PDOCM : Fast Text Compression on MasPar Machine

PDOCM : MasPar머쉰상의 새로운 압축기법과 빠른 텍스트 축약

Yong Sik Min*
민 용 식*

Abstract

Due to rapid progress in data communications, we are able to acquire the information we need with ease. One means of achieving this is a parallel machine such as the MasPar. Although the parallel machine makes it possible to receive/transmit enormous quantities of data, because of the increasing volume of information that must be processed, it is necessary to transmit only a minimal amount of data bits.

This paper suggests a new coding method for the parallel machine, which compresses the data by reducing redundancy. Parallel Dynamic Octal Compact Mapping (PDOCM) compresses at least 1 byte per word, compared with other coding techniques, and achieves a 54.188-fold speedup with 64 processors to transmit 10 million characters.

요 약

본 논문은 redundancy를 제거함으로 해서 데이타의 축약을 할 수 있는 새로운 방법론 즉, 병렬 컴퓨터인 MasPar 머쉰에 적합한 새로운 데이타 구조를 제시하고자 하는데 그 주된 목적이 있다. 이것을 실제로 구현한 결과, 본 논문에 제시된 방법인 PDOCM(Parallel Dynamic Octal Compact Mapping)은 기존의 방법중 가장 효율이 좋은 것으로 나타난 Huffman 코드와 비교할때는 평균적으로 30%정도, bit-mapping방법과 비교할때는 평균적으로 40% 정도의 우수성을 보였다. 그리고 10백만개의 영문자를 이용해서 MasPar 기계에서 64개의 프로세서를 이용하여 구현시킨 결과 54.188의 가속화율을 얻으므로서 우수한 방법임을 알 수가 있었다.

I. Introduction

Although we have so far been able to handle with ease all the information in our society, it is becoming increasingly necessary to transmit only a minimal amount of data bits because of the sheer volume of information being transmitted.

In developing this paper, we considered several data-compression methods, but we elaborate here on two of them : the bit-mapping technique and the binary compact code. The bit-mapping technique produces a great compression effect when there are many spaces in the source symbol stream. The technique involves these procedures. In the source symbol stream $S = \{s_1, s_2, \ldots, s_q\}$,

*호서대학교 전자계산학과
접수일자 : 1995년 1월 9일

after determining the total number of all symbols, we create a one-byte bit-map zone part. If the source symbol is a blank, the bit-map zone corresponds to 0; otherwise, it corresponds to 1. We then create the EBCDIC with the right side of the bit-map zone. We proceed with this method until the last symbol. In the source symbol stream $S=\{s_1, s_2, \ldots, s_q\}$, if s_q is not the last 8th symbol, (i.e., $q \neq 8 \cdot i$ where $i \geq 1$ and i is an integer), the remaining part of the bit-map zone is regarded as blanks. The compression rate of this method is explained in [2].

The bit-mapping method deals with fixed-length codes. The second method, called compact binary code [2], deals with the variable code lengths of different symbols. In this method, the symbol that occurs infrequently in the text represents a long code length whereas the symbol that occurs frequently represents a short code length. The binary compact code just described compresses the data to 58% of its original length. This method has a major disadvantages, however. If the source symbols have many blanks, or there are many symbols in the text that occur frequently, this method requires more code lengths in total. Despite this shortcoming, the binary compact code technique is the best [2, 5, 6].

This paper suggests a new data-compression technique that improves upon the binary compact code method by overcoming the disadvantage of longer code lengths. Our improved method, called PDOCM was implemented on a parallel machine such as the MasPar. In practice, PDOCM achieved a 54. 188-fold speedup with 64 processors to transmit 10 million source symbols.

The rest of this paper is organized as follows. Secion II describes the sequential method of data compression, and section III describes the parallel method. Section IV includes the results of PDOCM implementation, and section V presents our conclusions.

II. Sequential Method

Given non-negative weights $(w_1, w_2, \ldots, w_n)$, we can use the well-known algorithm of the Huffman code to construct a binary tree with n external nodes and n-1 internal nodes, where the external nodes are labeled with weights $(w_1, w_2, \ldots, w_n)$ in increasing/decreasing order. Huffman's tree has the minimum value of $w_1 l_1 + \ldots + w_n l_n$ over all such binary trees, where l_j is the level at which w_j occurs in the tree. Binary trees with n external nodes are in one-to-one correspondence with sets of n strings or {0, 1} [1]. For example, the binary tree in Fig. 1 corresponds to the minimal code {0, 10, 110, 111}. In Fig. 1, Huffman's method combines the two smallest weights w_i and w_j (the characters that have the lowest probabilities to appear), replaces them by their sum $w_i + w_j$, and repeats this process until only one weight is left. In this situation (Fig. 1), there is no way to distinguish weight 6 associated with symbol A from weight 6 associated with symbol C and D. As a consequence, this procedure may form two different trees (Fig. 1 and Fig. 2), depending on where the weight 6 that is associated with '2+4=6' is placed. Both trees are optimum for the given weights, since

