Sliding-Window Compression on the Hypercube*

Charalampos Konstantopoulos, Andreas Svolos, and Christos Kaklamanis

Computer Engineering and Informatics Department, University of Patras and Computer Technology Institute, 11 Aktaiou and Poulopoulou, GR 118 51, Athens, Greece
{konstant,svolos,kakl}@cti.gr

Abstract. Dictionary compression belongs to the class of lossless compression methods and is mainly used for compressing text files [1, 2, 3]. In this paper, we present a parallel algorithm for one of these coding methods, namely the LZ77 coding algorithm also known as a sliding-window coding algorithm. Although there exist PRAM algorithms [4, 5] for various dictionary compression methods, their rather irregular structure has discouraged their implementation on practical interconnection networks such as the mesh and hypercube. However in the case of LZ77 coding, we show how to exploit the specific properties of the algorithm in order to achieve an efficient implementation on the hypercube.

1 Introduction

The main idea of the LZ77 algorithm [6] is that strings in the text are replaced with pointers to a previous occurrences of these strings in the text. Specifically, let $x[0 \cdots N-1]$ be the input string. Assume also the prefix $x[0 \cdots i-1]$ has been compressed so far. The dictionary at this moment consists of all the substrings $x[i-j \cdots k]$ where $j \in [1, \cdots, M]$, $k \in [i-j, \cdots, i-j+F-1]$ and M, F are two parameters of the algorithm. The next step is to find the longest prefix of $x[i \cdots N]$ which matches an entry of this dictionary. If this prefix is of length r and $x[i-q \cdots i-q+r-1]$ is the matching string in the dictionary ($q \in [1 \cdots M]$), then we replace the prefix $x[i \cdots i+r-1]$ with the pointer (q, r) and we proceed to the position $i+r$ of the input string. The string up to the position $i+r-1$ has now been compressed. Notice that if the character x_i does not occur within the last M preceding characters, then we cannot find any matching prefix at position i. In this case, we replace the character x_i with the pointer $(x_i, 1)$ [7].

In this paper, we present an efficient implementation of the LZ77 (sliding window coding) algorithm on the hypercube network. A basic assumption of our implementation is that the employed multiprocessor is fine grained with only limited local memory per processing element (PE). Taking advantage of the properties of sliding-window coding algorithms we show that this kind of algorithms can be efficiently implemented on the hypercube network by using only a small number of fast communication primitives.

* Work is supported in part by the General Secretariat of Research and Technology of Greece under Project ΠΕΝΕΔ 95 ΕΔ 1623.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 835–838, 2000.

2 LZ77 Coding on the Hypercube

In the following, we present our parallel algorithm for sliding-window compression on a hypercube-based multiprocessor. The input string $x[0 \cdots N-1]$ is distributed one character per PE, i.e character x_i is initially stored in PE i where $i = 0, \cdots, N-1$. Our basic goal is to present an efficient parallel algorithm using the least amount of memory at each PE. For convenience, we also assume that $N = 2^n$.

In LZ77 coding algorithms the size M of the sliding window is finite and usually much smaller than the total length of the input string. In practice, a window of moderate size, typically $M \leq 8192$, can work well for a variety of texts [1]. Due to this small value, we can perform the required string matching operations using only simple search techniques without increasing the computational overhead unduly.

The parallel algorithm consists of two phases.

First phase. In this phase, for each position of the input string we find the longest substring starting at this position which also matches an entry in the adaptive dictionary. Specifically, for each position i of the input string we determine the longest common prefix of string $x[i \cdots i+F-1]$ with the strings $x[i-m_i \cdots i-m_i+F-1]$ where $m_i \in [1, \cdots M]$. This can be carried out in $O(M \log N)$ time by executing a series of M shift and prefix sum operations [8, 9]. If each PE can communicate with all its neighbors at the same time (all-port capability), the above complexity can be reduced to $O(M + \log N)$ time. This can be achieved by overlapping in time the execution of successive shift and prefix sum operations.

Second phase. It can be easily seen that the sequence of pointers obtained from the sequential LZ77 coding algorithm is a subset of the pointers derived from the first phase of the parallel algorithm. The goal in the second phase is to determine which of these pointers will be finally included in the compressed file.

This can be easily done in two stages by using the following simple technique. Let the pointer (m_i, l_i) of PE i point to the character $i + l_i$ of the input string, that is the first character after the longest common prefix at position i. The first pointer in the compressed file is definitely that of position 0, namely (m_0, l_0). The second pointer is that of position l_0, (m_{l_0}, l_{l_0}). The third pointer is that of position $l_0 + l_{l_0}$ and so on. Clearly, if we start from position 0 and follow the pointers defined above, we will eventually visit all the breakpoints[1] defined by the sequential LZ77 parsing. This sequential traversal of pointers can be performed in parallel in $O(\log N)$ steps by using the well-known pointer jumping technique [10].

As this is important, each PE saves the addresses of PEs it visits at each pointer jumping step. However, only $O(\log N)$ local memory at each PE is needed for this information, since each PE visits at most $\lceil \log N \rceil$ PEs during these steps.

We prove the following lemma:

[1] Breakpoints are the positions in the input text at which the parsing process splits the text.

Lemma 1. *If p_i, p_j are the pointers of PEs i, j respectively at a pointer jumping step , then* $\forall\ i,j\ i \leq j \Rightarrow p_i \leq p_j$.

Proof. We prove the lemma by induction on the number of elapsing pointer jumping steps. In the first step, pointer p_i is equal to $i+l_i$ and thus the inequality $p_i \leq p_j$ can be written as $i + l_i \leq j + l_j$.

