

Università degli Studi di Salerno

Dipartimento di Informatica

Compressione Dati

Studio della trasformata di Burrows e Wheeler e l'applicazione negli algoritmi BZIP2 & PBZIP2

Antonio Leone – leonant@hotmail.it

06-2016

Indice

1 Introduzione	2
2 Cenni storici	4
3 Tecniche di compressione.....	6
4 Trasformata di Burrows e Wheeler (BWT).....	9
4.1 Algoritmo C.....	10
4.2 Algoritmo D.....	11
4.3 Varianti C & D	13
4.4 Compressione con la BWT	13
4.5 Move-To-Front	14
4.5.1 Algoritmo M: Codifica MTF	14
4.5.2 Algoritmo W: Decodifica MTF	16
5 Codici di Huffman.....	17
5.1 Algoritmo H: Codifica di Huffman.....	17
5.2 Decodifica di Huffman.....	19
5.3 Efficienza	19
6 Algoritmi della trasformata di B&W	21
6.1 Algoritmo di codifica della trasformata di B&W	21
6.2 Algoritmo di decodifica della trasformata di B&W.	22
6.3 Performance e confronto con altri programmi.....	23
6.3Conclusioni.	25
7 BZIP2.....	27
7.1 Opzioni.	30
7.2 Gestione della memoria.....	32
7.3 Ricostruzione di dati da file danneggiati.	34
7.4 Performance.....	35
8 BZIP2 Parallelo – PBZIP2	37
8.1 Parallelizzazione dell'algoritmo BWT	37
8.1.1 Ordinamento parallelo	37
8.1.2 Blocchi Multipli	37
8.1.3 Combinazione Ordinamento & Blocchi multipli.....	38
8.2 BZIP2 Sequenziale.....	38
8.4 PBZIP2.....	39

8.5 Risultati PBZIP2.....	40
8.5.1 Athlon-MP 2600+.....	41
8.5.2 Itanium 2	42
8.5.3 SunFire 6800	43
9 Conclusioni	45
10 Bibliografia	46

Abstract

Dopo dei brevi cenni storici a quelli che sono i compressori maggiormente utilizzati, in questo lavoro tratteremo la trasformata di B & W. In particolare scenderemo nel dettaglio dei due algoritmi utilizzati che compongono la trasformata, e l'algoritmo Move-To-Front, mostrando i vari passaggi con un esempio. Vengono quindi esposti i risultati che mostrano l'efficacia dell'algoritmo mostrato. Verranno poi mostrate le sue applicazioni nell'algoritmo di compressione BZIP2 che è diventato uno standard per le maggiori infrastrutture IT.

Successivamente analizzeremo l'algoritmo BZIP2 nella versione parallela, che funziona sempre con la trasformata di B&W, ma sfrutta più processori utilizzando pthread. Vedremo che l'output dell'algoritmo sarà completamente compatibile con la versione sequenziale di BZIP2 oggi in uso. I risultati mostrano una crescita quasi lineare su sistemi multiprocessore, a favore dei tempi di esecuzione.

1 Introduzione

In questo lavoro tratteremo la Trasformata di Burrows e Wheeler. Prima di addentrarci nel merito sono stati menzionati i tipi di compressori che si sono evoluti negli anni, ed i tipi di compressori esistenti, ovvero compressori statici e compressori con dizionari, presentando le varie differenze fra questi.

Viene così presentato il funzionamento della trasformata di B&W; questa non è una vera e propria tecnica di compressione, bensì una permutazione reversibile dell'ordine dei caratteri che, data in pasto ad un altro algoritmo (Move-to-front), permette di comprimere in modo più efficiente il testo in esame. In particolare l'algoritmo C tratta la codifica del testo da comprimere e l'algoritmo D ricostruisce il testo codificato a ritroso. A sua volta l'algoritmo Move-to-front è composto dagli algoritmi M e W che, rispettivamente, comprimono e decomprimono il testo a partire dalla trasformata di B&W. La fase successiva è quella di applicare l'algoritmo di Huffman per codificare in modo efficiente l'output dell'algoritmo Move-to-Front; con Huffman alle parole ripetute con più frequenza verrà assegnata una sequenza di bit minore rispetto alle parole ripetute con minor frequenza. Mostriamo quindi l'efficienza ottenuta con gli algoritmi applicati.

Presentiamo poi una possibile implementazione dell'algoritmo relativo alla trasformata di Burrows-Wheeler. Vengono mostrati i risultati della compressione dell'algoritmo di Burrows-Wheeler ottenuti su 14 file usati comunemente dal Calgary Compression Corpus. Questi risultati indicano che l'algoritmo lavora bene sia su file di testo che non.

L'algoritmo bzip2 è basato sulla trasformata di Burrows – Wheeler; spiegheremo nel dettaglio tutti i passi che vengono eseguiti da questo programma per comprimere e decomprimere. In generale la compressione ottenuta è considerata migliore rispetto a quella ottenuta dai più tradizionali algoritmi di compressione basati su LZ77/LZ78.

Nell'ultima sezione vedremo il calcolo parallelo, che consente al software di funzionare, quando possibile, più velocemente, attraverso l'uso di più processori. Sappiamo infatti che BZIP2 ha bisogno di una notevole potenza di elaborazione per analizzare e quindi codificare i dati per comprimerli. Mostriamo che il calcolo parallelo riduce i tempi di compressione, soprattutto quelli di taglia più grande. Verranno citati i vari tipi di parallelizzazione per l'algoritmo B&W e verranno comparate le performance di BZIP2 con quelle di PBZIP2 su diversi sistemi, che metteranno in evidenza significativi miglioramenti in termini di velocità.

2 Cenni storici

La compressione dati assunse un ruolo significativo a partire dagli anni '70, in contemporanea con l'aumento della popolarità di informatica e telematica e con la pubblicazione degli algoritmi di Abraham Lempel e Jacob Ziv, ma la sua storia ebbe inizio molto prima (1). I primi esempi di compressione dati si hanno in due codici molto famosi, il codice Morse ed il codice Braille. Il codice Morse (2) fu inventato nel 1838. In esso le lettere più comuni nella lingua inglese come 'E' e 'T' corrispondono a codici di minore lunghezza. Il codice Braille, fu introdotto nel 1829. La sua scrittura richiede molto più spazio della scrittura in nero, questo fatto ha comportato l'introduzione, in alcune lingue, di un codice modificato, detto "Braille contratto", in cui un singolo segno rappresenta singole parole o particolari gruppi di lettere (ad esempio, in inglese, si ricorre ad un carattere contratto per la rappresentazione del suffisso -ing).

In seguito, con l'invenzione dei primi calcolatori, nel 1949, Claude Shannon e Robert Fano inventarono la codifica di Shannon-Fano. Il loro algoritmo assegna parole di codice più corte ai simboli più frequenti.

Tre anni più tardi David Huffman elaborò una tecnica di compressione molto simile a quella di Shannon-Fano ma più efficiente. La differenza fondamentale tra queste tecniche sta nella costruzione dell'albero di probabilità, Huffman riuscì a creare un risultato ottimale. Nel corso degli anni '70, con la diffusione di Internet, iniziarono ad essere sviluppati molteplici software di compressione basati sulla codifica di Huffman. Nelle situazioni reali non si hanno conoscenze a priori sulle caratteristiche della sorgente, in pratica non si conoscono le probabilità dei

singoli elementi dell'alfabeto. Nasce la necessità di ricercare codici universali, cioè codici che non richiedono la conoscenza delle probabilità e che danno prestazioni prossime al tasso d'entropia della sorgente, senza conoscerlo, purché si codifichino N-ple sufficientemente lunghe.

Lempel e Ziv furono i primi a prendere in considerazione l'idea dell'utilizzo degli schemi a dizionario adattivo con i loro scritti del 1977 e del 1978. I due articoli descrivono due distinte versioni dell'algoritmo. Ci si riferisce ad esse con LZ77 o Lempel-Ziv con sliding window e con LZ78 o Lempel-Ziv con struttura ad albero. Entrambi gli algoritmi sono universali, crebbero rapidamente in popolarità e ne vennero sviluppate molte varianti. La maggior parte di queste scomparvero velocemente dopo la loro invenzione, mentre una ristretta minoranza è tuttora diffusa, come DEFLATE, LZMA, e LZX.

