Università di Modena e Reggio Emilia
Facoltà di Ingegneria Enzo Ferrari di Modena

Studio sulla parallelizzazione dell'algoritmo di Huffman e implementazione in C++ mediante Threading Building Blocks

Progetto per Laboratorio di Ingegneria Informatica

Referente: Prof. Costantino Grana

Project Leader: Andrea Palazzi
Guido Borghi
Andrea Corbelli

Anno Accademico 2014/2015

Sintesi

La presente relazione descrive il lavoro di analisi e di implementazione dell'algoritmo di compressione di Huffman parallelo, mediante le librerie della Intel denominate Threading Building Blocks (TBB), svolto nell'ambito dell'esame di Laboratorio di Ingegneria Informatica.
Vengono mostrati le analisi teoriche, le soluzioni implementate e i valori di *speedup* ottenuti rispetto a un implementazione dell'algoritmo di Huffman completamente sequenziale.

Il documento può essere suddiviso in alcune parti principali:

- Il primo capitolo descrive a livello teorico la celebre tecnica di compressione a codici di lunghezza variabile, ideata da Huffman nel 1952; in particolare vengono analizzate la fase di compressione, la relativa fase di decompressione e considerazioni riguardanti la varianza dei codici prodotti e dei cosiddetti codici canonici.

- Il secondo capitolo descrive le librerie utilizzate per parallelizzare l'algoritmo di compressione, andandone ad analizzare i costrutti messi a disposizione degli sviluppatori e altri dettagli quali la cronologia delle versioni (per questo progetto si è utilizzata l'ultima disponibile) e i sistemi operativi supportati.

- Nel terzo capitolo si riportano le scelte fatte in fase di implementazione, come nel caso di gestione della memoria, e si mostrano frammenti di codice con i costrutti delle TBB.

- Infine nei capitoli finali (4, 5) si riportano i risultati ottenuti tramite grafici e si fanno considerazioni a riguardo: si cerca di stabilire che risultati sono stati raggiunti grazie all'implementazione parallela dell'algoritmo di Huffman.

Indice

Capitolo 1

La codifica di Huffman

1.1 Premessa

La codifica di Huffman è un noto metodo di compressione di dati che utilizza codici di lunghezza variabile.

Dato un insieme di simboli (alfabeto) e l'informazione della frequenza con cui occorrono (la loro probabilità), tale metodo costruisce un insieme di parole di codice con la minor lunghezza media possibile e le assegna ai simboli in input; Huffman stesso dimostrò che nessun altro metodo di codifica può produrre una lunghezza media del codice minore.

Dal momento in cui fu proposta da Huffman nel 1952 tale metodologia ha incontrato grande successo ed è tuttora utilizzata in numerose applicazioni, sia singolarmente sia come fase di un più complesso processo di compressione *multi-step*.

1.2 Codifica

1.2.1 Algoritmo di compressione

L'algoritmo di compressione di Huffman comincia con la costruzione di una lista di tutti i simboli dell'alfabeto, ordinata in ordine decrescente di probabilità.

Una volta creata la lista, viene poi costruito (procedendo in modo *bottom-up*) un albero binario con un simbolo in ogni foglia. Questo procedimento avviene in più passi, dove ad ogni passo i due simboli con la minore probabilità

sono selezionati, aggiunti alla cima dell'albero temporaneo, quindi cancellati dalla lista e rimpiazzati con un simbolo ausiliario rappresentante entrambi i simboli originali.
Una volta che la lista si riduce ad un unico simbolo ausiliario rappresentante l'intero alfabeto, l'albero è completo. L'albero è infine percorso dalla radice alle foglie per determinare le parole del codice.

Le parole di codice così ottenute posseggono una proprietà assai desiderabile in un contesto di compressione dati, di seguito esposta: nessuna *codeword* costituisce il prefisso di una *codeword* di lunghezza maggiore, ovvero abbiamo ottenuto una codifica *prefix-free*.
Il vantaggio di una codifica *prefix-free* è che le parole di codice possono essere scritte nel file compresso una di seguito all'altra senza che siano intervallate da separatori.
Nonostante il notevole risparmio di spazio, in fase di decompressione il decodificatore potrà comunque eseguire il parsing del file senza che sussistano ambiguità, dal momento che nessuna *codeword* è contenuta in un'altra di lunghezza maggiore.

Per maggiore chiarezza, si veda il seguente esempio:

Simbolo	Codice
a	00
b	01
c	10
d	11

Simbolo	Codice
a	0
b	1
c	01
d	11

Tabella 1.1: Confronto tra una codifica *prefix-free* (a sinistra) ed una che non lo è (tabella di destra).

