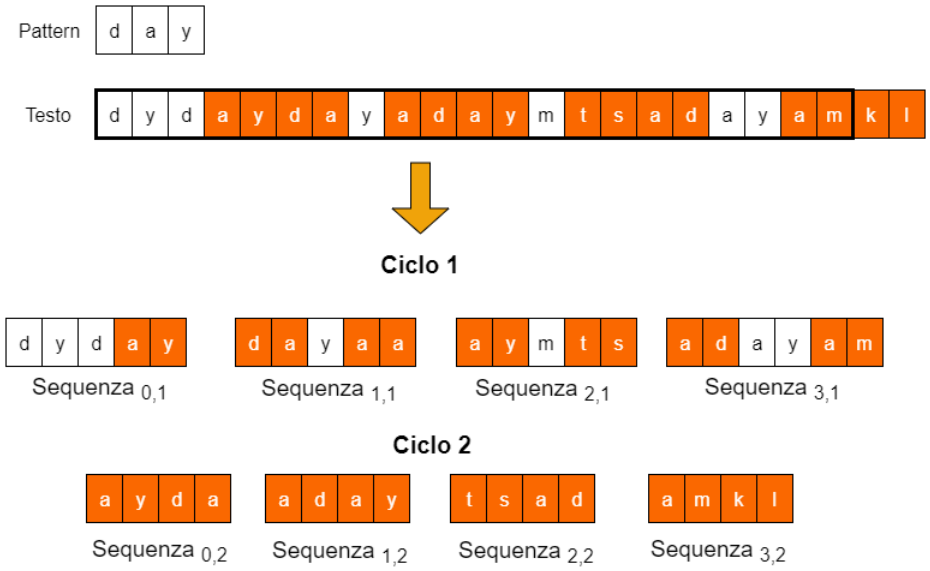

Figura 2.2: Esempio di computazione delle sequenze dei Cicli 1 e 2, utilizzando 4 processori.

eseguita la stessa ricerca sulle stringhe del **Ciclo 2**. I risultati vengono poi inviati verso il "master" che effettuerà la gestione dell'output.

3

Analisi sperimentale

In questa sezione, viene analizzata l'efficienza del programma in termini di tempi di esecuzione.
Il programma descritto nella **Sezione 2.2** è stato eseguito su una macchina in locale utilizzando OpenMPI, senza alcun problema in termini di allocazione di memoria, fino a file di input di 2GB.
Nel corso dei test, invece sul Power7 del dipartimento, si sono verificati problemi con allocazioni di memoria superiori complessivamente a circa 120 MB. Per questo motivo, l'algoritmo è stato modificato per poter leggere blocchi di 50 MB alla volta da un file di dimensione qualsiasi.
Ciò garantisce che l'allocazione complessiva di memoria sia inferiore a 120MB di memoria ma causa numerosi ritardi. Questi ultimi sono dovuti alla lettura del testo in blocchi e alle comunicazioni che devono essere effettuate per trasferire informazioni tra processori.
Il codice è stato prima di tutto ottimizzato mediante il miglioramento della gestione dei cicli, della dichiarazione delle variabili e del loro utilizzo. L'analisi dei tempi di esecuzione è stata svolta su file di testo ASCII e analizzando i tempi di esecuzione totale e il relativo speedup, i tempi di comunicazione, MIPS (Million Instructions Per Seconds), tra i vari processori e la variazione dei tempi di esecuzioni in base alla taglia dell'input.

3.1 Tempi totali d'esecuzione e relativo Speed Up

La prima analisi riguarda i tempi di esecuzione del programma al variare del numero di processori su un testo selezionato di grandezza 140MB per garantire l'elaborazione di più blocchi da 50MB di testo (**Figura 3.1**). Il testo selezionato presenta un'unica occorrenza in corrispondenza dell'indice 146800601. Questo permette di analizzare il tempo di esecuzione nel caso quasi peggiore, in quanto quello massimo si avrebbe se il testo fosse composto da caratteri completamente casuali e non contenesse quindi alcuna occorrenza del pattern. Il tempo di esecuzione riguarda il lavoro utile ed è quindi il tempo complessivo impiegato dall'algoritmo senza considerare la lettura da file.
Nel selezionare quale fosse l'ottimizzazione da utilizzare in fase di compilazione, è stato scelto O3. In **Figura 3.2**, è stato riportato lo speedup ricavato dai tempi ottenuti precedentemente.