La fase di ordinamento della compressione mette insieme le stringhe simili nel file; vedremo che i file contenenti molte sequenze lunghe di simboli ripetuti possono essere compressi più lentamente rispetto alla norma. Vedremo che il rapporto del tempo di compressione tra il caso peggiore e quello medio è 10:1.

3 Tecniche di compressione

Le varie tecniche di compressione organizzano in modo più efficiente i dati. Sarà quindi necessario un decompressore per ricostruire i dati originali grazie all'algoritmo contrario a quello usato per la compressione. Come controparte la compressione dati necessita però di potenza di calcolo per le operazioni di compressione e decompressione, spesso anche elevata se tali operazioni devono essere eseguite in tempo reale.

Esistono diversi algoritmi di compressione, alcuni sono generali e offrono prestazioni simili su molte varietà di dati, altri sono più mirati e in genere permettono compressioni superiori e presuppongono che l'utente conosca in anticipo il tipo di dati da comprimere. Spesso le prestazioni della compressione dati dipendono da ciò che sappiamo delle caratteristiche del file sorgente, quindi dato un file, le sue caratteristiche possono essere usate per migliorare la compressione della sua stringa in output. Quando queste caratteristiche sono determinate prima della compressione, questa è detta conoscenza a priori delle caratteristiche dei dati da comprimere, ed è utile per ottenere algoritmi di compressione più efficienti. Ad ogni modo, nella maggior parte delle applicazioni reali non possiamo avere una conoscenza a priori delle caratteristiche del file. I diversi algoritmi di compressione possono essere raggruppati in due grandi categorie:

- ✓ Compressione statistica: si basa sullo studio dell'input da comprimere. Si fanno studi statistici sul formato dell'input per ottenere poi una buona compressione. Per esempio, nella compressione di un file di testo, si studia la frequenza relativa di ciascun carattere per associare poi al carattere

presente più volte nel testo il codice più corto, viceversa a caratteri presenti con frequenza bassa si associa la parola codice più lunga.

- ✓ Compressione con dizionario: questa è basata sull'idea di rimpiazzare, in un file, occorrenze di stringhe ripetute con puntatori a precedenti copie. La compressione è dovuta al fatto che la lunghezza di un puntatore è in genere più piccola della lunghezza della stringa che è rimpiazzata. Da ciò risulta che maggiori sono le ripetizioni di occorrenze di stringhe nel file da comprimere, maggiore è il grado di compressione raggiunta. Gli algoritmi di compressione basati sulla sostituzione, inoltre, vengono frequentemente usati poiché non richiedono una conoscenza a priori delle proprietà del file. Essi possono imparare in modo adattivo dalle caratteristiche del file durante la fase di codifica.

Fino agli anni '90 gli algoritmi di compressione più utilizzati erano sicuramente quelli basati sulle due tecniche di Lempel e Ziv, ovvero LZ77 e LZ78. Esse si basano su un dizionario *adattivo*, ossia aggiornato durante la fase di codifica in base all'evolversi dell'input processato, e trasmesso implicitamente nel testo codificato. Le tecniche di compressione statistica erano meno usate in quanto, seppur in grado di fornire una compressione migliore, la loro velocità è molto inferiore.

La trasformata di Burrows e Wheeler (3) (detta anche algoritmo Block-Sorting), scoperta da David Wheeler nel 1983 ma resa nota al mondo solo nel 1994, non è concettualmente un algoritmo di compressione a tutti gli effetti, e quindi non rientra in nessuna delle due categorie

definite poco fa. Si tratta infatti di una trasformazione reversibile (*loss/less*, ovvero senza perdita di informazione) che prende in input un blocco di testo e manda in output lo stesso testo con un opportuno riordinamento dei caratteri in esso contenuti, dunque senza modificare la dimensione dell'input. Il vantaggio di una trasformazione di questo tipo risiede nel fatto che il testo ottenuto può essere compresso più efficientemente rispetto all'originale con dei semplici e comuni algoritmi di compressione, in quanto essa tende a raggruppare le diverse istanze dello stesso carattere sparse nel testo. Questo approccio, ovvero l'idea di trasformare l'input per renderlo più adatto alla compressione, rappresentò una novità nel campo della Data Compression, anche se, dopo l'apparizione della trasformata di Burrows e Wheeler, alcuni ricercatori dimostrarono che essa è comunque legata ad altre tecniche di compressione precedentemente conosciute.

4 Trasformata di Burrows e Wheeler (BWT)

Il metodo di compressione presentato da Burrows e Wheeler in è una combinazione dei due seguenti algoritmi:

- ✓ Trasformata di Burrows-Wheeler: se applicata ad una stringa di caratteri non effettua una vera e propria compressione, ma permuta in maniera reversibile l'ordine dei caratteri. Se la stringa originale contiene molte sottostringhe che si ripetono spesso, la stringa trasformata ha numerose parti in cui un carattere viene ripetuto diverse volte di fila.
- ✓ Metodo Move-To-Front: è un semplice metodo di compressione utile a comprimere stringhe caratterizzate da caratteri ripetuti.

Il cuore dell'algoritmo BWT è costituito dalle due operazioni di trasformazione (chiamata nel documento "Algoritmo C") e antitrasformazione ("Algoritmo D") del blocco di testo.

4.1 Algoritmo C

L'algoritmo funziona nel modo seguente: si forma una matrice $N \times N$, che chiameremo M , le cui righe sono le rotazioni della stringa S in input, di N caratteri $S[0], \dots$,

```
a b r a c a d a b r a  
b r a c a d a b r a a  
r a c a d a b r a a b  
a c a d a b r a a b r  
c a d a b r a a b r a  
a d a b r a a b r a c  
d a b r a a b r a c a  
a b r a a b r a c a d  
b r a a b r a c a d a  
r a a b r a c a d a b  
a a b r a c a d a b r
```

$S[N-1]$ appartenenti ad un alfabeto X . Ad esempio, se la nostra stringa è $S = \text{'abracadabra'}$, M sarà la seguente matrice 11×11 :

Le righe della matrice devono quindi essere disposte in ordine lessicografico a partire dal primo carattere; nel nostro esempio, M diventa:

Infine, sia L l'ultima colonna di M e sia I l'indice della riga in cui appare la stringa originale; l'algoritmo restituisce in output L ed I . Nell'esempio: $L = \text{'rdarcaaabb'}$ e $I = 2$.

```
0 a a b r a c a d a b r  
1 a b r a a b r a c a d  
2 a b r a c a d a b r a  
3 a c a d a b r a a b r  
4 a d a b r a a b r a c  
5 b r a a b r a c a d a  
6 b r a c a d a b r a a  
7 c a d a b r a a b r a  
8 d a b r a a b r a c a  
9 r a a b r a c a d a b  
10 r a c a d a b r a a b
```

4.2 Algoritmo D

Come primo passo l'algoritmo di antitrasformazione calcola la prima colonna della matrice M vista in precedenza; ciò può essere fatto in maniera molto semplice ordinando i caratteri di L (dato che le righe di M sono ordinate lessicograficamente). Chiamiamo questa colonna F ; nel nostro esempio è:

$F = \text{'aaaaabbcdrr'}$

Ora siamo a conoscenza di L , I e F . Definiamo la matrice M' come la matrice ottenuta ruotando di una posizione verso destra tutte le righe di M , ossia $M'[i, j] = M[i, (j-1) \bmod N]$. Chiaramente, ad ogni riga di M' corrisponde una riga di M , poiché le righe di M' sono tutte rotazioni di S . In M' le righe sono ordinate a partire dal secondo carattere; si può quindi dedurre che ogni sottoinsieme di righe di M' che iniziano con lo stesso carattere ch appare in M' in ordine lessicografico. Ovviamente la stessa proprietà valeva anche per la matrice M ; notiamo anche che L è la prima colonna di M' .

M		M'
a a b r a c a d a b r	0	r a a b r a c a d a b
a b r a a b r a c a d	1	d a b r a a b r a c a
a b r a c a d a b r a	2	a a b r a c a d a b r
a c a d a b r a a b r	3	r a c a d a b r a a b
a d a b r a a b r a c	4	c a d a b r a a b r a
b r a a b r a c a d a	5	a b r a a b r a c a d
b r a c a d a b r a a	6	a b r a c a d a b r a
c a d a b r a a b r a	7	a c a d a b r a a b r
d a b r a a b r a c a	8	a d a b r a a b r a c
r a a b r a c a d a b	9	b r a a b r a c a d a
r a c a d a b r a a b	10	b r a c a d a b r a a

Considerando le righe che iniziano con il carattere 'a': le righe 0, 1, 2, 3 e 4 in M corrispondono alle righe 2, 5, 6, 7 e 8 in M'.

