Z
699
C3
no. 91-44

Parallel Data Compression

Lynn M. Stauffer
University of California, Irvine
Irvine, CA 92717
stauffer@ics.uci.edu

Daniel S. Hirschberg
University of California, Irvine
Irvine, CA 92717
dan@ics.uci.edu

Technical Report 91-44

May 1, 1991

ABSTRACT

Data compression schemes remove data redundancy in communicated and stored data and increase the effective capacities of communication and storage devices. Parallel algorithms and implementations for textual data compression are surveyed. Related concepts from parallel computation and information theory are briefly discussed. Static and dynamic methods for codeword construction and transmission on various models of parallel computation are described. Included are parallel methods which boost system speed by coding data concurrently, and approaches which employ multiple compression techniques to improve compression ratios. Theoretical and empirical comparisons are reported and areas for future research are suggested.

Contents

1. Introduction

Data compression attempts to remove redundancy from data and thereby increases the effective density of transmitted or stored data. Traditionally, there has been a tradeoff between the benefits of employing data compression and the computational overhead required to perform encoding and decoding. Parallelism represents an avenue for increasing the speed (throughput) and performance of a data compression subsystem. Consequently, parallel data compression is suitable for a wider range of applications. The purpose of this paper is to present and analyze parallel data compression methods. For a reference on data compression terminology, see [LH87], [S88A] and [BCW90].

Parallel computing, the process of solving problems on parallel computers, has risen out of the need for higher performance systems. Weather prediction, nuclear reactor monitoring, DNA sequencing and artificial intelligence applications demand time-critical computers that are extremely fast [Q87]. For sequential systems, data compression improves communication speed and storage utilization. In the parallel environment, processor interconnection data-rates and data availability play even more critical roles in system performance. Data compression in these parallel systems motivates the concurrent coding schemes surveyed in this paper.

Data compression has become an essential component of high speed data storage and communication. Performance of distributed computing systems is often restricted by the speed of the communication channel. Compacting messages before transmission increases the effective bandwidth of the communication link. Advances in VLSI continue to expand the practicality of placing sophisticated data compression algorithms implemented in VLSI at each end of every communication channel. These encoding/decoding chips increase the capacity of the interconnection. Other services, such as data processing, which manipulate large volumes of data that must be retrieved from and stored in external storage devices, also benefit from widespread incorporation of data compression schemes. By compressing the data before it is stored and later expanding the stored form, the effective capacity of the storage medium is increased. Data compression provides additional benefits such as increased security and efficiency in search operations on compressed files and reduction in backup and recovery costs in computer systems.

From a practical point of view, speed distinguishes the use of parallelism in data compression. Parallel data compression is appropriate for a wider range of time-critical applications. High-resolution stereoscopic color television broadcasting is one example of an application whose data rate is so critical that the overhead required for compressing data may outweigh the benefits of reducing data redundancy [C90]. In addition to speed, the throughput of many parallel computational models is independent of the size of the model; this has both theoretical and practical ramifications.

The state-of-the-art in software data compression systems is the UNIX[1] *compress* utility which is based on a variation of Lempel-Ziv coding [ZL78] due to Welch [W84]. The UNIX *compress* utility provides compression savings of up to 80% at a relatively high input bandwidth[2] of 30 Kbytes per second on a 1 MIPS machine [TW89]. Higher compression savings achieved by high-order Markov models and improved versions of *compress* operate at limited bandwidths of approximately 10 Kbytes per second on a 1 MIPS machine. Thomborson and Wei describe a high-bandwidth (40 Kbytes per second) systolic text compression system with compression savings ranging from 20% to 70% [TW89]. A number of other implementations achieve good compression at input rates of several hundred million bits per second [HR90, SR90, Z90A, Z90B, Z90C].

Although the practicality and applicability of performing data compression is increased by introducing parallelism to speed up coding, multiprocessing is an

[1] UNIX is a trademark of AT&T Bell Laboratories.
[2] *Bandwidth* is synonymous with *data transfer rate*.

alternative use of concurrency which employs various coding schemes to improve compression effectiveness. At the expense of system time and hardware resources, multiple compression algorithms, operating simultaneously, can improve compression rates. While this use of parallelism may be less practical because of its resource demands, the benefits of compression may outweigh the increased overhead for certain applications. Competitive parallel processing and algorithm pipelining are discussed in Section 5.