$$6X1+5X2+4X3+2X3=2X4+2X2+5X2+6X2.$$

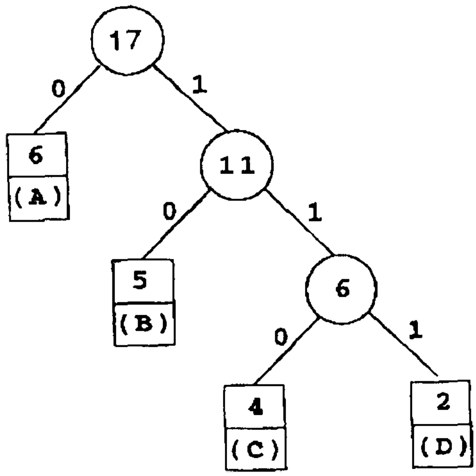


Fig. 1. Huffman Method

We call this method a dynamic compact code [1]. This procedure reforms Huffman's tree dynamically, in order to reduce the height of the tree. If the weight 6 associated with A increases to 7, Fig. 1 is better : but, if weight 2 associated with D increases to 3, Fig. 2 is better. In the average case, Fig. 2 is better even though it has some disadvantages [1].

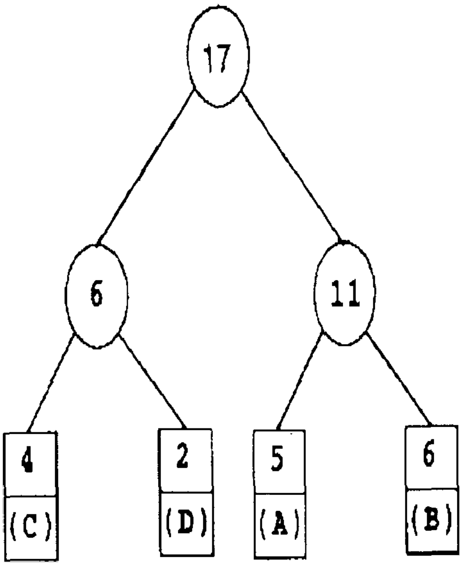


Fig. 2. Dynamic tree

To construct Huffman's tree, we must go through several steps. First, we investigate the probabilities of each symbol in the context in order for it to correspond to the character. This exercise proves that the statistical data of 1 million characters from arbitary text is suitable for this purpose. As a result, we know that one word has 8 symbols ; that is, we need at least 3 bits ($2^3=8$) to represent one word. A new data structure is thus formed, which is supported in dynamic octal-compact mapping as follows (see Fig. 3).

The data structure of PDOCM consists of two parts. First, there is a zone part which has two subparts. One is a check bit (1 bit), and the other

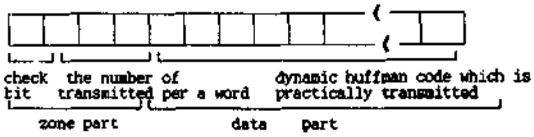


Fig. 3. The data structure of PDOCM

consists of 3 data bits representing one word. If a check bit is 0 (i.e., its corresponding word is a blank), a rest of 3 bits in its zone part represents the number of blanks to be transmitted. This method is not considered in the dynamic compact code. If the symbol is not a blank, in the first part of the new data structure, 1 is placed to the check bit and the number of the symbol is placed to the data bits. The data part, which is the second part of the new data structure, represents the number of symbols that are practically transmitted. For example, text data "AB□□□□CD"(where □represents a blank symbol) is represented by the dynamic octal-compact method as follows :

1010	10	11	0100	1010	00	01
zone	A	B	zone	zone	C	D

This structure combines the space-compression advantage of the bit-mapping method with, the advantage of describing the variable code length of each symbol (binary compact code), and the advantage of eliminating spaces (the octal-compact mapping method). Its practical code is equal to a dynamic compact code except that PDCOM excludes spaces and introduces a new data structure.