Ora, per mezzo di F e L calcoliamo un vettore T che indica la corrispondenza tra le righe delle due matrici M e M'.

Nel nostro esempio $T = [9\ 8\ 0\ 10\ 7\ 1\ 2\ 3\ 4\ 5\ 6]$

Usando la notazione $M'[j] = M[T[j]]$ per ogni $j = 0, \dots, N-1$, notiamo che se $L[j]$ è la k-esima istanza di *ch* in L, allora $T[j] = i$, dove $F[i]$ è la k-esima istanza di *ch* in F. Sostituendo, $F[T[j]] = L[j]$. Ora, per ogni $i = 0, \dots, N-1$, i caratteri $L[i]$ e $F[i]$ sono rispettivamente l'ultimo e il primo carattere della riga i di M; quindi il carattere $L[i]$ precede ciclicamente il carattere $F[i]$ in S. Sostituendo $i = T[j]$ nella relazione ricavata in precedenza, otteniamo che $L[T[j]]$ precede ciclicamente $L[j]$ in S.

A questo punto abbiamo tutti gli elementi per ricostruire la stringa S. Prendiamo innanzitutto in considerazione l'indice I: per definizione sappiamo che $L[I]$ è l'ultimo carattere di S, essendo la riga I di M quella contenente la stringa originaria. Usando il vettore T riusciamo a ricavare tutti i caratteri predecessori di $L[I]$: per ogni $i = 0, \dots, N-1$: $S[N-1-i] = L[T^i[I]]$, dove $T^0[x] = x$, e $T^{i+1}[x] = T[T^i[x]]$. Concluse le iterazioni, otteniamo S. Nel nostro esempio

$$S[10] = L[2] = 'a'$$

$$S[9] = L[T[2]] = L[0] = 'r'$$

$$S[8] = L[T[0]] = L[9] = 'b'$$

$$S[7] = L[T[9]] = L[5] = 'a'$$

$S[6] = L[T[5]] = L[1] = 'd'$

$S[5] = L[T[1]] = L[8] = 'a'$

$S[4] = L[T[8]] = L[4] = 'c'$

$S[3] = L[T[4]] = L[7] = 'a'$

$S[2] = L[T[7]] = L[3] = 'r'$

$S[1] = L[T[3]] = L[10] = 'b'$

$S[0] = L[T[10]] = L[6] = 'a'$

4.3 Varianti C & D

Ci sono diverse varianti degli algoritmi C e D, una delle quali ad esempio è quella di definire T in modo che S venga generata a partire dal primo carattere verso destra invece che dall'ultimo. Un'altra variante può essere quella di ordinare lessicograficamente da destra verso sinistra le righe della matrice M nell'algoritmo C, prendendo poi al posto della colonna L la prima colonna della matrice, e così via. Tuttavia si tratta solo di questioni di notazione: l'efficienza della compressione sulla stringa trasformata resta invariata.

4.4 Compressione con la BWT

Dopo aver presentato le operazioni di trasformazione e antitrasformazione c'è da spiegare perché il testo trasformato con la BWT viene compresso in maniera più efficiente rispetto all'originale.

Innanzitutto, prima di addentrarci in un'analisi vera e propria, portiamo un esempio. Consideriamo un blocco di testo in lingua inglese: è ragionevole pensare che la parola "the" appaia svariate volte. Quando disponiamo le

rotazioni del testo in ordine lessicografico, molte tra le rotazioni che iniziano in “he” finiranno con il carattere “t” (altre potrebbero finire in “s” o in “ ”, che però di solito sono più rare); quindi l’operazione di trasformazione tende a raggruppare insieme nella stringa L molte tra le occorrenze del carattere “t” nel testo. Dunque se guardiamo meglio L possiamo notare che in diverse zone tendono a raggrupparsi molte occorrenze degli stessi caratteri intervallate sporadicamente da pochi altri caratteri. E’ questa la forza della BWT: infatti la probabilità che un certo carattere appaia in L vicino a un’altra occorrenza dello stesso carattere è molto alta.

Questa proprietà spiana la strada ad una codifica di tipo Move-To-Front (MTF), che per funzionare al meglio deve aver a che fare con stringhe con molti caratteri consecutivi ripetuti; l’output del codificatore MTF potrà quindi essere efficientemente compresso con una codifica di ordine zero, come la codifica di Huffman o quella aritmetica. Vediamo ora più in dettaglio i passi appena citati per la vera e propria compressione, da combinare con gli algoritmi C e D visti in precedenza. Analizzeremo in particolare la codifica MTF e la codifica di Huffman.

4.5 Move-To-Front

L’algoritmo MTF codifica ogni carattere *ch* di una stringa in un numero intero, in base al numero di istanze di caratteri diversi visti dopo l’ultima occorrenza di *ch*. La decodifica è in tutto e per tutto speculare alla codifica.

4.5.1 Algoritmo M: Codifica MTF

La codifica processa L carattere per carattere, andando a guardare ad ogni iterazione la posizione del carattere corrente in un’apposita lista Y di caratteri dinamicamente

aggiornata. Più precisamente, Y contiene un'occorrenza per ogni carattere dell'alfabeto X (dunque avrà lunghezza $|X|$) e deve essere inizializzata prima della scansione di L . Costruiamo un vettore di interi $R[0], \dots, R[N-1]$. Ora, per ogni $i=0, \dots, N-1$, contiamo il numero di caratteri che precedono il carattere $L[i]$ nella lista Y , mettiamo questo valore in $R[i]$ e subito dopo spostiamo il carattere $L[i]$ in testa ad Y . Mostriamo di seguito i vari passi dell'algoritmo applicati al nostro solito esempio:

$L = \text{'rdarcaaaabb'}$

Inizializziamo Y arbitrariamente: $Y = [a\ b\ c\ d\ r]$

$Y = [a\ b\ c\ d\ r], L[0] = \text{'r'} \rightarrow R[0] = 4$

$Y = [r\ a\ b\ c\ d], L[1] = \text{'d'} \rightarrow R[1] = 4$

$Y = [d\ r\ a\ b\ c], L[2] = \text{'a'} \rightarrow R[2] = 2$

$Y = [a\ d\ r\ b\ c], L[3] = \text{'r'} \rightarrow R[3] = 2$

$Y = [r\ a\ d\ b\ c], L[4] = \text{'c'} \rightarrow R[4] = 4$

$Y = [c\ r\ a\ d\ b], L[5] = \text{'a'} \rightarrow R[5] = 2$

$Y = [a\ c\ r\ d\ b], L[6] = \text{'a'} \rightarrow R[6] = 0$

$Y = [a\ c\ r\ d\ b], L[7] = \text{'a'} \rightarrow R[7] = 0$

$Y = [a\ c\ r\ d\ b], L[8] = \text{'a'} \rightarrow R[8] = 0$

$Y = [a\ c\ r\ d\ b], L[9] = \text{'b'} \rightarrow R[9] = 4$

$Y = [b\ a\ c\ r\ d], L[10] = \text{'b'} \rightarrow R[10] = 0$

$\rightarrow R = [4\ 4\ 2\ 2\ 4\ 2\ 0\ 0\ 0\ 4\ 0]$.