This survey of parallel data compression considers *lossless* data compression techniques which operate under the constraint that the decompressed data must be identical to the original data stream. Lossless compression demands that no form of deviation be introduced into the data during encoding or decoding. Text compression, the focus of this survey, is normally restricted to lossless compression. Image processing is an example of an application that can tolerate inconsistencies between the original data and its compressed form. Lossy image compression techniques often subdivide the input into subimages which are compressed and expanded independently in parallel. Errors introduced along the boundaries of the substreams cause deviation from the input; however, normally the decompressed image closely approximates the original. Lipton and Lopresti present a systolic array for string comparisons with applications to lossy data compression [LL85].

It is further assumed that the communication channels and storage devices are *noiseless*. That is, it is assumed that no data inaccuracies are introduced during data transmission. Many techniques are available for error detection and correction but are not included in this survey.

Background concepts in data compression and parallel computation are provided in Section 2. Parallel coding systems based on concurrent manipulation of source data are described in Sections 3 and 4. Aggregate compression systems incorporating collections of compression methods to improve coding effectiveness are discussed in Section 5. Finally, in Section 6, topics for future research and their relationship to known results are examined.

2. Preliminaries

A brief introduction to data compression and parallel processing is provided in this section. The terms and assumptions necessary for a careful evaluation and comparison of parallel data compression methods are presented. For a more detailed discussion of data compression in the sequential setting see [LH87], [S88A] and [BCW90].

2.1. Models of Parallel Computation

The purpose of this section is to introduce some of the concepts, formal models, and performance measures from the area of parallel computation. There are a variety of abstract models of parallel machines that correspond to different system designs. Closest to the physical hardware are VLSI models that focus on technological limits. Other models, slightly removed from the actual implementation, emphasize the importance of processor interconnection organization. Another class of machines is defined by Flynn's Taxonomy which categorizes an architecture by the presence or absence of multiplicity in the instruction and input streams [F66]. Furthest from physical system design is a general-purpose theoretical model, the parallel random access machine (PRAM) in which it is assumed that each processor has random access in unit time to any cell of a global memory. The following discussion on parallel models includes only those models that are bases for the data compression methods presented in this paper. For a more thorough discussion of parallel models, see [M88].

Flynn's classification distinguishes parallel architectures based on the concepts of instruction stream and data stream. An instruction stream is a sequence of instructions executed by a computer and a data stream is the sequence of input data. Single-Instruction, Single-Data (SISD) computers are essentially enhanced sequential computers capable of pipelining the instruction stream. Multiple-Instruction, Multiple-Data (MIMD) computers include multiprocessing systems that have independent processors operating on non-overlapping sequences of input. The most common model is the Single-Instruction, Multiple-Data (SIMD) computer which typically consists of a number of uniform processors, an interconnection network, and an associative memory. The synchronized processing elements of an SIMD computer, also referred to as a *processor array*, simultaneously perform the same operation on different data. Processor arrays can differ in terms of number of processors and method of interprocessor communication.

The principal model of computation considered in the theoretical study of parallel algorithms and the complexity of parallel computation is the parallel random access machine (PRAM) which is in the SIMD classification [FW78, G78]. The PRAM consists of a number of identical general-purpose sequential processors, all of which are connected to a large shared, random access memory. Each processor has a private memory for local computation, but communication between processors is done through information exchange in a global random access memory. It is further assumed that each processor may access any cell in the common memory in constant time. In the PRAM model, each processor is assigned an index and all processors execute the same instruction sequence. However, each processor may perform differently depending on its corresponding index. The PRAM is not a physically realizable model since it is impossible to provide a constant length communication link amongst an arbitrarily large number of processors. Nevertheless, the intention of the PRAM model is to permit the study of parallel computation abstracted away from the issues of interprocessor communication.