Si può notare facilmente che utilizzando la seconda tabella di codici ci possono essere situazioni di ambiguità: ad esempio la stringa 0001 può essere interpretata dal decodificatore in modo univoco utilizzando la tabella di sinistra, in numerosi modi utilizzando la tabella di destra.

1.2.2 Varianza del codice

Come si può dedurre dal funzionamento dell'algoritmo di compressione, il codice di Huffman generato non è l'unico possibile. Ciò è dovuto al fatto che nel caso in cui ci siano più di due simboli con probabilità minima, questi possono essere accoppiati in modo arbitrario, dando origine a differenti configurazioni dell'albero dei codici.

Dal momento che dato in input un insieme di simboli è possibile generare più codici di Huffman, è naturale chiedersi quale tra questi sia da preferire. La risposta è semplice: quello con la varianza minore.

Per ottenere varianza minore è necessario, nel momento in cui si vanno a ordinare le probabilità dei simboli, mettere sempre in prima posizione, in caso di parità, la probabilità dei codici combinati.

A questo proposito è necessaria una precisazione, poiché si è detto che l'algoritmo di Huffman produce in ogni caso un codice di lunghezza media minima possibile, ragion per cui può non essere immediato capire perché un codice a minor varianza sia desiderabile.
In effetti, se il codificatore si limita a scrivere i dati compressi su un file, non fa alcuna differenza. Tuttavia, nel caso in cui i dati compressi siano trasmessi *on-the-fly* su una rete, un'alta varianza del codice causa la continua variazione della *bit-rate* del tramsettitore, richiedendo quindi l'utilizzo di un buffer, che dovrà essere di dimensioni tanto maggiori quanto più grande è la varianza del codice.

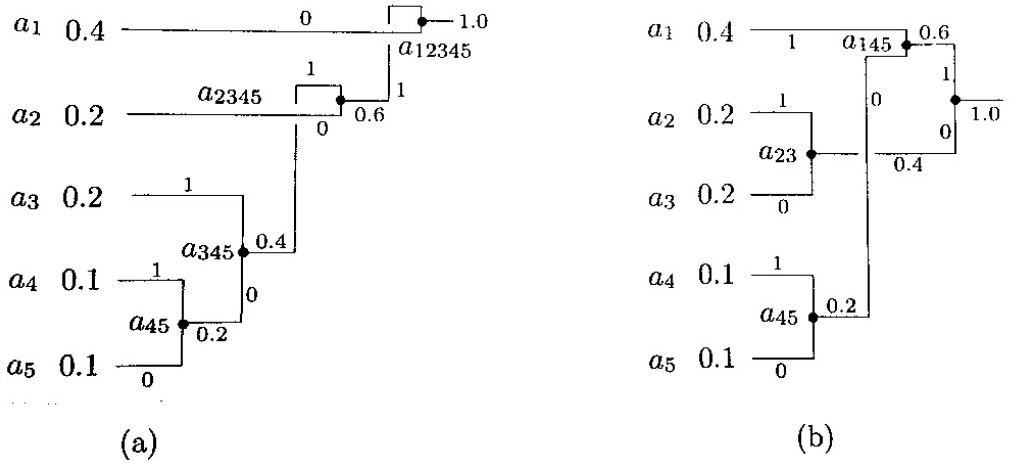

Fig. 1.1: Due codici di diversa varianza, dato lo stesso alfabeto

1.2.3 Codici canonici

Nella sezione precedente si è visto che, dato il medesimo alfabeto, l'applicazione dell'algoritmo di Huffman può portare alla creazione di più alberi

distinti: sono state inoltre esposte le ragioni per cui in numerosi contesti è preferibile scegliere l'albero maggiormente bilanciato, ovvero quello a varianza minore.
E' stato proposto un metodo per costruire codici di Huffman canonici, ovvero codici costruiti in maniera particolare tale da renderne possibile la creazione avendo a disposizione solamente la loro lunghezza; questo permette di poter risparmiare in fase di compressione lo spazio altrimenti occupato dai codici (necessari da trasmettere in quanto il decodificatore non saprebbe ricostruirli).
Sostanzialmente il metodo prevede di ordinare i codici per lunghezza, dalla minore alla maggiore; si procede ad assegnare al primo codice tanti bit posti a 0 quanti ne indica la sua lunghezza: se il codice successivo ha la medesima lunghezza si aggiunge 1, altrimenti si aggiunge 1 e si moltiplica per due il suo valore (e questo consiste in uno shift verso sinistra di una posizione).