4.5.2 Algoritmo W: Decodifica MTF

Assumiamo che il valore iniziale della lista Y usata nell'algoritmo W sia nota al decodificatore. L'operazione di decodifica avviene in maniera del tutto analoga alla codifica: per ogni $i = 0, \dots, N-1$, prendiamo $R[i]$, guardiamo la sua posizione in Y (contando da zero) e mettiamo questo valore in $L[i]$, e subito dopo spostiamo il carattere $L[i]$ in testa ad Y . Nel nostro esempio,

$R = [4\ 4\ 2\ 2\ 4\ 2\ 0\ 0\ 4\ 0]$, $Y = [a\ b\ c\ d\ r]$

$Y = [a\ b\ c\ d\ r]$, $R[0] = 4 \rightarrow \square L[0] = 'r'$

$Y = [r\ a\ b\ c\ d]$, $R[1] = 4 \rightarrow \square L[1] = 'd'$

$Y = [d\ r\ a\ b\ c]$, $R[2] = 2 \rightarrow \square L[2] = 'a'$

$Y = [a\ d\ r\ b\ c]$, $R[3] = 2 \rightarrow \square L[3] = 'r'$

$Y = [r\ a\ d\ b\ c]$, $R[4] = 4 \rightarrow \square L[4] = 'c'$

$Y = [c\ r\ a\ d\ b]$, $R[5] = 2 \rightarrow \square L[5] = 'a'$

$Y = [a\ c\ r\ d\ b]$, $R[6] = 0 \rightarrow \square L[6] = 'a'$

$Y = [a\ c\ r\ d\ b]$, $R[7] = 0 \rightarrow \square L[7] = 'a'$

$Y = [a\ c\ r\ d\ b]$, $R[8] = 0 \rightarrow \square L[8] = 'a'$

$Y = [a\ c\ r\ d\ b]$, $R[9] = 4 \rightarrow \square L[9] = 'b'$

$Y = [b\ a\ c\ r\ d]$, $R[10] = 0 \rightarrow \square L[10] = 'b'$

$\rightarrow \square L = 'rdarcaaaabb'$.

5 Codici di Huffman

A questo punto abbiamo la stringa R e l'indice I che ci eravamo portati dietro dall'algoritmo C . L'ultimo passo dell'algoritmo di compressione può essere realizzato con un qualsiasi codificatore di ordine 0; tra questi, il più noto usa l'algoritmo di Huffman. L'idea che sta alla base dell'algoritmo è quella di rappresentare ogni carattere presente nel testo con un certo numero di bit, in base alla frequenza con cui questo carattere appare nel testo: in questo modo, i caratteri più frequenti saranno rappresentati con pochi bit, mentre quelli meno frequenti con qualche bit in più. In più, i codici assegnati ai diversi caratteri hanno la proprietà di essere liberi da prefissi: ovvero, nessun codice è prefisso di un altro. Vediamo come funziona l'algoritmo di Huffman applicato all'output del nostro algoritmo M .

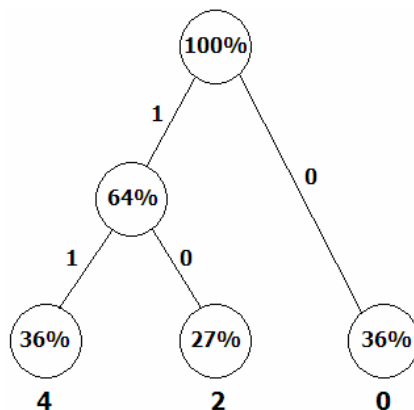
5.1 Algoritmo H: Codifica di Huffman

L'algoritmo si divide in due parti: l'assegnazione di un codice ad ogni carattere dell'alfabeto $|A|$ contenuto in R e la trasformazione di R nella stringa di bit Q .

Il primo passo calcola per ogni carattere (in questo caso per ogni intero) la frequenza in percentuale con cui questo appare in R che viene dunque associata ad esso. Quindi ogni coppia (carattere, frequenza) viene memorizzata in un nodo isolato di un albero. Si guardano ora i due nodi a minor frequenza, e si assegna ad esse un nodo padre con frequenza uguale alla somma delle frequenze dei suoi figli. Il procedimento viene eseguito $|A|-1$ volte, finché non risulterà un albero connesso con radice a frequenza 100%. I codici per ogni carattere vengono generati "navigando" l'albero dalla radice fino al nodo foglia che contiene quel

$$R = [4 \ 4 \ 2 \ 2 \ 4 \ 2 \ 0 \ 0 \ 0 \ 4 \ 0]$$

Carattere	# occorrenze	Frequenza
0	4/11	36,36%
2	3/11	27,27%
4	4/11	36,36%



Quindi: $0 \rightarrow 0$, $2 \rightarrow 10$, $4 \rightarrow 11$.

18

$R = [4\ 4\ 2\ 2\ 4\ 2\ 0\ 0\ 0\ 4\ 0] \rightarrow Q = 111110101110000110.$

5.2 Decodifica di Huffman

Supponendo di conoscere già l'albero di Huffman per la stringa di bit che vogliamo decodificare, l'algoritmo H' processa Q da sinistra a destra e percorre l'albero dalla radice seguendo ad ogni iterazione il ramo etichettato con il bit corrente; quando viene raggiunta una foglia viene restituito il carattere corrispondente e il percorso ricomincia dalla radice. E' nella fase di decodifica che si nota la necessità di un codice libero da prefissi. Nel nostro esempio, la trasformazione è ovviamente l'inversa di quella dell'algoritmo H:

$Q = 111110101110000110 \rightarrow R = [4\ 4\ 2\ 2\ 4\ 2\ 0\ 0\ 0\ 4\ 0].$

5.3 Efficienza

Complessivamente l'intera fase di compressione è composta dagli algoritmi C, M e H: chiameremo BW0 il "super-algoritmo" che li esegue in cascata. In output a BW0 vengono restituiti la stringa di bit Q e l'indice I che avevamo ottenuto in output all'algoritmo C.

Consideriamo l'esempio che ci portiamo sulle spalle dall'inizio di questo lavoro e vediamo se la compressione congiunta alla BWT sia davvero migliore rispetto alla compressione senza la trasformazione iniziale. Se proviamo a codificare la stringa $S = \text{'abracadabra'}$ direttamente con gli algoritmi M e H in cascata, usando la stessa inizializzazione della lista Y, ciò che otteniamo è una stringa di 24 bit, contro gli 88 che sarebbero stati

necessari se avessimo memorizzato la stringa S senza comprimerla (supponendo di usare una codifica ASCII estesa a 8 bit); quindi lo spazio occupato è in rapporto il $24/88 \approx 27\%$ e vengono usati in media circa 2,18 bit per carattere.

Da notare che nel calcolo dell'efficienza non abbiamo considerato l'overhead dovuto alla memorizzazione dell'indice I, della lista Y e dell'albero di Huffman; tuttavia l'effetto sul rapporto di compressione all'aumentare della lunghezza di S (l'algoritmo viene generalmente applicato a blocchi di dati dell'ordine dei MB) tende a zero.

Per la dimostrazione matematica formale rifarsi a (4)

6 Algoritmi della trasformata di B&W

In questa sezione presentiamo una possibile implementazione dell'algoritmo relativo alla trasformata di Burrows-Wheeler.

6.1 Algoritmo di codifica della trasformata di B&W

L'algoritmo **CodificaBW** prende in input la stringa **S'** e restituisce in output la coppia (**OUT**, **I**) in cui il valore di **I** indica la posizione della stringa **S'** nella matrice **M** ed **OUT** indica la codifica Move-to-front della stringa **L** che è l'ultima colonna della matrice **M**.

CodificaBW(S')

```
1  n ← lenght(S')
2  V ← Sort (S')
3  for i ← 0 to n - 1 do
4      if V[i] = 0 then
5          I ← i
6      end
7      L[i] ← S[V[i] - 1]
8  end
9  OUT ← M(L, I)
10 return(OUT, I)
```

Notiamo che nella linea 2 viene richiamata la procedura **Sort** che effettua l'ordinamento dei suffissi della stringa **S'**.

Il vettore **V**, inizializzato nell'algoritmo **Sort**, è un vettore di interi che, come detto, rappresenta l'indice dell'*i*-esimo suffisso in ordine lessicografico, e come già detto ordinare i suffissi di una stringa è la stessa cosa che ordinare le

rotazioni della stringa stessa, per cui il vettore **V** contiene l'ordinamento delle rotazioni della stringa **S'**.

Nella terza linea dell'algoritmo troviamo un ciclo **for** che calcola la stringa **L** che rappresenta il risultato della trasformata di Burrows-Wheeler la quale verrà codificata tramite la procedura **M** la quale ne calcola la codifica Move-to-front.

6.2 Algoritmo di decodifica della trasformata di B&W.

L'algoritmo **DecodificaBW** prende in input la coppia (**OUT**, **I**), l'output dell'algoritmo **CodificaBW**, per ricostruire la stringa iniziale **S'** di lunghezza **n**.

OUT rappresenta la codifica Move-to-front, ed **I** la posizione della stringa **S'** nella matrice **M**.