There are several variants of the PRAM which differ in their handling of simultaneous reading and writing of the global memory. The weakest of these variants is the Exclusive-Read, Exclusive-Write (EREW) PRAM in which concurrent reads and writes are prohibited. The Concurrent-Read, Exclusive-Write model permits multiple processors to access a common memory location but forbids simultaneous writes. The least restrictive model, the Concurrent-Read, Concurrent-Write (CRCW) PRAM, allows different processors to read from and write to identical positions in the shared memory. CRCW PRAM models are further distinguished by their methods of handling write conflicts. Even though there are a variety of PRAM models, they do not differ widely in computational power.

Although the PRAM provides a useful framework for studying parallel computation, other SIMD models that view a parallel computer as a set of processors interconnected in a fixed pattern more closely resemble actual hardware. These models assume that each processor has its own local memory and that data passes between elements via a communication network. In the *mesh-connected* SIMD model, processors are arranged in a lattice with connections between neighboring processors. *Systolic arrays* are linearly connected SIMD computers consisting of synchronized rudimentary processing elements (see Figure 1). Two-dimensional mesh-connected processor arrays allow links between adjacent elements. A d-dimensional *cube-connected* SIMD model considers processing elements as the corners of a d-dimensional cube and connects each processor to its d neighbors. A three-dimensional cube network *hypercube* is shown in Figure 2a. A *tree-connected* network restricts data movement to links between a processor and its parent (or children). For example, in Figure 2b, processing element with index 6 may route data to processor 3 and processor 3 may send messages to processors 1, 6 and 7.

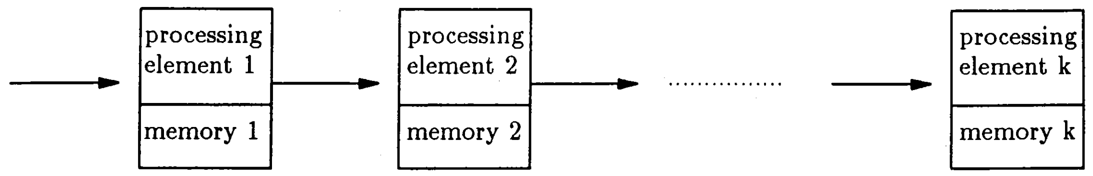


Figure 1

Linearly connected SIMD network of k processing elements

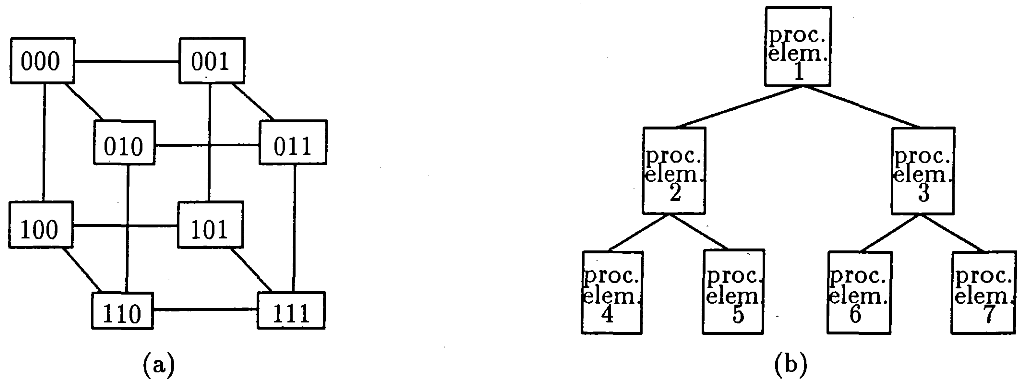


Figure 2

(a) 3-D cube connected network (hypercube) on 8 processing elements (processor i.d.'s represented in binary) (b) tree connected network on 7 processing elements

In a tree-connected system with n processors, data communication takes at most logarithmic time. A tree-connected processor array is used by Gonzales-Smith and Storer to maintain a dynamic coding dictionary [GS85, S88A].

2.2. Categorization of Compression Methods

There are a number of ways of classifying compression methods. This section describes the classifications that are relevant to parallel data compression.