1.3 Decodifica

In fase di decodifica si ha un algoritmo sostanzialmente semplice. Ci si pone nel caso in cui in fase di codifica siano stati utilizzati i codici canonici di Huffman calcolati mediante una stima statica della sequenza.
Viene preso in input il file compresso; nel suo header deve essere presente una tabella che contiene informazioni riguardo ai simboli presenti e alla loro lunghezza; non contiene i codici in quanto possono essere facilmente ricostruiti in maniera molto simile a quanto visto nel paragrafo precedente: pur in maniera limitata, questo permette di risparmiare spazio.
L'algoritmo prende in esame questa tabella, costruisce quindi i codici canonici, legge in maniera sequenziale il file compresso sostituendo a ogni codice trovato il rispettivo simbolo; tale operazione risulta notevolmente semplificata dal fatto che in fase di compressione è stata utilizzata, come abbiamo visto, una codifica *prefix-free*.

Capitolo 2

Threading Building Blocks (TBB)

Le Threading Building Blocks sono delle librerie sviluppate dalla Intel con l'obiettivo specifico di offrire agli sviluppatori software uno strumento con cui sfruttare appieno le caratteristiche dei processori di nuova generazione multi-core.

I processori multi-core sono caratterizzati dalla presenza di più nuclei di processori all'interno dello stesso chip e quindi di un'architettura altamente parallela, in grado di aumentare il numero di operazioni eseguibili in un unico ciclo di clock rispetto ai processori single-core.
Lo sviluppo di soluzioni hardware parallele fu dovuto sostanzialmente a un limite nello sviluppo tecnologico dei processori tradizionali, in cui alla volontà di avere un aumento della potenza di elaborazione corrispondeva principalmente un aumento della frequenza di clock, con evidenti limiti sia in relazione al calore dissipato che ai consumi energetici, troppo elevati se paragonati al modesto aumento di prestazioni.

L'introduzione sul mercato di nuovi processori multi-core ha portato alla necessità di avere software scritto appositamente per achitetture parallele, quindi ottimizzato per un utilizzo multi-thread. E' semplice allora intuire come le grandi casi produttrici di processori, come appunto l'Intel, avessero interesse nel fornire strumenti per rendere possibile la programmazione parallela.
In particolare le TBB oltre che rendere possibile lo sviluppo di software multi-thread, cercano di rendere semplice la programmazione stessa introducendo un livello di astrazione ulteriore per permettere ai programmatori di parallelizzare il codice senza preoccuparsi dei dettagli di basso livello quali, per

esempio, la sincronizzazione dei threads o la loro chiusura.

2.1 Costrutti

I componenti messi a disposizione dei programmatori possono essere così riassunti:

- **Costrutti di base**: parallel_for, parallel_reduce, parallel_scan;
- **Costrutti avanzati**: parallel_while, parallel_do, parallel_pipeline, parallel_sort;
- **Containers**: concurrent_queue, concurrent_priority_queue, concurrent_vector, concurrent_hash_map;
- **Gestione memoria**: scalable_malloc, scalable_free, scalable_realloc, scalable_calloc, scalable_allocator, cache_aligned_allocator;
- **Operazioni atomiche**: fetch_and_add, fetch_and_increment, fetch_and_decrement, compare_and_swap, fetch_and_store

Vengono inoltre messi a disposizione due strumenti, utili soprattutto in fase di sviluppo e test del software:

- **Timing**: strumenti per la rilevazione dei tempi di esecuzione
- **Task Scheduler**: strumento per l'accesso diretto alla creazione e attivazione dei task

2.2 Cronologia delle versioni

Lo sviluppo del progetto è stato fatto nel corso dell'ultima parte dell'anno 2014, si è utilizzata quindi la versione 4.3 delle Threading Building Blocks (l'ultima disponibile).
Per completezza si riporta la cronologia delle versioni:

- **1.0**: Agosto 2006
- **2.0**: Aprile 2007
- **3.0**: Maggio 2010
- **4.0**: Settembre 2011

- **4.3**: Agosto 2014

In particolare la versione 4.3 introduce nuovi costrutti quali le *tasks arenas* e la piena compatibilità con l'interfaccia standard C++11.

2.3 Sistemi Operativi supportati

Sono molti i sistemi operativi supportati dalle TBB; di seguito un elenco di alcuni di questi:

- **Windows** (XP/7 o più recente)
- **OS X** (dalla versione 10.5.8)
- Sistemi *Open Source* quali ad esempio:
  - Sun Solaris
  - Debian
  - Red Hat Fedora
  - OpenSUSE

Capitolo 3

Implementazione

3.1 Strumenti utilizzati

Gli strumenti utilizzati per lo sviluppo del progetto sono i seguenti:

- **Doxygen**: tool utilizzato per generare la documentazione relativa al codice; supporta il linguaggio C++ (utilizzato per il progetto), ma anche altri linguaggi come C, C#, PHP ecc.);
- **Visual Studio 2012**: framework per lo sviluppo del codice C++
- **GitHub**: software di sincronizzazione utilizzato per condividere in maniera sempre aggiornata i file di programmazione fra i tre membri del gruppo.