Indichiamo con **C** un array di interi in cui **C[ch]** rappresenta il numero di caratteri minori di **ch**, con **P** un array di interi in cui ogni **P[i]** rappresenta il numero di occorrenze di caratteri **L[i]** nel prefisso **L[0, ..., i-1]** e definiamo il vettore **T**.

DecodificaBW(OUT, I)

```
1  (L, n) ← W(OUT)
2  for ogni carattere ch ∈  $\Sigma$  do
3      C[ch] ← 0
4  for i ← 0 to n-1 do
5      P[i] ← C[L[i]]
6      C[L[i]] ← C[L[i]] + 1
7  end
8  somma ← 0
```

```

9   for ogni carattere in ordine lessicografico  $ch \in \Sigma$ 
    do
10      somma  $\leftarrow$  somma +  $C[ch]$ 

11       $C[ch] \leftarrow$  somma -  $C[ch]$ 
12  end
13   $i \leftarrow I$ 
14  for  $j \leftarrow n-1$  downto 0 do
15       $S'[j] \leftarrow L[i]$ 
16       $i \leftarrow P[i] + C[L[i]]$ 
17  end
18  return  $S'$ 

```

Notiamo che, inizialmente tutti gli elementi del vettore C sono messi a zero con il ciclo **for** della linea 2.

Nel ciclo **for** della linea 4 si inizializza $C[ch]$ con il numero di occorrenze in L del carattere ch .

Al termine del ciclo **for** della linea 9 un carattere generico $C[ch]$ conterrà il numero di caratteri minori di ch .

Il risultato dell'algoritmo lo si ha alla fine del ciclo **for** della linea 14 il quale è rappresentato dall'array $S'[0, \dots, n-1]$.

6.3 Performance e confronto con altri programmi.

Nella Tabella 1, di seguito riportata, vengono espressi i risultati della compressione dell'algoritmo di Burrows-Wheeler ottenuti su 14 file usati comunemente dal Calgary Compression Corpus.

Questi risultati indicano che l'algoritmo lavora bene sia su file di testo che non.

Nell'implementazione di questo algoritmo la dimensione n del blocco è la lunghezza del file che deve essere compresso.

L'efficienza della compressione è espressa come bit di output per ogni carattere di input.

La misura del tempo di CPU è stata effettuata su una DECstation 5000/200, la quale ha un processore MIPS R 3000 sincronizzato a 25MHz con 64 byte di cache.

Tabella 1 - Variazioni dell'efficienza della compressione

File	Dimensione in bytes	Tempo di CPU compressione	in secondi decompressione	Dimensione della compressione in bytes	bit per caratter e
bib	111.261	1.6	0.3	28.750	2.07
book1	768.771	14.4	2.5	238.989	2.49
book2	610.856	10.9	1.8	162.612	2.13
geo	102.400	1.9	0.6	56.974	4.45
news	377.109	6.5	1.2	122.175	2.59
obj1	21.504	0.4	0.1	10.694	3.98
obj2	246.814	4.1	0.8	81.337	2.64
paper1	53.161	0.7	0.1	16.965	2.55
paper2	82.199	1.1	0.2	25.832	2.51
pic	513.216	5.4	1.2	53.562	0.83
progc	39.611	0.6	0.1	12.786	2.58
prog1	71.646	1.1	0.2	16.131	1.80
progp	49.379	0.8	0.1	11.043	1.79
trans	93.695	1.6	0.2	18.383	1.57
TOT	3.141.622	51.1	9.4	856.233	—

Nella Tabella 2 vengono mostrate le variazioni dell'efficienza della compressione considerando la grandezza del blocco in input ogni volta diversa per 2 input.

Il primo input è il file "book1" del Calgary Compression Corpus (una raccolta di file di testo e binari utilizzata comunemente negli anni '90 per il confronto degli algoritmi di compressione dei dati.); il secondo è l'intero "Hector corpus" che contiene circa 100 MByte

di testo in inglese moderno. Come si evince dalla tabella, la compressione migliora con il crescere della dimensione del blocco di input, anche quando esso è abbastanza grande.

Se la dimensione del blocco di input crescesse indefinitamente, ci si aspetta che l'algoritmo di Burrows-Wheeler non raggiunge la compressione ottima, poiché la codifica Move-to-front introduce alcune perdite.

6.3 Conclusioni.

In questi paragrafi abbiamo descritto una tecnica di compressione che lavora applicando una trasformazione reversibile ad un blocco di testo, creando ridondanze nell'input più accessibili per semplificare gli schemi di codifica.

L'algoritmo di Burrows-Wheeler descritto è general-purpose, ossia esso lavora bene su file di input che sono sia formato testo che non.

Per ottenere una buona compressione, sono necessari blocchi di input di alcune migliaia di caratteri.

L'efficienza dell'algoritmo continua a migliorare con il crescere della dimensione del blocco di input almeno fino a blocchi di alcuni milioni di caratteri.

L'algoritmo di Burrows-Wheeler ottiene una compressione confrontabile con buoni modelli statistici, ma è ancora meno veloce dei codificatori basati sull'algoritmo di Lempel e Ziv.

Come gli algoritmi di Lempel e Ziv, esso decompone più velocemente rispetto alla compressione.

Anche se l'algoritmo di compressione Burrows - Wheeler è di dominio pubblico, i programmatori devono essere cauti quando usano il codificatore per il passo finale della compressione.

I codici aritmetici offrono un'eccellente compressione, ma sono coperti da qualche brevetto. I codici di Huffman, anche se leggermente meno efficienti, sembrano meno adatti a provocare battaglie legali.

7 BZIP2

L'algoritmo bzip2 (5) è basato sulla trasformata di Burrows - Wheeler, i cui passi principali sono: data la stringa di input S viene costruita la matrice M che contiene gli shift ciclici di tale stringa in ordine lessicografico, da tale matrice viene prelevata la prima colonna e viene così ottenuta la stringa L, da essa viene costruita la stringa R che contiene il numero di caratteri che precedono un determinato carattere in L nell'alfabeto dopodiché tale carattere viene spostato all'inizio dell'alfabeto, ad ogni elemento di R viene applicato il codice di Huffman e viene così ottenuta la stringa OUT che è la stringa compressa di S. Per la decompressione si applica il processo inverso.

In generale la compressione ottenuta è considerata migliore rispetto a quella ottenuta dai più tradizionali algoritmi di compressione basati su LZ77/LZ78 (6).

Le opzioni della linea di comando sono deliberatamente molto simili, ma non identiche, a quelle di GNU *gzip*.

Bzip2 si aspetta una lista di nomi di file subito dopo i flag della linea di comando. Ogni file è rimpiazzato dalla versione compressa di se stesso con il nome "*nome_originale.bz2*".

Ogni file compresso ha la stessa data di modifica, i permessi, e quando possibile, la proprietà del corrispondente file originario, cosicché queste proprietà possono essere correttamente ripristinate durante la fase di decompressione.

La manipolazione del nome del file è semplice, nel senso che non c'è un meccanismo per mantenere i nomi del file originale, i permessi, le proprietà o le date nel file system

al quale mancano questi concetti oppure ha importanti limitazioni sulla lunghezza del nome del file, come nel sistema operativo MS-DOS.

Bzip2 e *bunzip2* non riscrivono, per default, su file esistenti, se invece ciò si vuole basta specificare il flag “-f”.

Se non viene specificato nessun nome del file, **bzip2** comprimerà dallo standard input allo standard output.

Bunzip2, o **bzip2 -d**, decompime tutti i file specificati.

I file che non sono stati creati da **bzip2** saranno scoperti ed ignorati, ed un warning (errore) sarà emesso.

Bzip2 tenta di trovare il nome del file decompresso da quello del file compresso come segue:

- ❖ *filename.bz2* diventa *filename*
- ❖ *filename.bz* diventa *filename*
- ❖ *filename.tbz2* diventa *filename.tar*
- ❖ *filename.tbz* diventa *filename.tar*
- ❖ *anyothername* diventa *anyothername.out*

Se il file non termina in uno di questi modi: “.bz2”, “.bz”, “.tbz2”, “.tbz”, **bzip2** purtroppo non può calcolare il nome del file originale per cui usa il nome originale al quale appende l'estensione “.out”.