Bell, Cleary and Witten draw a careful distinction between statistical and dictionary based data compression [BCW90]. Methods, such as Huffman coding and arithmetic coding, based on character frequencies are labeled *statistical* methods. Other compression techniques function by replacing large blocks of input with references to earlier occurrences of identical data. These *dictionary* methods, also called *textual substitution* and *Lempel-Ziv compression*, achieve compression by replacing phrases with pointers into some dictionary. Typically, a table or dictionary of target strings is constructed and the table indexes are encoded. Two factors differentiate versions of Lempel-Ziv coding, whether to limit how long an entry remains in the dictionary and which substrings become part of the dictionary. Sliding window compression restricts references to a fixed size window and allows any substring in the window to be included in the dictionary. Methods of this type are often labeled LZ1 compression. Fully adaptive techniques are characterized by an independently maintained dictionary that is separate from the data and are referred to as type LZ2. Instead of allowing references to any string that has previously appeared, LZ2-type dictionary compression parses the processed data into phrases, where each phrase consists of the longest matching phrase already processed plus one additional appended character. Each phrase is encoded as an index to its prefix, plus the extra character. The new concatenated string is added to the dictionary.

Not only are data compression schemes categorized as statistical or dictionary methods, but they are also classified as static or dynamic. A *static* compression method creates a fixed mapping from input characters or strings to an encoded representation. The classical static statistical method is Huffman coding [H52]. Huffman coding assigns codewords to input strings based on the probabilities of source characters. The probabilities are calculated before transmission and are used to create a prefix coding table of variable length codewords. Compression is achieved by assigning short codewords for highly probable input strings and longer codewords for less probable input. An earlier static method, Shannon-Fano coding, also attempts to assign short codewords to frequently occurring source strings [F49, FW78]. Shannon-Fano coding creates a minimal prefix code that differs only slightly from optimal. Static dictionary compression replaces repeated substrings by references to a fixed table of strings. Parallel static statistical compression schemes are discused in Section 3 and parallel static dictionary compression is the focus of Section 4.1.

Dynamic or *adaptive* models incorporate a mapping between input characters or strings and the encoded representation that evolves as the input is being transmitted. These models adapt to changes in input characteristics. Ziv and Lempel devised an adaptive dictionary coding method that parses the input into strings that are used to build a table whose indexes are then encoded into fixed length codewords [ZL78]. Frequently occurring strings are grouped together and represented by a single codeword. Welch improved the Ziv and Lempel algorithm by initializing the dictionary with the character set and building the table using the current match augmented with the subsequent input character [W84]. Thomborson and Wei implement a dynamic move-to-front text compressor on the systolic array [TW89]. Gonzales-Smith and Storer investigate dynamic dictionary compression employing different learning rules [GS85, S88A, S88B]. A number of other systolic designs implement a changing dictionary and are described in Section 4.3 [HR90, SR90, TW89, Z90A, Z90B, Z90C].

In between static and dynamic methods are *semi-adaptive schemes*, such as sliding window data compression, which encode substrings of the input as references to identical substrings occurring in a fixed-size window of characters preceding the input. The contents of the window can be viewed as a semi-adaptive dictionary.

Sliding window compression methods are based on the LZ1 model (described in [ZL77]) and systolic implementations are reported in Section 4.2.

Most compression methods are codeword-based. Codeword-based compression schemes replace input substrings by codewords to obtain a more compact representation of the input. Huffman coding is an example of codeword-based compression. However, in some compression schemes, such as arithmetic coding, it is not possible to identify the particular input character that caused a particular bit of the encoded stream. For codeword-based compression methods, let $\alpha = \{s_1, s_2, \ldots, s_{n-1}, s_n\}$ be the *source alphabet.* A *source message* is a concatenated sequence of characters over the alphabet α. Let $\beta = \{0, 1, 2, \ldots, \gamma - 1\}$ be the *code alphabet.* A code $\mathcal{C} = \{c_1, c_2, \ldots, c_m\}$ is a finite nonempty set of finite sequences over code alphabet β. Each c_i is a *codeword.* A *message* over $\mathcal{C}$ is a string resulting from the concatenation of codewords from $\mathcal{C}$. A code $\mathcal{C}$ is *distinct* if the assignment of source words (strings over the source alphabet) to codewords is one-to-one. A code $\mathcal{C}$ is *uniquely decipherable* if the message created by the encoding of a sequence of input words has a unique decomposition (i.e. codewords are distinguishable from the entire compacted message). A code $\mathcal{C}$ is a *prefix* code if no codeword in $\mathcal{C}$ is a prefix of another codeword. Note that any prefix code is uniquely decipherable.