3.2 Gestione della memoria

La natura stessa del progetto richiede la possibilità di elaborare file di grandi dimensioni. Per tale motivo risulta necessario garantire un corretto utilizzo della memoria a livello di programmazione.
Per ottenere questi risultati si è deciso di dividere in pezzi (*chunks*) il file in ingresso di una determinata grandezza, ottenendo i seguenti risultati:

- Un utilizzo della memoria RAM non troppo aggressivo, per evitare improvvisi *crash* di sistema e permettere all'utente di eseguire eventualmente altri programmi durante la fase di codifica o decodifica;
- Il programma risulta essere eseguibile su una fascia molto ampia di calcolatori in quanto il requisito minimo di RAM disponibile è molto basso, ovvero di circa 100 MB.

- Si è notato come il lavorare in maniera parallela su chunk di determinata grandezza, anzichè lavorare in maniera diretta su un unico file di grandi dimensioni, favorisca le prestazioni generali e quindi i tempi di elaborazione del programma;

Nello specifico all'interno del codice la divisione in chunks avviene in due parti distinte: la prima per limitare la grandezza massima del file a livello generale, la seconda per evitare che le strutture dati interne al codice necessarie per l'elaborazione possano diventare di dimensione critica.
La prima divisione in chunks è effettuata in maniera **statica**, ovvero la dimensione delle parti è stata decisa a tempo di programmazione e non può essere modificata a tempo di esecuzione; in particolare si sono effettuati numerosi test per stabilire la dimensione migliore che privilegiasse le prestazioni.
La seconda divisione in chunks è effettuata in maniera **dinamica**, ovvero il programma legge, tramite le API del sistema operativo, la quantità di memoria disponibile ed in base a questo dato stabilisce la grandezza delle parti.

3.3 Soluzioni adottate

La parallelizzazione del codice ha riguardato tre particolari aree: quella di calcolo dell'istogramma delle occorrenze, quella di lettura dei caratteri con relativa ricerca dei codici di Huffman e la scrittura del file compresso (o meglio, la scrittura dei vari chunks compressi su un buffer temporaneo che viene poi scritto su disco, soluzione adottata in quanto evidentemente la scrittura fisica delle informazioni non può essere parallelizzata).
Altre parti del codice risultano essere poco adatte alla parallelizzazione sia per la difficoltà implementativa che per i benefici che potrebbero portare in termini di aumento di prestazioni.

Riportiamo di seguito parti del codice che implementano l'utilizzo delle librerie TBB in modo tale da rendere l'idea dei costrutti utilizzati, senza pretesa di completezza e analisi dei dettagli in quanto il codice completo e i relativi commenti sono fruibili nella all'interno della documentazione creata con Doxygen (fornita in allegato).

3.3.1 Generazione dell'istogramma

Il costrutto utilizzato è il *parallel_reduce*.
In particolare la funzione prende in input l'istogramma globale che memorizza le occorrenze di tutti i chunk in cui è diviso il file (che quindi viene passato

per riferimento), e la dimensione stessa dei chunk, in maniera tale da poter specificare al suo interno le aree di lettura di ciascun core del processore.

```
void ParHuffman::create_histo(TBBHistoReduce& tbbhr,
   uint64_t chunk_dim){
        parallel_reduce(
        blocked_range<uint8_t*>(_file_in.data(),
        _file_in.data()+chunk_dim), tbbhr);
}
```

3.3.2 Lettura e scrittura su buffer

Per ogni chunk in cui è diviso il file in input e per ogni microchunk in cui è ulteriormente diviso un singolo chunk, vengono letti in maniera parallela i simboli presenti e viene ricercato il codice di Huffman associato; viene quindi scritto un buffer temporaneo che verrà poi scritto su disco una volta terminata l'elaborazione di tutti i microchunk. I range per il *parallel_for*, il costrutto delle TBB qui utilizzato, vengono stabiliti in base alla dimensione del microchunk e dal numero dell'iterazione ad esso collegato.

```
for (size_t i=0; i < num_microchunk; ++i) {
 vector<pair<uint32_t, uint32_t>>
 buffer_map(microchunk_dim);
 parallel_for(blocked_range<int>(i*microchunk_dim,
 microchunk_dim*(i+1),10000),
 [&](const blocked_range<int>& range) {
        pair<uint32_t,uint32_t> element;
        for( int r=range.begin(); r!=range.end(); ++r ){
                element = codes_map[_file_in[r]];
                buffer_map[r-i*microchunk_dim].first =
                   element.first;
                buffer_map[r-i*microchunk_dim].second =
                   element.second;
        }

 });
 for (size_t j = 0; j < microchunk_dim; j++)
        btw.write(buffer_map[j].first,
        buffer_map[j].second);
 buffer_map.clear();
```