Come per la compressione se non viene fornito il nome del file *bunzip2* decomprimerà dallo standard input allo standard output.

Bunzip2 decomprimerà correttamente un file che è la concatenazione di due o più file compressi, il risultato è la concatenazione dei corrispondenti file non compressi.

È anche supportato il testing di integrità sui file concatenati compressi inserendo il flag “-t”.

Si può anche comprimere o decomprimere file dallo standard output semplicemente inserendo il flag “-c”.

Più file possono essere compressi e decompressi allo stesso modo poi gli output risultanti vengono sequenzialmente forniti allo standard output.

La compressione di più file in questo modo genera un flusso contenente le rappresentazioni dei più file compressi; tale flusso può essere decompresso correttamente solo dalla versione 0.9.0 o successive di **bzip2**.

Le vecchie versioni di **bzip2** si fermeranno subito dopo la decompressione del primo file nel flusso.

Bzcat, o **bzip2 -dc**, decomprime tutti i file specificati nello standard output.

bzip2 leggerà gli argomenti dalle variabili d'ambiente **BZIP2** e **BZIP** nell'ordine, poi li processerà prima degli argomenti letti dalla linea di comando, utile per fornire argomenti di default.

La compressione è sempre eseguita anche se il file compresso è leggermente più grande dell'originale.

I file più piccoli di cento bytes tendono a diventare più grandi tramite il meccanismo di compressione hanno un costante overhead di cinquanta byte.

Dati casuali, incluso l'output di più file compressi, sono codificati con circa 8.05 bits per ogni byte ottenendo così un'espansione che si aggira allo 0,5%.

bzip2 usa la sequenza CRC a 32 bit per essere sicuro che la versione decompressa di un file sia identica all'originale.

Questo protegge dai dati compressi "corrotti" e dai virus non scoperti in **bzip2**.


La probabilità che i dati "corrotti" siano non trovati è microscopica, infatti è circa 1 su 4 miliardi per ogni file processato, purtroppo però ci accorgiamo di ciò solo durante la fase di decompressione quando veniamo informati che si è verificato un errore.


bzip2 non può aiutarci a ricostruire i dati non compressi originali, ma è possibile usare *bzip2recover* per tentare di ricostruire i dati dai file danneggiati.

I valori ritornati sono:








- uscita senza errore;
- problemi ambientali: errori di I/O, file non trovato, flags non valido.....;
- file compresso "corrotto";
- errore di coerenza interna, ad esempio i virus, che mette **bzip2** in allarme.

7.1 Opzioni.

 **-c** "*stdout*" Comprime o deprime dallo standard output.

 **-d** "*decompress*" Forza la decompressione. **bzip2**, *bunzip2* e *bzcat* sono realmente lo stesso

programma e la decisione su quali azioni compiere è fatta sulla base di quale nome viene usato. Questo flag non tiene conto di questo meccanismo e forza **bzip2** alla decompressione.

-  **-z** “*compress*” Complemento di **-d**: forza la compressione.
-  **-t** “*test*” Controlla l’integrità del/dei file specificati, ma non li decomprime. In realtà viene eseguita una prova di decompressione e si butta via il risultato.
-  **-f** “*force*” Forza la riscrittura dei file di output. Normalmente **bzip2** non riscrive i file di output esistenti, così viene forzato a rompere i links ai file, cosa che altrimenti non avrebbe fatto.
-  **-k** “*keep*” Mantiene, non cancella, i file di input durante la compressione / decompressione.
-  **-s** “*small*” Riduce l’uso della memoria per la compressione, la decompressione ed il testing. I file sono decompressi e testati usando un algoritmo modificato che richiede solo 2.5 bytes per blocco. Questo significa che un file può essere decompresso in 2300k di memoria anche se con una velocità dimezzata. Durante la compressione **-s** seleziona un blocco grande 200k, che è una via di mezzo tra il risparmio di spazio di memoria e la percentuale di compressione.
-  **-q** “*quiet*” Abolisce i messaggi d’errore non essenziali. I messaggi che fanno riferimento agli errori di I/O ed altri eventi critici non possono essere soppressi.
-  **-v** “*verbose*” Mostra la percentuale di compressione per ogni file processato. Inoltre viene incrementato il livello di verbosità per cui fuoriescono delle informazioni che interessano principalmente per scopi diagnostici.

- + **-L** “*license*” Mostra il termine di licenza e le condizioni.
- + **-V** “*version*” Mostra la versione del software.
- + da **-1** (o “*fast*”) a **-9** (o “*best*”) Pone la grandezza del blocco a 100K, 200K, ..., 900K quando comprime, mentre non produce effetti quando decompime. Gli pseudonimi *fast* e *best* servono principalmente per la compatibilità con GNU *gzip*. In particolare *fast* non produce cose significativamente più veloci e *best* seleziona solo il comportamento di default.
- + **--** Tratta tutti gli argomenti successivi come nomi di file, anche se iniziano con “-“. In questo modo si possono manipolare i file che iniziano con “-“ ad esempio: **bzip2 -- -nomefile**.
- + **-- *repetitive-fast*, --repetitive-best** Questi flags sono ridondanti nella versione 0.9.5 e oltre. Essi facevano dei controlli grossolani sul comportamento dell’algoritmo di ordinamento delle prime versioni, i quali qualche volta erano utili. La versione 0.9.5 e oltre ha un algoritmo migliorato che rende questi flags irrilevanti.

7.2 Gestione della memoria.

BZIP2 comprime grandi file in blocchi.

La grandezza del blocco influisce sia sul percentuale di compressione ottenuta che sulla quantità di memoria necessaria per la compressione e la decompressione.

I flags da **-1** a **-9** specificano la grandezza del blocco da 100.000 a 900.000 bytes rispettivamente.

Durante la fase di decompressione, la grandezza del blocco usata per la compressione è letta dall'intestazione del file compresso, così *bunzip2* stanziava abbastanza memoria per decomprimere il file.

Poiché le grandezze dei blocchi sono memorizzate nei file compressi, da ciò segue che i flags da **-1** a **-9** sono irrilevanti e così ignorati durante la decompressione.

I bytes necessari per la fase di compressione e decompressione possono essere così stimati:

Compressione: $400\text{ K} + (8 * \text{grandezza_blocco})$

Decompressione: $100\text{ K} + (4 * \text{grandezza_blocco})$ oppure

$100\text{ K} + (2.5 * \text{grandezza_blocco})$

La maggior parte della compressione viene fatta sui primi 200 o 300 K della grandezza del blocco e questa cosa è bene tenerla presente quando viene usato **bzip2** su macchine piccole.

È anche importante apprezzare il fatto che la memoria richiesta per la decompressione è settata al momento della compressione dalla scelta della grandezza del blocco.

In generale, usare blocchi di memoria più grandi permette di massimizzare la compressione ottenuta.

La velocità di compressione e decompressione non sono virtualmente influenzate dalla grandezza del blocco.

7.3 Ricostruzione di dati da file danneggiati.

Bzip2, come detto, comprime i file in blocchi, abitualmente grandi 900 K bytes.

Ogni file è manipolato indipendentemente.

Se un errore di trasmissione danneggia un file “.bz2” multiblocco, deve essere possibile ricostruire i dati dal blocco non danneggiato nel file.

La rappresentazione compressa di ogni blocco è delimitata da un percorso a 48 bit e ciò rende possibile trovare i confini del blocco con una certezza ragionevole.

Ogni blocco contiene anche la sua sequenza CRC a 32 bit, così i blocchi danneggiati possono essere distinti da quelli non.

Bzip2recover è un semplice programma che ha lo scopo di cercare i blocchi nei file “.bz2” e scrivere ogni blocco nel proprio file “.bz2”.

Si può poi usare **bzip2 -t** per testare l'integrità dei file risultanti e decomprimere quelli che non sono danneggiati.

Bzip2recover prende un singolo argomento, il nome del file danneggiato, e scrive un numero di file: *rec0001file.bz2*, *rec0002file.bz2*, ..., contenenti i blocchi estratti.

I nomi dei file di output sono stati ideati cosicché l'uso dei caratteri jolly nelle trattazioni successive listano i file nell'ordine corretto.

Bzip2recover dovrebbe essere il più usato per i file “.bz2” grandi, poiché essi conteranno molti blocchi.

È ovviamente inutile usarlo per file danneggiati con un singolo blocco, poiché un blocco danneggiato non può essere ricostruito.