Given the data structure above, it is not difficult to design pseudo-algorithms of the binary code tree as follows.

procedure binarycodetree(float p)
/*the source S with symbols $\{S_1, S_2, \ldots, S_q\}$ and symbol probabilities $\{P_1, P_2, \ldots, P_q\}$*/
begin
(1) Let the symbols(except blank symbol) be ordered so that $P_1 \geq P_2 \geq \ldots \geq P_q$

(2) We assigned the words 0 and 1 to the last sequence
(3) Combine the last two symbols of S into one symbol
(4) Search back from the last sequence to the original sequence through the reduced sources
(5) Repeat (2)-(4) until there left only two symbols codes
end

The total time for the procedure binarycodetree requires O(n log n) to construct the binary code tree. Step 1 requires O(n log n), which is the time complexity of the best sorting algorithm such as Quicksort or Mergesort [3]. Step 2 requires O(log n), which constructs the tree. Step 3 takes a constant time ; and step 4 takes O(l), where l is the level of the tree.

We construct a dynamic Huffman tree from the binary code tree as follows [1].

procedure dynamictree
begin
(1) Represent a binary code tree with weights in each symbol
(2) Maintain a linear list of symbols, in nodecreasing order by weight
(3) Find the last symbol in this linear list that has the same weight as a given symbol
(4) Interchange two subtrees of the same weights
(5) Increase the weight of the last node in some block by unity
(6) Represent the correspondance between letters and external symbols
end

This procedure requires O(n) ; that is, a binary code tree is constructed by steps 1 and 2 in the same manner as the above procedure binarycodetree. Step 3 takes O(log n), which traverses the tree. Steps 4 and 5 require O(l), which updates an element at level l of the tree and step 6 requires O(n). Together, the steps require an overall O(n) time.

Ⅲ. Parallel Dynamic Octal-Compact Mapping

In this section, we describe the improved parallel method referred to in the last section. The Parallel Dynamic Octal-Compact Mapping method (PDOCM) has three phases that compress the source symbols. In the first phase, the binary code tree is constructed from raw source symbols, each of which have a probability. Before constructing of the binary code tree, one has to consider the number of processors that are going to be used on the machine. In this situation, there are three cases. P, the number of processors, is less than, equal to, or greater than the number of symbols at level l, which contains either all the symbols or part of the symbols.

If P is greater than or equal to the number of symbols at level l, then each processor at level i is connected to a single parent processor at level i-1 and to each of its two child processors at level i+1, except for the root processor at level 0 (which has no parent) and the leaf processor at level d-1 (which has no children). If P is less than the number of symbols at level l, then each processor at level i can be connected to either the same or a different parent processor. Afterward, we use the processor to construct the binary code tree described in the previous section.

Let us consider step 1 in the procedure of the binarycodetree. In that case, we use the parallel algorithms to sort the sequence $S=\{x_1, x_2, \ldots, x_n\}$ of distinct probabilities in increasing order [3]. This method requires n^{1-e} processors, where $0<e<1$ runs in $O(n^e \log n)$ time. In steps 2 through 5, the code is produced using the same method as the parallel tree construction. It requires O(log n), which supports the code. A pseudo-algorithm of this method is as follows.

procedure firststepinparallel
begin

(1) Parallel quicksort using each probability
(2) for(traverse from the root to leaves) do in parallel
　(2.1) We assigned each processor's word 0 or 1
　(2.2) Search previous two symbols of S which were combined as one symbol
allfor
end

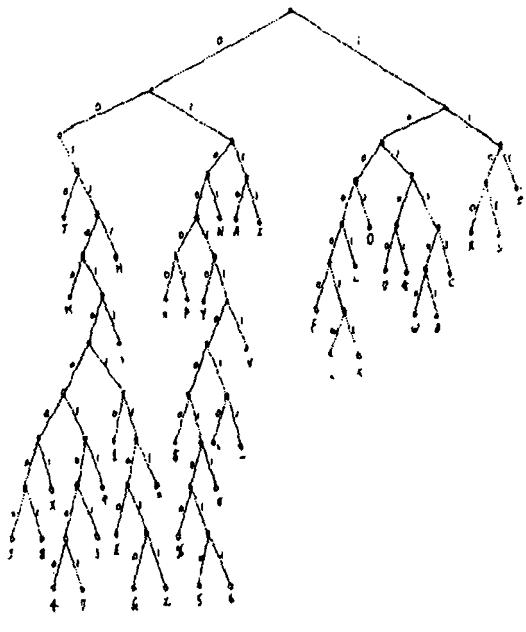
The second phase is analogous to the first. The second step only requires exchanging the two subtrees of the same weights different processors have. It is quite simple to implement. This phase requires O(1) to update an element at level l of the tree.