Capitolo 4

Risultati

Sono state testate due versioni simili dell'algoritmo di compressione: differiscono principalmente per l'utilizzo di un costrutto *map* o *vector* all'interno del codice per trovare le corrispondenze tra simbolo, codice di Huffman e sua lunghezza (cfr. Capitolo 1).
Al di là delle possibili considerazioni riguardo al loro funzionamento, si può affermare che il carico computazionale, e quindi le prestazioni, che si ottengono dai due costrutti sono molto differenti, e questo ha ricadute riguardo a test sulle perfomance dell'algoritmo.
Bisogna inoltre tenere in considerazione che la maggior parte delle operazioni richieste dall'algoritmo sono di lettura/scrittura su disco, contraddistinte quindi da tempi di esecuzioni di vari ordini di grandezza più grandi rispetto a quelle di calcolo.
Queste considerazioni si rivelano fondamentali per interpretare correttamente i risultati ottenuti: non sempre le librerie di parallelizzazione hanno apportato i benefici sperati in termini di tempo di esecuzione.
I test sono stati eseguiti su una macchina sui cui erano presenti i componenti:

- Intel Core i7 - 2670QM
- 8 GB RAM
- Hard-Disk 5400 rpm

Consideriamo l'implementazione tramite costrutto *map*.
Nella figura 4.1 vengono riportati i risultati ottenuti eseguendo la compressione su file con grandezze dell'ordine dei Megabyte. Mettendo a confronto i tempi di esecuzione per la modalità sequenziale e parallela si possono notare incrementi nelle prestazioni.

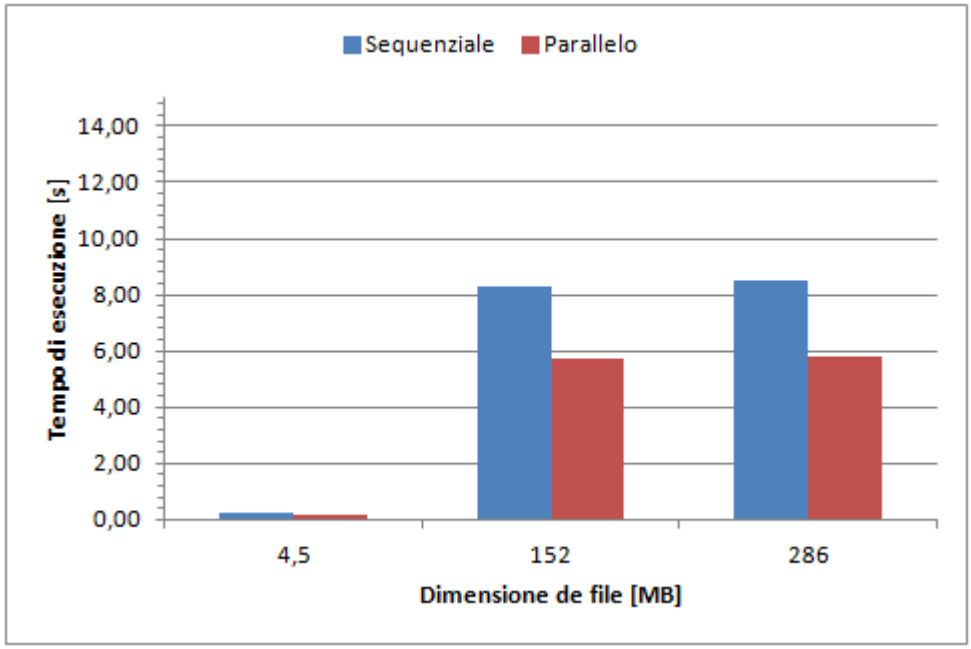

Fig. 4.1: Tempo di elaborazione per file nell'ordine dei Megabyte

Si è poi proceduto ad eseguire test su file molto più grandi, partendo da un file di circa 600 MB, arrivando a file nell'ordine dei Gigabyte: i risultati sono visibili nella figura 4.2.
Come intuibile, in questo caso il miglioramento nei tempi di esecuzione è risultato più marcato.
E' stato possibile eseguire questi test solamente dopo l'implementazione delle tecniche di gestione della memoria esposte nel paragrafo 3.2.