We distinguish two cases. If $i + l_i \leq j$, it is obvious that the above inequality holds. Now consider the case $i \leq j < i + l_i$. Clearly, character x_j belongs to the longest common prefix of position i and thus the longest common prefix at position j cannot be smaller than $i + l_i - j$ characters, that is $l_j \geq i + l_i - j \Rightarrow i + l_i \leq j + l_j$. Thus we have proved the lemma for the first step.

Assume now that the statement $\forall\ i,j\ i \leq j \Rightarrow p_i \leq p_j$ holds for all the elapsing pointer jumping steps up to the step k. We will prove that it also holds for the step $k + 1$. Suppose that at step k position i points to position a_i and position a_i in turn points to position b_i. After step $k + 1$, position i will point to position b_i. We can prove that $\forall\ i,j\ i \leq j \Rightarrow b_i \leq b_j$. If $j \geq b_i$, the proof is obvious. Consider now the case $i \leq j < b_i$. From the induction hypothesis, it follows that $a_i \leq a_j$. If $b_i > b_j$ then the statement $a_i \leq a_j \Rightarrow b_i \leq b_j$ does not hold, thereby contradicting the induction hypothesis. Thus the lemma is true for the step $k + 1$ as well. □

Now it is clear that each pointer jumping step can be performed using monotone routing [8, 9] in place of expensive sorting steps. Each step takes $O(\log N)$ time and thus the $O(\log N)$ pointer jumping steps of the first stage can be performed in $O(\log^2 N)$ time overall.

After the above pointer jumping steps, the next stage is to "mark" the positions of the input string that are breakpoints in the sequential LZ77 parsing. PE 0 knows that its position, position 0, is a breakpoint. It also knows $O(\log N)$ positions which are certainly breakpoints as well. These positions correspond to the PEs which it visited during the pointer jumping steps. Let $i_1, i_2, \cdots, i_{O(\log N)}$ be the addresses of these processors. PE 0 should notify them that hold breakpoints. After a PE, say PE i_k, receives the notification, it should in turn notify those PEs in the interval $[i_k+1 \cdots i_{k+1}-1]$ which it has visited during the pointer jumping steps. This process proceeds recursively and finally all the breakpoints of the sequential LZ77 parsing are marked. It can be easily seen that this recursive marking can be carried out by reversing the steps of the first stage. At each step, each PE that has already received a notification packet sends such a packet to the PE that it had visited at the corresponding pointer jumping step of the first stage. The communication complexity of each step is only $O(\log N)$ since we can use monotone routing again.

We have described the second phase of the parallel LZ77 coding algorithm. The complexity $O(\log^2 N)$ of this phase is mainly due to the fact that pointer jumping steps are performed along a string of length N. However, it is possible to limit pointer jumping steps along shorter segments of the input string, thereby largely decreasing the complexity of the second phase. Let us see how this can be done. After the end of the first phase, each PE i holds the pointer (m_i, l_i) in

its local memory. Then, using a prefix sum operation, each PE i estimates the expression $max_i = max(e_0, e_1, \cdots, e_{i-1})$[2] where $e_i = l_i + i$ ($O(\log N)$ delay). One can easily notice that if $i \geq max_i$, pointer (m_i, l_i) is definitely included in the compressed file. The corresponding position i is called cut-point; cut-point is a position in the input string for which we are certain in advance that it is one of the breakpoints of the LZ77 parsing [1]. Clearly, cut-points split the input string into non-overlapping substrings and thus we can execute the second phase of the parallel LZ77 coding algorithm independently for each substring. Due to the shorter length of these substrings, the complexity of the second phase is largely decreased. Specifically, if L is the length of the longest substring between two successive cut-points, the second phase can be executed now in $O(\lceil \log L \rceil \cdot \log N)$ time. In practice, $L \simeq 100$ since cut-points almost always occur well under 100 characters apart [1].

3 Conclusions

We presented an efficient parallel algorithm for LZ77 coding on the hypercube network. General simulations of PRAM dictionary compression algorithms on the hypercube surely leads to solutions with high communication overhead. However, by carefully examining the way LZ77 parsing splits the text into phrases, we managed to considerably lower the communication overhead. In doing so, we used only a small set of communication primitives which can be efficiently executed on the hypercube network. In addition, we further enhanced the performance by exploiting known facts from the text compression (frequent cut-points).

References

[1] T. C. Bell, J. G. Cleary, I. H. Witten, Text Compression Prentice Hall Advanced Reference Series Computer Science, (1990).
[2] K. Sayood, Introduction to Data Compression Morgan Kaufmann Publishers Inc. (1996).
[3] J. A. Storer, Data Compression Methods and Theory, Computer Science Press, Rockville, MD (1988).
[4] L. M. Stauffer, D. S. Hirschberg, Dictionary Compression on the PRAM, Parallel Processing Letters 7 (3) 1997 297–308.
[5] M. Farach, S. Muthukrishnan, Optimal Parallel Dictionary Matching and Compression (Extended Abstract), in Proc. SPAA 1995, 244–253.
[6] J. Ziv, A. Lempel, A Universal Algorithm for Sequential Data Compression, IEEE Trans. Inf. Theory 23 (3) 1977 337–343.
[7] T. C. Bell, Better OPM/L Text Compression, IEEE Trans. Communications COM-34 (12) 1986 1176–1182.
[8] S. Ranka, S. Sahni, Hypercube Algorithms with Applications to Image Processing and Pattern Recognition, Springer Verlag (1990).
[9] T. F. Leighton, Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes, Morgan Kaufmann Publishers, San Mateo, CA (1992).
[10] J. Jàjà, An Introduction to Parallel Algorithms, Addison-Wesley (1992).

[2] We assume $max_0 = 0$.