Se si vuole minimizzare la perdita di dati ottenuta attraverso gli errori di trasmissione, si deve considerare la compressione con la grandezza dei blocchi più piccola.

7.4 Performance.

La fase di ordinamento della compressione mette insieme le stringhe simili nel file.

A causa di questo, i file contenenti molte sequenze lunghe di simboli ripetuti possono essere compressi più lentamente rispetto alla norma.

Nella versione 0.9.5 e successive ciò viene fatto molto meglio rispetto alle precedenti versioni.

Il rapporto del tempo di compressione tra il caso peggiore e quello medio è 10:1, mentre nelle precedenti versioni era 100:1.

È possibile usare l'opzione `-vvvv` per monitorare i progressi con maggiore dettaglio.

La velocità di decompressione non è affetta da questo fenomeno.

bzip2 alloca diversi megabytes di memoria per operare e poi fa tutto da solo in modo abbastanza casuale.

Questo significa che le performance, sia per la compressione che per la decompressione, dipendono fortemente dalla velocità con cui la macchina lavora.

In riferimento a quanto detto, è stato osservato che piccoli cambiamenti al codice per ridurre la percentuale di perdita danno sproporzionatamente grandi miglioramenti sulla performance.

8 BZIP2 Parallelo – PBZIP2

Il calcolo parallelo consente al software di funzionare, quando possibile, più velocemente, attraverso l'uso di più processori. La maggior parte dei software in uso sono progettati in modo sequenziale, lo stesso vale per BZIP2 (7). Sappiamo però, per quanto detto fin ora, che BZIP2 ha bisogno di una notevole potenza di elaborazione per analizzare e quindi codificare i dati per comprimerli. Il calcolo parallelo ridurrebbe quindi i tempi di compressione, soprattutto quelli di taglia più grande.

L'algoritmo BWT non processa i dati sequenzialmente, ma prende i blocchi di dati come unità singole. Il blocco trasformato contiene gli stessi caratteri del blocco originale in forma più facilmente comprimibile dall'algoritmo.

8.1 Parallelizzazione dell'algoritmo BWT

Per parallelizzare l'algoritmo BWT in un'architettura a memoria condivisa ci sono tre modi: ordinamento parallelo, blocchi multipli o combinazione degli ultimi due.

8.1.1 Ordinamento parallelo

La trasformata di B&W la maggior parte del tempo di esecuzione lo impiega per ordinare l'input. Parallelizzare l'algoritmo di ordinamento può aumentare la velocità del BWT su una macchina parallela.

8.1.2 Blocchi Multipli

Per far fronte alla mole di dati che devono essere processati dall'algoritmo, i dati vengono divisi in blocchi indipendenti di taglia fissa. Questo può indurre a pensare che è possibile processare più blocchi simultaneamente.

Un'implementazione parallela avrà bisogno di tenere traccia dell'ordinamento dei blocchi e di scrivere i blocchi compressi sul disco nell'ordine corretto.

8.1.3 Combinazione Ordinamento & Blocchi multipli

A seconda del numero di processori della macchina parallela, potrebbe essere utile combinare l'esecuzione dell'algoritmo su blocchi multipli e contemporaneamente utilizzare un'implementazione parallela dell'algoritmo di ordinamento lessicografico

8.2 BZIP2 Sequenziale

BZIP2 elabora i dati in blocchi che vanno da 100.000 a 900.000 byte. La dimensione del blocco predefinito è 900.000 byte. Si leggono 5000 byte di dati alla volta, fino a riempire la size del blocco. Questo fino a quando viene elaborato l'intero set di dati. (Vedi Figura 1)

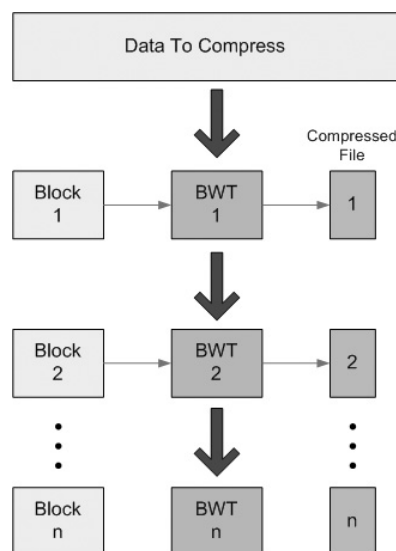


Figura 1 - Algoritmo sequenziale BZIP2

8.4 PBZIP2

Per lavorare su macchine parallele, è stata progettata una versione, PBZIP2, che utilizza design multi thread sfruttando C++ e pthreads. Il codice è generico abbastanza per essere compilato su tutti i compilatori C++ che supportano pthreads. Il codice è possibile trovarlo a (8).

Il programma pbzip2 funziona anche suddividendo i dati in blocchi di uguali dimensioni, configurabili dall'utente. PBZIP2 legge un intero blocco alla volta, aumentando leggermente le prestazioni, anche con un singolo processore. L'utente può anche specificare il numero di processori per pbzip2 da utilizzare durante la compressione. Esso supporta la compressione di singoli file proprio come fa bzip2. Se sono necessari più file da comprimere insieme, i file devono essere compattati, ad esempio con TAR, per creare un unico archivio; il file .tar verrà poi processato da PBZIP2. PBZIP2 utilizza una FIFO, e la taglia della coda è fissata al numero di processori messi a disposizione. Questo crea un buon equilibrio tra velocità e quantità di memoria richiesta. Per ogni processore a disposizione di PBZIP, l'algoritmo legge un blocco ed inserisce un puntatore a questi dati all'interno della coda FIFO. La mutua esclusione (mutex) viene utilizzata per evitare che più thread o processi accedano alla memoria contemporaneamente. Appena la coda viene popolata, i thread possono rimuovere i blocchi dalla coda e processare i dati con B&W. Terminata l'elaborazione, la memoria del blocco originale è rilasciata e il puntatore al blocco compresso, insieme al numero del blocco, viene memorizzato in una tabella di visione globale. Una volta che lo spazio libero nella coda è disponibile, altri blocchi di

dati verranno letti dal file e aggiunte alla coda. Questo processo continua fino a quando tutti i blocchi sono stati letti, a quel punto una variabile globale è impostata per notificare che tutti i blocchi sono stati letti. I thread continueranno ad elaborare i blocchi fino a svuotare la coda. PBZIP2 termina quando i thread terminano la compressione di tutti i blocchi ed il loro ordinamento nell'ordine corretto (Vedi Figura 2).

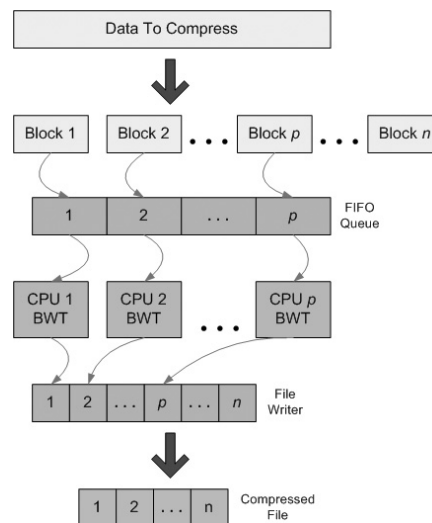


Figura 2 Algoritmo parallelo PBZIP2

Visto che la divisione dei dati iniziali comporta minime comunicazioni tra processi, PBZIP2 impatta minimamente per quanto riguarda tempi di overhead.

8.5 Risultati PBZIP2

PBZIP2 è stato compilato su diverse piattaforme parallele con compilatore gcc e con libreria libzip2 1.0.2. Il file di input usato per testare le performance di BZIP2 e PBZIP2 era un database di 1.83GB non compresso.

Tutti gli esperimenti sono stati misurati in secondi. Questi tempi includono la lettura dei file di input e la scrittura nei file di output. Ogni esperimento è stato effettuato tre volte ed è stata registrata la media dei tre risultati. Dal momento

che bzip2 non sfrutta più processori, il tempo di esecuzione è lo stesso indipendentemente dal numero di processori che il sistema ha. Per motivi di spazio, un solo set di risultati viene presentato. Molteplici esperimenti sono stati eseguiti e mostrano risultati simili che sono disponibili presso <http://compression.ca>.