Vengono infine riportati i valori di *speedup* ottenuti nella figura 4.3: si può notare come i migliori risultati si abbiano con file di grandezza considerevole.

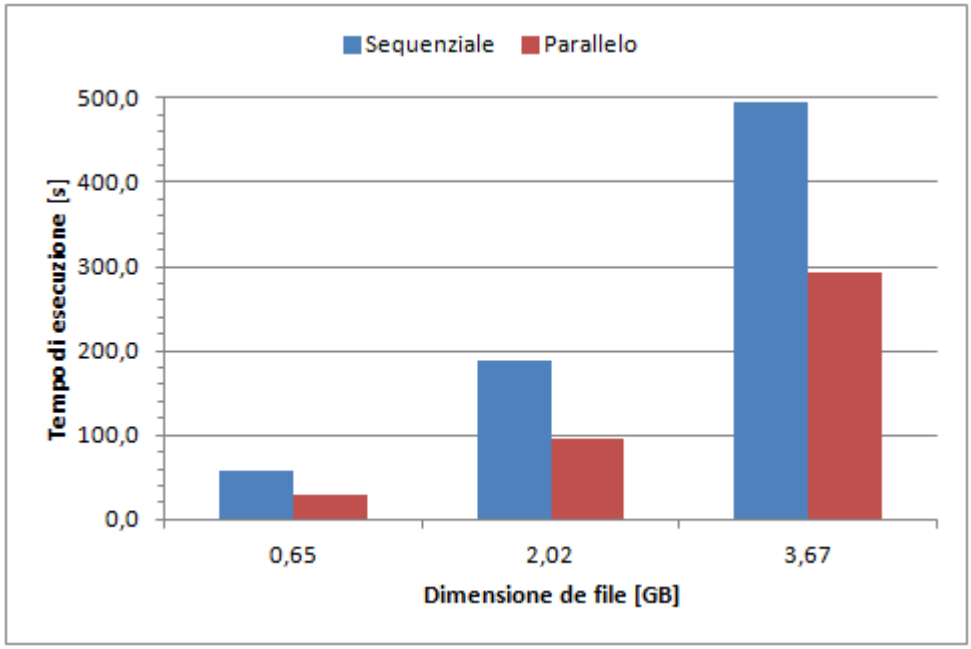

Fig. 4.2: Tempo di elaborazione per file nell'ordine dei Gigabyte

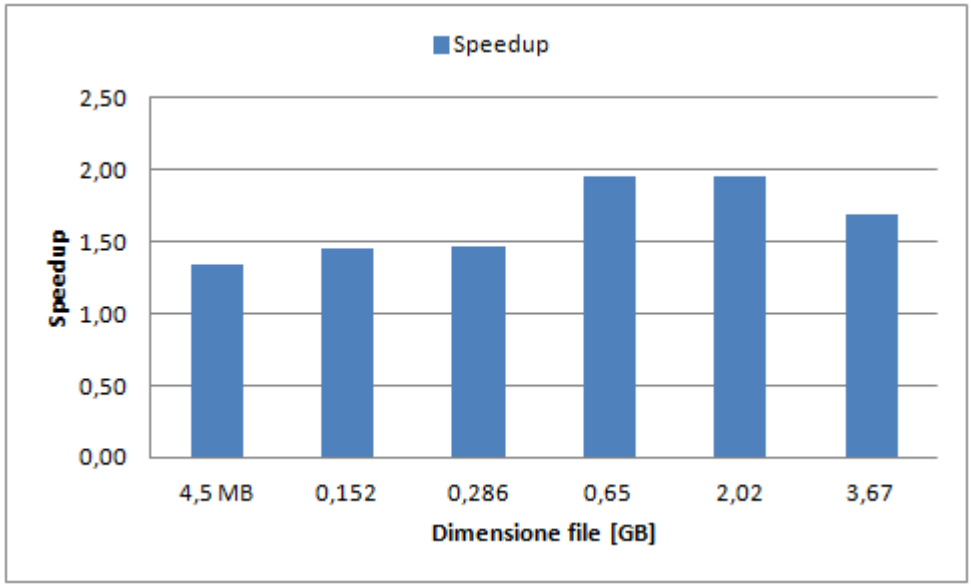

Fig. 4.3: Valori di speedup ottenuti

Consideriamo ora l'implementazione tramite costrutto *vector*.
Come nel caso precedente inizialmente si sono compressi file con ordini di grandezza del Megabyte. I valori ottenuti sono differenti da quelli di prima.

Probabilmente, andando a sottrarre complessità computazionale all'interno dell'algoritmo, adottando il costrutto vector, si ottiene un incremento netto delle prestazioni dell'algoritmo sequenziale, mentre i costrutti paralleli sembrano invece rallentare l'elaborazione.