Instead of constructing a mapping from source messages to codewords, arithmetic coding represents the input string by a subinterval of the interval between 0 and 1 on the real line [WNC87]. The method uses the probabilities of the source to successively narrow the interval used to represent the input. Ultimately, the interval is be narrowed sufficiently so that only the source string would be represented by any number in the interval. Arithmetic coding dispenses with the restriction that every character in the source message must be represented by an integral number of bits. Because of this property, arithmetic coding is capable of achieving compression results that are arbitrarily close to the entropy of the source, defined below.

There are a number of measures used to determine the "goodness" of a particular code. Of interest to this survey is the notion of an *optimal* or *minimum-redundancy* code.[3] A minimum-redundancy code has minimum average codeword length for a given discrete probability distribution of the source [LH87]. This definition is based on the information theory concept of *entropy* which is a measure of the information content of a message. For an unpredictable source, entropy (information content) is high; for an ordered source, entropy is low. Formally, for source alphabet $\alpha = \{s_1, s_2, \ldots, s_n\}$ with probabilities of occurrence $\{p_1, p_2, \ldots, p_n\}$ and distinct code $C = \{c_1, c_2, \ldots, c_n\}$, the expression[4] $\sum_{k=1}^{n} -p_k \lg p_k$ denotes the entropy of the source and $\sum_{k=1}^{n} p_k len(c_k)$ is the average codeword length, where $len(c_k)$ is the length of codeword c_k. Theoretically, the minimum length of a compressed message should equal its entropy. That is, since the length of a code message must be sufficient to carry the information of the corresponding source message, entropy imposes a lower bound on codeword length [LH87]. A minimum-redundancy code for source alphabet α minimizes the difference between the average codeword length and the entropy of the source.

Data compression schemes can be further categorized as either off-line, on-line, or real-time. An *off-line* model can manipulate and preprocess the entire input string prior to coding. In *on-line* models, neither the sender or receiver can see all of the data at once; that is, data is constantly flowing through the encoder, transmitted, and pushed through the decoder. On-line algorithms are further distinguished as *real-time* methods if, for some constant k, exactly one new character is read into the encoder and one character is written by the decoder

[3] In the parallel computation community, the term "optimal" is used to describe efficient parallel algorithms. Therefore, in this paper, "optimal" is reserved for describing parallel algorithms and "minimum-redundancy" is used for coding.
[4] In this paper, lg denotes the base 2 logarithm.

every k units of time [SR90]. On-line algorithms are forced to construct the coding dictionary "on the fly". The scheme is designed to "learn" an approximate distribution of the data and to adapt to fluctuations in the source.

Many static compression schemes, such as Huffman coding, that create the compression mapping prior to data transmission, can be viewed as two-phase methods. The first phase, operating off-line, analyzes the character probabilities. The second phase matches the input against the codeword table to perform the actual compression. The work of Teng [T87], Kirkpatrick and Przytycka [KP90], Larmore and Przytycka [LP91], and Atallah et al. [AKLMT89] focuses on the first phase by investigating the off-line parallel construction of a Huffman prefix code. Gonzales-Smith and Storer use a two-phase parallel data compression method, implemented on a systolic array, that assumes the existence of the static coding dictionary (created off-line) and transmits the coded message on-line [GS85, S88A].

Parallel static compression systems based on character statistics are surveyed in Section 3. Section 4.1 presents systolic implementations of static dictionary methods and sliding substitutional designs are described in Section 4.2. Dynamic dictionary approaches implemented on the systolic pipe are covered in Section 4.3.