In the third phase, we encode or decode the text data from the dynamic octal-compact mapping code. In this phase, each processor reads the text data to determine whether the character read is a blank symbol or not. If the character is a blank, the check bit in the zone part is set to 0. The following code would not be set since there is no code for a blank symbol. If the check bit is 1, however, we set the following part as a dynamic compact code of the character. This procedure processes by the word which includes, at most, 8 symbols. If a word exceeds 8 symbols, it is split by 8 symbols. If the last word has fewer than 8 symbols, however, we process it with words that have at least 8 symbols. This pseudo-algorithm, which is implemented by O(n/p) time in each processor, is as follows.

```
procedure thirdstepinparallel
/* n: the size of text data.
   p: the number of processors */
begin
   for(i=p*⌊n/p⌋ to ((p+1)*⌊n/p⌋)-1) do in parallel
        p-read(one character in text data)
        if(the symbol read is a blank) then
            repeat
               character count; p-read(one character);
            until (symbol read is not blank);
          else
            repeat
               character count; p-read(one character);
            until ( symbol read is a blank);
        endif
        if(the number of counts exceed 8 ) then
            zi = the number of counts div 8 + 1
        coustruct the zone part which has a value of i or zi
        satisfying zi's value
   allfor
end
```

Ⅳ. Experimental Results

To implement the PDOCM on the MasPar, we tested randomly generated text sentences with various distributions. To find the probabilities of each symbol, we extracted 10 million characters from a random text. The probability of each symbol was computed to create the statistical data used in the previous section. The result of the dynamic compact tree is shown in Fig. 4. The size of the sentences to be compressed ranged from 0.01 million to 10 million symbols. Experiments were conducted using each of 1, 2, 4, 8, 16, 32, and 64 processors on the MasPar machine. Each data point presented in this section was obtained from the average of one program's execution. Each processed 10 million characters.



(b) dynamic compact tree

Fig. 4. Dynmic Huffman tree

We have developed a progran that provides the optinal seguential DOCM. The time was used on one processor. It needs the speedup which eraluates a new data-compression method for some problems. The speedup[1] is defined as the time elapsed from the moment the algorithn stats to the moment it termioates. It is reasonable to assume that the time of data compression using seguetial DOCM is one PE :

$t_{pe}(n) = (n \log n)$

where c is a constant *independent of sizen.* seguential times for lists of more than 0.2 million elements were calculated using the fornula :

$$t_{pe}(n) = \frac{n \log n}{100{,}000 \log 100{,}000} \cdot t_{pe}(100{,}000),$$

where 0.02 million $\leq n \leq$ 10 million and t_{pe}(100, 000) = 0.62 seconds. Note that if one uses this *formula to compute* $t_{pe}(200{,}000)$, the result is almost a perfect match with the corresponding experimental time.

Table 1 shows the time required to compress the data using PDOCM, and Fig. 5 plots the speedups achieved. As the problem size increases, task granularity increases. Offsetting the overheads of the algorithms results in better speedup. Compression of 10 million text data with 64 processors yielded a 54.188-fold speedup, compared with what can be achieved with only one processor. This method was implemented in each processor's local memory. Global memory was used to communicate the code.

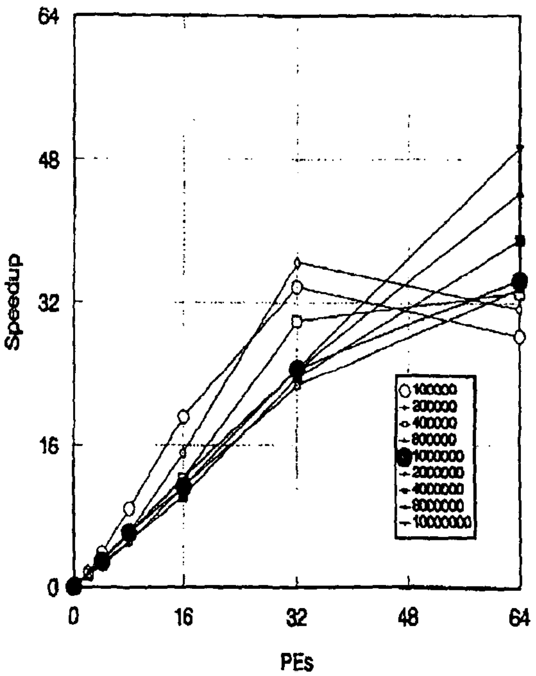


Fig. 5. Speedup of PDOCM

Table 1. Time to compress using PDOCM(unit:second)