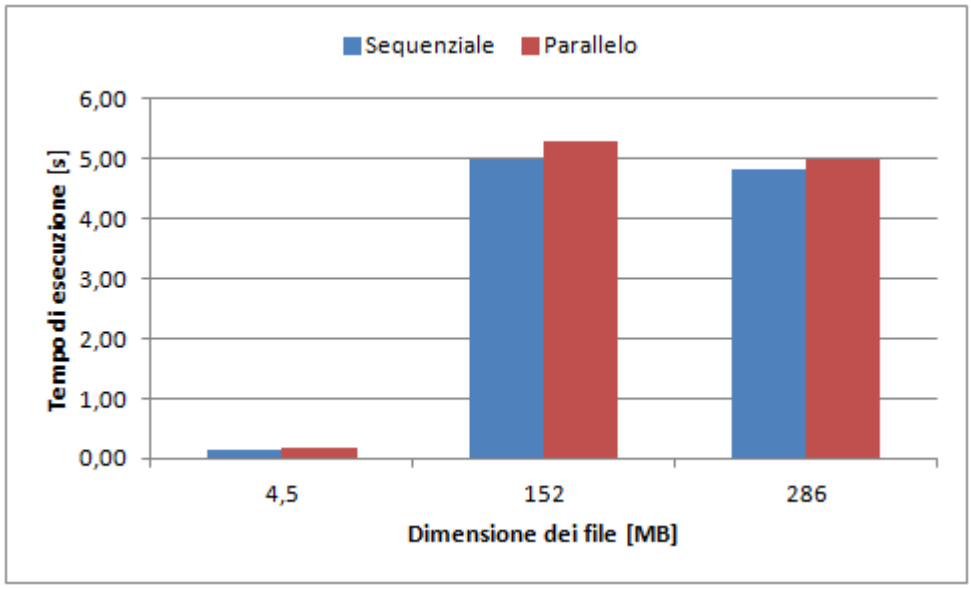

Fig. 4.4: Tempo di elaborazione per file nell'ordine dei Megabyte

Abbiamo voluto testare questo comportamento anche con file di dimensione molto maggiore, analogamente al caso precedente. Pure in questa situazione la versione parallela dell'algoritmo non sembra apportare benefici ai tempi di esecuzione; solamente nel caso del file più grande considerato si è avuto un piccolo incremento delle prestazioni, comunque di molto inferiore rispetto alla precedente implementazione.

I valori di speedup ottenuti sono visibili nella figura 4.6.

E' evidente quindi che l'elevato numero di operazioni che prevedono l'accesso in memoria da una parte e la presenza di un carico computazionale non troppo elevato dall'altra, rendono, soprattutto per file di dimensioni limitate, l'integrazione di costrutti paralleli particolarmente difficile, specialmente nell'ottica del miglioramento globale delle prestazioni.