2.3. Evaluation of Compression Methods

A common measure used to evaluate and compare coding techniques is compression ratio. There are several different definitions of compression ratio which attempt to describe the space reduction attained by compression. For example, compression ratio has been defined as the ratio (encoded message length)/(source message length), as the ratio (source message length)/(encoded message length), as the number of bits per input character, and as 1−(encoded message length)/(source message length) [LH87, S88, BCW90]. In this paper, *compression ratio* is defined as the ratio (encoded message length)/(source message length). *Compression savings* is defined by 1−(compression ration). For example, if the input string to an encoder consists of 2000 bits and the corresponding output is 500 bits, the compression ratio is $500/2000 = .25$ or 25% and the compression savings is $1 - 500/2000 = .75$ or 75%. Compression ratios describe compression effectiveness but do not take other important performance measures into consideration. For instance, one compression scheme may achieve 80% compression savings but may take an unreasonable amount of time to execute. Another scheme may give poorer compression ratio but perform in real-time. When possible, methods are compared in terms of speed, space usage, compression ratio, and system bandwidth[5].

In the study of parallel complexity, problems are classified according to their use of time and processor resources. The class NC incorporates a hierarchy of problems that are solvable by deterministic parallel algorithms that operate in time bounded by a power of the logarithm of the size of the input using a polynomially-bounded number of processors.

Work is another measure used to evaluate parallel algorithm performance. The work done by a parallel algorithm is defined as the product of the time and processor requirements. If Seq($\mathcal{P}$) is the time complexity of the fastest known sequential algorithm for a problem $\mathcal{P}$ then a parallel algorithm is *optimal* if it takes $O(\text{Seq}(\mathcal{P})/\text{P})$[6] time using $O(\text{P})$ processors. Moreover, the work performed by an optimal algorithm is proportional to the time required by the fastest known sequential algorithm. As mentioned earlier, parallel computation adds the dimension of processor usage to algorithm evaluation. The processor requirements of each parallel system are given to aid in this comparison. The theoretical PRAM model cannot be physically implemented and is therefore limited to theoretical evaluation. For other parallel models, empirical findings are included. Statistical coding on the PRAM is described in the next section.

[5] The *bandwidth* of a device is measured as the number of bytes transferred per unit time.
[6] *O-notation* represents an upper bound on the asymptotic behavior of a function.

3. Parallel Statistical Coding

A statistical compressor assigns codes based on probabilities of individual symbols. Static Huffman compression calculates character frequencies during a preprocessing pass over the source data. This information is used to assign codewords so that short codes correspond to high-frequency symbols and longer codes are given to low-probability characters. The second pass encodes the source data using the generated codewords. This section examines parallel approaches to statistical coding.

Huffman compression generates a prefix code such that the average word length is minimal. The prefix code is equivalent to a full[7] binary tree with the source symbol probabilities associated with the leaves. To construct this code tree, Huffman's algorithm proceeds as follows [H52]. Initially, each probability is assigned to a tree of height 0 (i.e., a single node). Iteratively, the pair of trees corresponding to the two smallest probabilities are combined into a single tree with an associated probability equal to the sum of the frequencies of the two original trees. Huffman's scheme constructs a minimum-redundancy prefix code in $O(n \log n)$ time, where n is the size of the source alphabet. If the symbol frequencies are presorted, Huffman's method requires only linear time. Applying a recursive description of Huffman tree creation, Teng developed the first parallel algorithm for Huffman coding [T87]. Teng's approach implements a parallel dynamic programming solution and runs in $O(\log^2 n)$ time using $O(n^6)$ processors. Although unreasonable resource bounds render this solution impractical, the results are significant since they were the first to place the Huffman coding problem in the computational class $\mathcal{NC}$. Further work by Atallah et al. lowered the time and processor requirements by taking advantage of implicit properties of the tree corresponding to the graph-theoretical interpretation of Huffman's solution [AKLMT89]. Section 3.1 describes parallel minimum-redundancy prefix code creation based on

[7] A binary tree is *full* if every internal (non-leaf) node has exactly two children.

dynamic programming and Section 3.2 surveys improved approaches which profit from concave matrix multiplication and approximation. Section 3.3 considers other parallel statistical coding methods.

3.1. Huffman Coding Reduced to Parallel Circuit Evaluation

The first parallel Huffman code construction algorithm solved the problem indirectly by a uniform reduction to a min-plus circuit value problem of polynomial size and linear degree [T87]. The min-plus circuit value problem can be solved in logarithmic time with a polynomial number of processors. This reduction coupled with the efficient circuit evaluation algorithm yielded the first $\mathcal{NC}$ algorithm for the creation of minimum-redundancy prefix codes.