	O3
1	0.508896
4	0.432822
8	0.308186
16	0.288454
32	1.217619

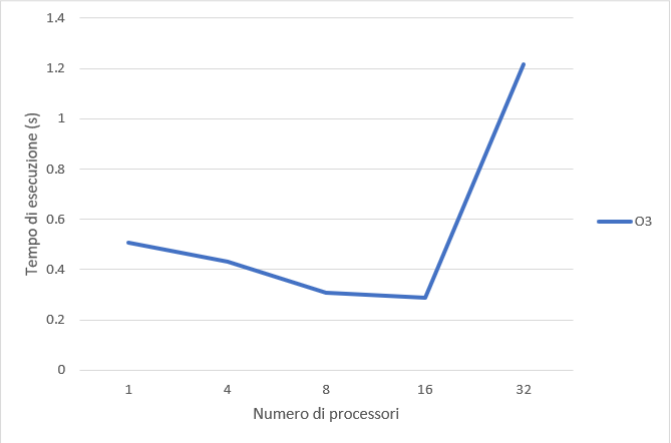

Figura 3.1: Tempi di esecuzione al variare del numero di processori.

Dalla **Figura 3.1** e **3.2** si evince una significativa diminuzione dei tempi di esecuzione, all'aumentare del numero di processori. L'utilizzo di 32 processori mette in evidenza un peggioramento significativo del tempo di esecuzione anche se, con questa elaborazione, i tempi di acquisizione dell'input sono diminuiti.
La seconda analisi è stata svolta utilizzando sempre 32 processori ma variando la grandezza dell'input da 1KB fino a 1GB (**Figura 3.3**). Anche in questo caso sono stati valutati i tempi di lavoro utile.

Si nota come all'aumentare della grandezza dell'input, aumentino anche i tempi di esecuzione, anche se l'ottimizzazione O3 garantisce migliori prestazioni rispetto alle altre con file di grandezza inferiore o uguale a 512MB. Non ottimizzando in compilazione il codice, si ha però un'anomalia sulla computazione di un file di 512MB, che risulta più lunga rispetto alla sua elaborazione sulla taglia massima analizzata, che invece è la migliore di tutti i test su questa grandezza del file.

3.2 Comunicazioni

In questa fase sono stati analizzate le percentuali di comunicazione all'interno dei vari test, riportati nella **Sezione 3.1**. In **Figura 3.4**, vengono riportate le percentuali di comunicazioni all'intero di una esecuzione al variare della taglia dell'input.

Si può notare come, aumentando la taglia dell'input, la percentuale di comunicazioni è sempre maggiore e ciò causa un rallentamento dovuto alle porzioni di testo che ogni processore deve ricevere dal master per analizzarlo.
In **Figura 3.5**, vengono invece riportate le percentuali di comunicazioni all'in-

	O3
4	1.175763
8	1.651263
16	1.764219
32	0.417944

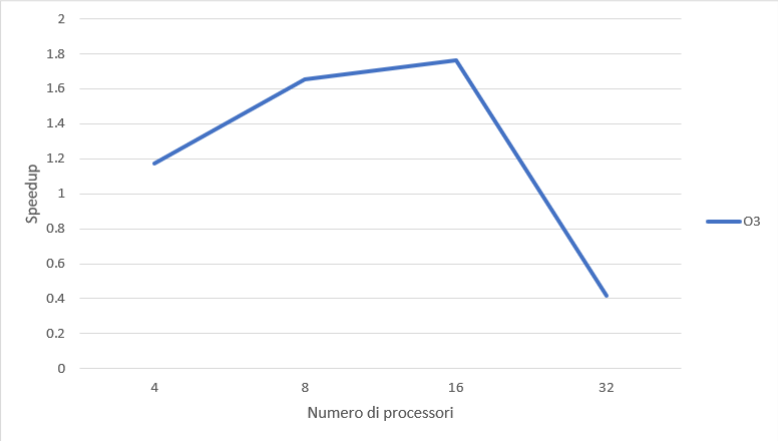

Figura 3.2: Speedup.

terno di una esecuzione al variare del numero di processori utilizzati e con taglia del file di input di 140MB.