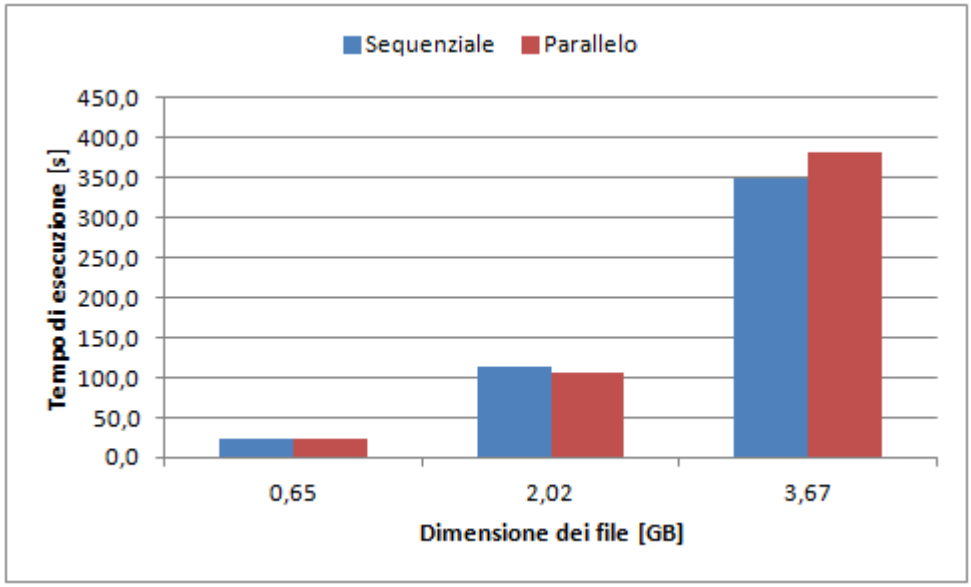

Fig. 4.5: Tempo di elaborazione per file nell'ordine dei Megabyte

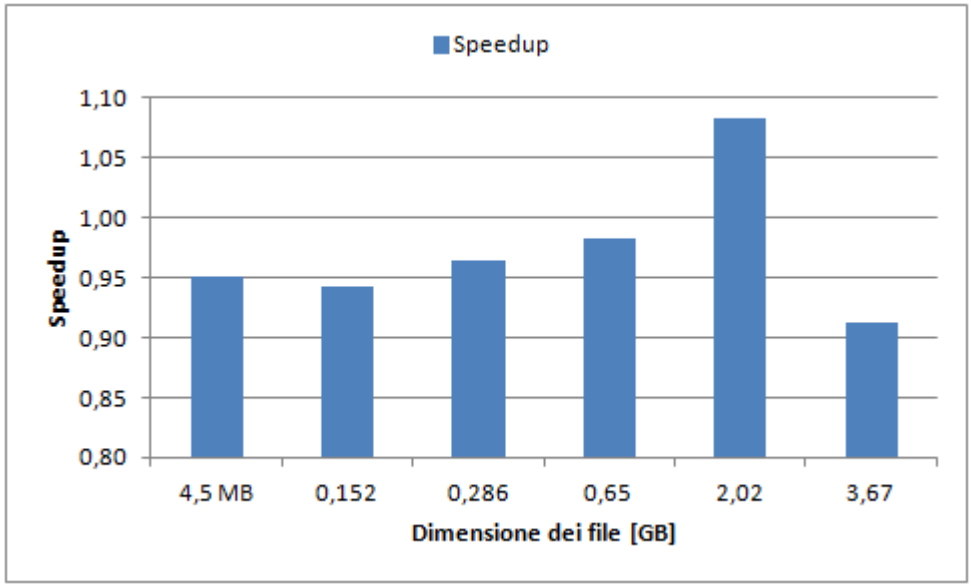

Fig. 4.6: Valori di speedup ottenuti

Capitolo 5

Conclusioni

Possiamo schematicamente definire gli obiettivi del progetto in questo modo:

1. Analisi della codifica di Huffman
2. Analisi delle Threading Building Blocks
3. Studio di soluzioni parallele
4. Implementazione dei costrutti paralleli
5. Test dell'algoritmo ottenuto per verificare lo *speedup* ottenuto

1. La parte teorica di studio della codifica di Huffman è stata effettuata tramite materiale presente in rete e in larga parte basandosi sugli appunti presi durante il corso di *Sistemi di elaborazione multimediale* (Prof. Costantino Grana) frequentato da tutti e tre i membri del gruppo.
E' stato inoltre recuperato il codice per la codifica di Huffman creato durante le ore di laboratorio del medesimo corso.

2. Durante la fase iniziale del progetto sono state analizzate a livello teorico le librerie Threading Building Blocks (TBB) della Intel; si è consultato prevalentemente il materiale e i codici di esempio presenti sul sito delle librerie. Questo passo ha richiesto un certo impegno a causa della scarna documentazione che in alcuni tratti non dava spiegazioni complete per determinati aspetti. Ulteriore tempo è stato dedicato allo studio delle *Lambda Expression* e dei *Functor*, la cui comprensione risulta essere necessaria per utilizzare correttamente le TBB.

3. In questa fase si è partiti con lo studio del codice prodotto in laboratorio: l'obiettivo era quello di comprendere dove un'eventuale implementazione dei

costrutti paralleli avrebbe migliorato le prestazioni globali e se tale implementazione era possibile da realizzare.
In particolare ci si è concentrati sui costrutti messi a disposizione dalle TBB, mostrati nel paragrafo 2.1.

4. La fase di implementazione è quella che ha richiesto il maggiore tempo, dovuto in parte alle difficoltà implementative delle TBB e in parte al fatto che sono state provate molteplici soluzioni per capire quale fosse la migliore.
Inoltre si è reso necessario implementare i meccanismi di gestione della memoria visti nel paragrafo 3.2 e anche in questo caso si sono provate più soluzioni per capire quale offrisse i maggiori vantaggi.

5. La raccolta dei dati riguardanti le *perfomance* dell'algoritmo è stata una fase particolamente importante e delicata, in quanto lo scopo principale del progetto era quello di ottenere uno *speedup* mediante l'utilizzo di costrutti paralleli in fase di programmazione.
Si è reso necessario fare un numero elevato di prove al fine di trovare anche il giusto settaggio dei vari parametri presenti all'interno del programma (come, ad esempio, la dimensione dei chunk in cui viene diviso il file in input).