The reduction is based on a recursive definition of minimal average word length. Namely, let the input to the Huffman coding algorithm consist of a sequence $(p_1, p_2, \ldots, p_n)$ of source character probabilities and let $H(i,j)$ be the average word length of a Huffman code for probabilities $(p_i, \ldots, p_j)$. Initially the input sequence is sorted into nondecreasing order in $O(\log n)$ time using $O(n)$ processors. Then the values of $H(i,j)$ are given by the following recurrence relation:

$$H(i,j) = \begin{cases} 0 & i = j \\ \min_{k=i+1}^{j}\{H(i,k-1) + H(k,j)\} + \sum_{r=i}^{j} p_r & i < j \end{cases} \qquad (a)$$

An example of this dynamic programming approach to sequential code creation is given in Figure 3. The idea is to build a tree of size k by taking the minimum total path length over all possible tree configurations of size less than k.

Teng provides a sequential algorithm, implementing the above recursive definition, for building a minimum-redundancy prefix code. It can be sketched as:

1. Initialize $H(i,j) = 0$ for $i = j$ and $H(i,j) = +\infty$ for $i < j$.
2. For $i < j$ estimate $H(i,j)$ applying relation (a) and the values of H obtained during the previous step.
3. If any H value changed since previous iteration, return to step 2.

$p_1 = .36 \qquad p_2 = .29 \qquad p_3 = .25 \qquad p_4 = .1$

$$H(1,2) = H(1,1) + H(2,2) + p_1 + p_2 = .65$$

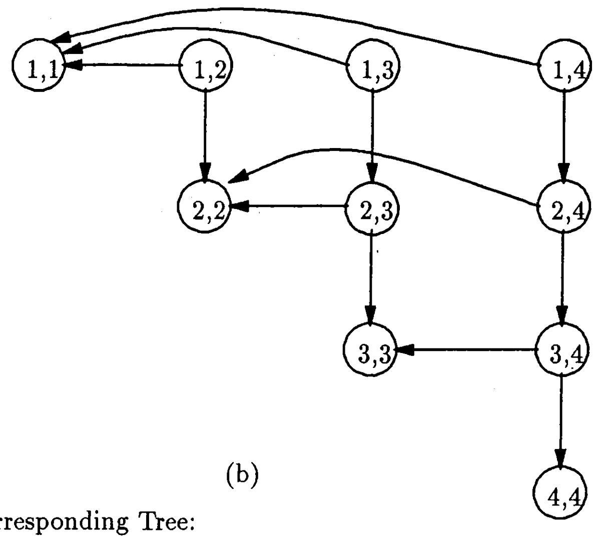
$$H(2,3) = H(2,2) + H(3,3) + p_2 + p_3 = .54$$

$$H(2,4) = \min \begin{cases} H(2,2) + H(3,4) + p_2 + p_3 + p_4 \\ H(2,3) + H(4,4) + p_2 + p_3 + p_4 \end{cases} = .99$$

$$H(1,4) = \min \begin{cases} H(1,1) + H(2,4) + p_1 + p_2 + p_3 + p_4 \\ H(1,2) + H(3,4) + p_1 + p_2 + p_3 + p_4 \\ H(1,3) + H(4,4) + p_1 + p_2 + p_3 + p_4 \end{cases} = 1.99$$

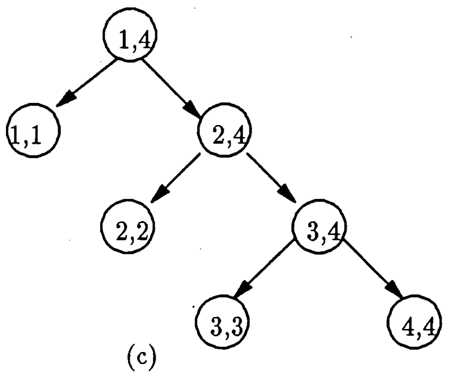
(a)

Directed graph induced by H(i,j):



(b)

Corresponding Tree:



(c)