3.3 Istruzioni al secondo

L'ultima analisi è stata incentrata sulla frequenza di istruzioni svolte dall'algoritmo durante l'intero tempo di esecuzione (compresa la lettura da file), ponendo l'attenzione sul processore che svolge l'attività di Master (Rank 0). Nella **Figura 3.6** è stato valutato il numero di istruzioni al secondo al variare della taglia dell'input. Nella **Figura 3.7** è stato analizzato invece il numero di istruzioni al secondo all'aumentare del numero di processori, sulla stessa taglia di 140MB.

Il numero di istruzioni eseguite al secondo decresce, come ci si aspetta, all'aumentare del numero di processori e rispettivamente della grandezza dei file. Questo è dovuto all'incremento del numero di comunicazione tra i vari processori. Ci sono principalmente due anomalie in questa progressione: nell'elaborazione di un testo di 1GB con ottimizzazione O2 e nell'elaborazione di 140MB con 32 processori.

	Normale	O2	O3
1KB	0.92275	0.839669	1.374216
1MB	1.231344	3.474741	1.179271
512MB	20.92448	18.97407	5.772525
1GB	16.98623	21.49396	26.0197

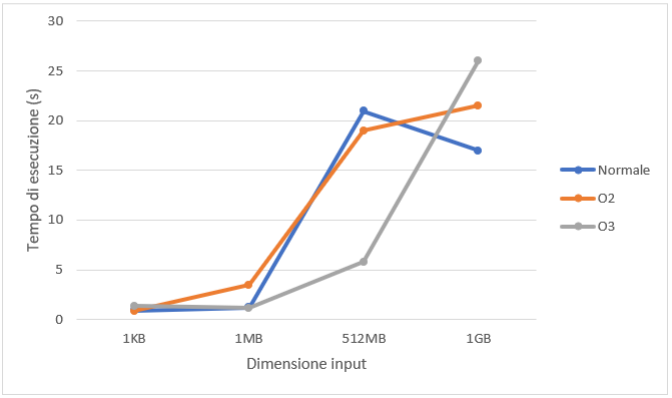

Figura 3.3: Tempi di esecuzione al variare della taglia dell'input

	Normale	O2	O3
1KB	99%	99%	99%
1MB	99%	99%	99%
500MB	98%	97%	97%
1GB	91%	98%	98%

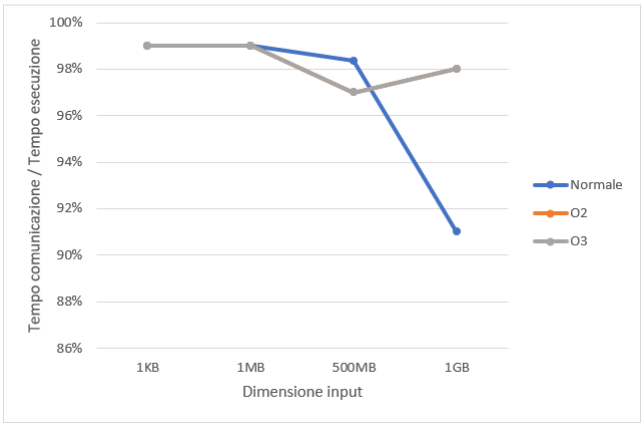

Figura 3.4: Percentuale di comunicazioni al variare della taglia dell'input

	O3
1	11%
4	57%
8	75%
16	83%
32	98%