n PE	1	2	4	8	16	32	64
100,000	0.62	0.364	0.139	0.056	0.025	0.0214	0.0196
200,000	1.23	0.645	0.306	0.140	0.058	0.0438	0.0398
400,000	2.35	1.347	0.641	0.307	0.142	0.108	0.0765
800,000	4.59	2.797	1.663	0.742	0.305	0.1992	0.1389
1,000,000	5.69	–	1.610	0.791	0.384	0.2578	0.1689
2,000,000	11.15	–	–	1.607	0.923	0.379	0.3068
4,000,000	22.38	–	–	–	1.736	0.795	0.6579
8,000,000	44.62	–	–	–	–	1.760	1.294
10,000,000	56.74	–	–	–	–	–	1.2841

	PROBABILITY	CODE		PROBABILITY	CODE
E	.09684	111	.	7.61E-03	1000010
T	.07907	0010	K	7.61E-03	1000011
N	.06833	0101	1	2.73E-03	001101010
A	.0654	0110	$	2.E-03	010011000
I	.05974	0111	'	1.95E-03	010011000
O	.0574	1001	–	1.95E-03	010011011
R	.05154	1100	X	1.56E-03	0011010001
S	.05154	1101	9	1.37E-03	0011010011
H	.03631	00111	"	1.17E-03	0011010111
L	.02714	10001	0	9.8E-04	0100110011
D	.02675	10100	J	9.8E-04	00110100000
G	.02655	10111	8	7.8E-04	00110100001
C	.02578	001100	3	5.9E-04	00110100101
M	.02011	010000	Z	5.9E-04	00110101100
U	.01952	010001	%	5.8E-04	01001100100
P	.01679	010010	4	3.9E-04	001101001000
Y	.0164	100000	7	3.9E-04	001101001001
F	.01542	101100	Q	3.9E-04	001101011010
W	.01308	101100	2	2E-04	001101011011
8 B	.01289	001101	5	2E-04	010011001010
,	8.39E-03	0011011	6	2E-04	010011001011
V	7.81E-03	0100111			

(a) the probability of each symbol

V. Conclusion

Table 2 shows the entropy for each of the techniques. In practice, with 10 million data on 64 processors, we used 4.08 bits per symbol, whereas the OCM method [5] uses 4.28 bits and the *Huffman code uses* 4.99 bits. Processing a word of 8 symbols (that is, the average length of a word), we show that the PDOCM method compresses at least 1 byte (in the average-case). In the worst-case, the bit-mapping method compresses 3 bytes.

In conclusions, PDOCM reduces redundancy so that we can send and receive more data with a minimal number of bits. Error-detection problems on the transmission line were not considered in this research.

Table 2. Comparison with other methods(unit:bytes)

method	worst-case	best-case	average-cdase
Bit-mapping	9	1	5
Huffman Code	12	3	4.5
OCM	13	1	3.5
our Method	12	1	3.0

References

1. Kunth, Donald E., "Dynamic Huffman Coding,"

Journal of Algorithm, vol. 6, no. 2, pp. 163-180, June, 1985
2. S. Roman, Coding and Information Theory, Springer-Verlag, 1992
3. Akl, Selim G. , Parallel Sorting Algorihtms, Academic press, 1985
4. Bookstein, A. and Klein, S. T., "Is Huffman Coding Dead," Proceedings of Data Compression, IEEE, p. 464, 1993
5. Kim, K. T. and Min, Y. S., "A Study on the Composition of Compact Code using OCM," Journal of KCI, vol. 9, no. 3, pp. 103-107, 1984
6. Kim, K. T. and Min, Y. S., "A Study on an Efficient Coding of Hanguel," Journal of KCI, vol. 14, no. 6, pp. 533-641, 1987

▲민 용 식(Yong Sik Min)

1981년 2월 : Dept. of Computer Science, Kwangwoon Univ.(B.S.)
1984년 2월 : Dept. of Computer Science, Kwangwoon Univ.(M.S.)
1991년 2월 : Dept. of Computer Science, Kwangwoon Univ.(Ph.D)
1984년 3월~1987년 2월 : Full-time lecturer, Songwon Junior College Dept. of Computer Science
1987년 3월 : present : Associate Professor, Hoseo Univ. Dept. of Computer Science
1993년 8월~1994년 8월 : Visiting Professor, Louisiana State Univ. Dept. of Computer Science