8.5.1 Athlon-MP 2600+

La piattaforma usata nell'esperimento ha una macchina AMD Athlon-MP 2600+, con due processori da 2.1 GHz, 256KB L2 cache ed 1GB di RAM. Il software è stato compilato con gcc v3.3.1 con cygwin 1.5.10-3 su Windows XP Pro.

PBZIP2 è stato eseguito una volta con lo switch -p1 con un processore, ed è stato testato di nuovo con lo switch -p2 a due processori.

Il programma BZIP2 stabilisce una baseline per le prestazioni. L'esecuzione di PBZIP2 su un processore si è mostrato leggermente più veloce di BZIP2. Questo probabilmente può essere attribuito al fatto che PBZIP2 legge grandi blocchi di dati rispetto a BZIP2, quindi riduce il numero di letture. È stato poi osservato che quando si esegue PBZIP2 su due processori si raggiunge un aumento di velocità del 87.9% come mostrato nel grafico di Figura 3

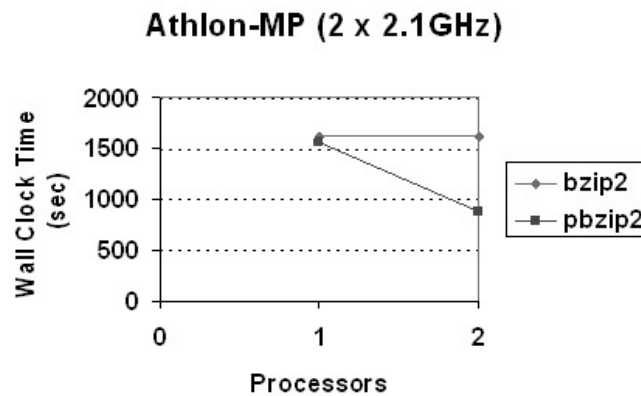


Figura 3 AMD Athlon-MP 2600+, con due processori da 2.1 GHz, 256KB L2 cache ed 1GB di RAM

8.5.2 Itanium 2

La piattaforma usata in questo esperimento è una macchina Intel Itanium2, con due processori da 900 MHz, 1.5MB L3 cache ed 4GB di RAM. Il software è stato compilato con gcc v2.96 ed eseguito su RedHat Linux 7.2 con kernel 2.4.19-4 SMP.

PBZIP2 è stato eseguito una volta con lo switch `-p1` con un processore, ed è stato testato di nuovo con lo switch `-p2` a due processori.

Il programma BZIP2 stabilisce una baseline per le prestazioni. L'esecuzione di PBZIP2 su un processore si è mostrato leggermente più veloce di BZIP2. Questo probabilmente può essere attribuito al fatto che PBZIP2 legge grandi blocchi di dati rispetto a BZIP2, quindi riduce il numero di letture. È stato poi osservato che quando si esegue PBZIP2 su due processori si raggiunge un aumento di velocità del 96.9% come mostrato nel grafico di Figura 4

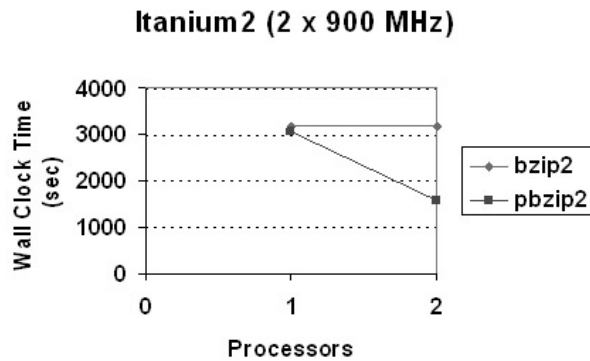


Figura 4 Intel Itanium2, con due processori da 900 MHz, 1.5MB L3 cache ed 4GB di RAM

8.5.3 SunFire 6800

La piattaforma usata in questo esperimento è una macchina SunFire 6800, con ventiquattro processori UltraSPARC-III da 1.05 GHz, 8MB L2 cache ed 96GB di RAM. Il software è stato compilato con gcc v3.2.3 ed eseguito su SunOS 5.9.

PBZIP2 è stato eseguito una volta con lo switch `-p1` con un processore, ed è stato testato di nuovo con gli appropriati comandi di linea a 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 e 22 processori. Visto che la macchina è stata utilizzata da diversi ricercatori, non è stato possibile avere accesso esclusivo, quindi a più di 22 processori.

Il programma BZIP2 stabilisce una baseline per le prestazioni. L'esecuzione di PBZIP2 su un processore ha mostrato approssimativamente la stessa velocità di BZIP2. È stato poi osservato che quando si esegue PBZIP2 su processori multipli, lo speedup vicino all'ottimo è stato raggiunto. Usando 18 processori, PBZIP2 raggiunge il 98% dello speedup ottimale, con 22 processori raggiunge il 97.4 dello speedup ottimale.

Nella Figura 5 possiamo osservare un grafico contenente i dati ottenuti sulla macchina SunFire 6800

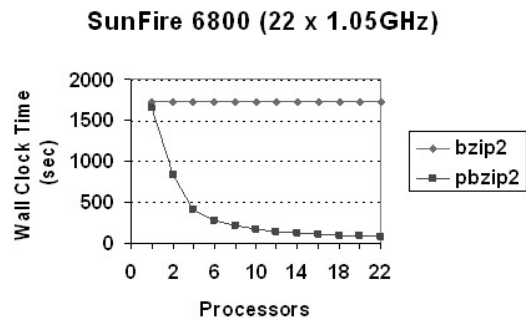


Figura 5 SunFire 6800, con 24 processori UltraSPARC-III da 1.05 GHz, 8MB L2 cache ed 96GB di RAM

Nella Figura 6 viene invece mostrato lo speedup dei relativi processori

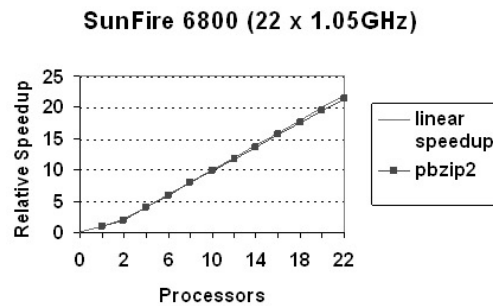


Figura 6 Speedup SunFire 6800, con 24 processori UltraSPARC-III da 1.05 GHz, 8MB L2 cache ed 96GB di RAM

9 Conclusioni

In questo studio sono stati affrontati diversi algoritmi. In particolare ci siamo soffermati sulla trasformata di B&W e abbiamo stilato diversi metodi per parallelizzare tale algoritmo; Mi sono soffermato sul metodo che permette l'esecuzione simultanea dei blocchi, valutando le performance su diverse architetture parallele a memoria condivisa. I risultati mostrano uno speedup prossimo all'ottimo con l'utilizzo dell'algoritmo PBZIP2 su sistemi con processori multipli. Vengono ridotti infatti significativamente i tempi di compressione per grandi quantità di dati, rimanendo completamente compatibile con la versione sequenziale di BZIP2.

Sviluppi futuri potrebbero essere rivolti all'implementazione di PBZIP2 su architetture di cluster a memoria distribuita ed approfondire i metodi di ordinamento parallelo per questi sistemi.

10 Bibliografia

1. **Marconi, Chiara.** *Tecniche di compressione senza perdita per dati unidimensionali e bidimensionali.* Bologna, 2012.
2. **Codice Morse.** *Wikipedia.* [Online]
https://it.wikipedia.org/wiki/Codice_Morse.
3. **Wheeler, Michael Burrows and David J.** *A Block-Sorting Lossless Data.* Digital System Research Center, 1994.
4. *Studio della trasformata di Burrows e Wheeler.* 2005-2006.
5. [Online] <http://www.bzip.org/> .
6. **Iannone, Annalisa.** *L'applicazione della Trasformata di Burrows & Wheeler nell'algoritmo di compressione bzip2: una sperimentazione su dati bidimensionali.* Fisciano, 2007.
7. **Gilchrist, Jeff.** *PARALLEL DATA COMPRESSION WITH BZIP2.* Ottawa, Canada : Elytra Enterprises Inc.
8. **Redhat.** *source.redhat.* [Online] [Riportato: 14 06 2016.]
<http://sources.redhat.com/bzip2/>.