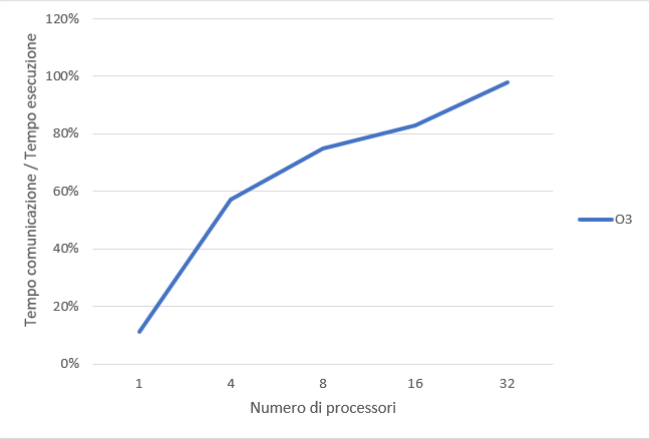

Figura 3.5: Percentuale di comunicazioni al variare del numero di processori

	Normale	O2	O3
1KB	367.752	1015.592	817.996
1MB	217.063	333.533	152.405
512MB	34.795	16.329	24.391
1GB	19.79	18.734	20.764

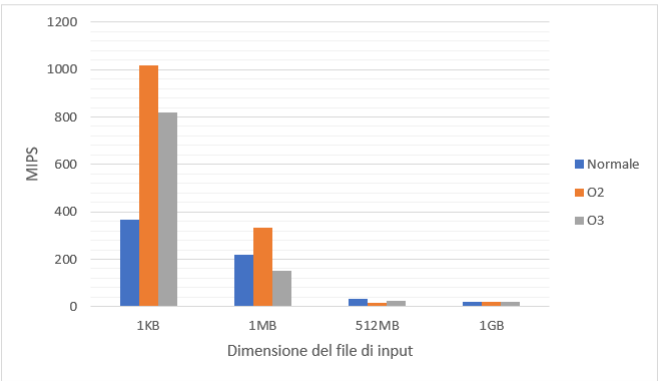

Figura 3.6: Numero di operazioni al secondo al variare della taglia dell'input

	O3
1	52.992
4	11.006
8	10.84
16	10.291
32	32.8

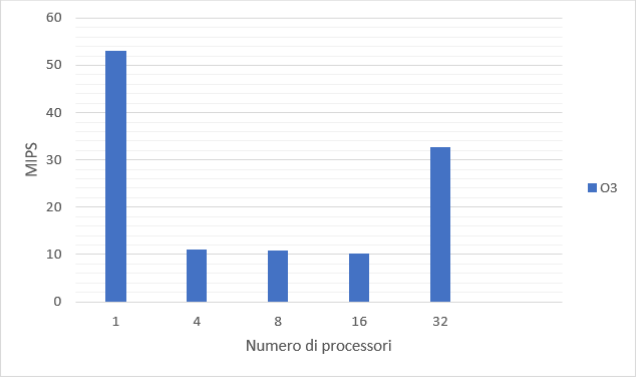

Figura 3.7: Numero di operazioni al secondo al variare del numero di processori

4

Conclusioni

L'algoritmo è molto efficiente su particolari taglie dell'input ma presenta numerosi ritardi temporali dovuti alle molte comunicazioni e alla lettura in blocchi del file in input.
Gran parte del tempo di esecuzione viene occupato interamente dalla lettura del file, ciò riduce di molto l'efficienza e soprattutto la possibilità di sfruttare a pieno la potenza di tutti i processori.
Ciò rende completamente inefficiente l'algoritmo in alcuni casi, su file di dimensione moderate, rendendolo peggiore della sua implementazione sequenziale.

Bibliografia

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein *Introduction to Algorithms, Second Edition, pp. 923-930, 2001*

[2] Akhtar Rasool, Nilay Khare *Parallelization of KMP String Matching Algorithm on Different SIMD architectures: Multi-Core and GPGPU's, International Journal of Computer Applications, Volume 49, pp. 26-28, 2012*