Figure 3

(a) Dynamic programming solution (b) Directed graph (c) Huffman tree

Teng reduces the algorithm to a min-plus circuit value problem which can be evaluated in $O(\log^2 n)$ time using a polynomial number of processors on the CRCW PRAM [MRK85]. This results in an $\mathcal{NC}$ algorithm for generating the values $H(i,j)$, for all i and j. It remains to derive the tree and corresponding codewords from the H values. Teng describes a construction which builds a directed graph whose vertex set is the collection $\{H(i,j) | 1 \leq i \leq j \leq n\}$ and whose edges connect vertex $H(i,j)$ with the vertices $H(i,k-1)$ and $H(k,j)$ where k is given by the recursive definition of $H(i,j)$. The directed graph induced by $H(1,n)$ is made into a tree by marking all of the nodes reachable from the root $H(1,n)$ in $O(\log n)$ time using a polynomial number of processors. Figures 3b and 3c show the directed graph and final tree for the problem in Figure 3c. The resulting tree represents a minimum-redundancy prefix code for the source character probabilities $(p_1, p_2, \ldots, p_n)$ and is constructed in $O(\log^2 n)$ using $O(n^6)$ processors on the CRCW PRAM model. The codeword for each source character can be generated in $O(\log n)$ time using $O(n/\log n)$ processors by tree contraction [MR85]. Tree contraction is useful in parallel tree manipulation and is the basis of the approach taken by Atallah et al. to improve on Teng's original result [AKLMT89].

Miller and Reif define RAKE and COMPRESS operations on trees [MR85]. Let RAKE be an operation that removes all leaves from a tree and let COMPRESS be an operation that halves each chain of nodes (from leaf to root) by pointer doubling. Atallah et al. consider a restricted form of the RAKE operation where a leaf is removed only if its siblings are leaves [AKLMT89]. They show that any left-justified[8] tree can be reduced to a single chain of vertices along the leftmost path of the tree in at most $\lceil \log n \rceil$ applications of RAKE. They also notice that each iteration of Step 2 in Teng's sequential algorithm simulates the RAKE operation and can be done in $O(\log n)$ using $n^3/\log n$ processors on the CREW PRAM model

[8] A binary tree T is *left-justified* if for every pair of siblings u and v, with u to the left of v, if the subtree T_v rooted at v is not empty at some level l in the tree then the subtree T_u rooted at u is full at level l.

of computation. However, the algorithm requires $O(n)$ iterations and therefore yields an $O(n \log n)$ total time bound.

The execution performance can be reduced to $O(\log n)$, using the same number of processors, by introducing a step which carries out the COMPRESS operation and thereby reduces the height of the tree [AKLMT89]. The COMPRESS step estimates a quantity $F(i,j)$, where $H(1,i) + F(i,j)$ is the minimum average word length of a tree over source frequencies $(p_1, p_2, \ldots, p_j)$ restricted to containing a subtree corresponding to $(p_1, p_2, \ldots, p_i)$. Quantity $F(i,j)$ can be defined recursively in terms of precomputed H values and previous F values as follows:

$$F(i,j) = \begin{cases} H(i+1,j) + \sum_{r=1}^{j} p_r & i+1 = j \\ \min \begin{cases} H(i+1,j) + \sum_{r=1}^{j} p_r \\ \min_{k=i+1}^{j-1} \{F(i,k) + F(k,j)\} \end{cases} & i+1 < j \end{cases} \qquad (b)$$

Atallah et al. provide the following sketch of the algorithm which performs $\lceil \log n \rceil$ RAKE operations followed by $\lceil \log n \rceil$ COMPRESS operations to reduce the tree to a single node [AKLMT89].

1. Initialize $H(i,j) = 0$ for $i = j$ and $H(i,j) = +\infty$ for $i < j$.
2. Iterate $\lceil \log n \rceil$ times: For $i < j$ estimate $H(i,j)$ applying relation (a) using the values of H obtained during the previous step.
3. Initialize $F(i,j) = H(i+1,j) + \sum_{r=i}^{j} p_r$.
4. Iterate $\lceil \log n \rceil$ times: For $i < j$ estimate $F(i,j)$ applying relation (b) using the values of F obtained during the previous step.

The final value of $F(1,n)$ gives the average word length of the minimum-redundancy prefix code. As noted above, since any left-justified tree can be reduced to a single leftmost chain of nodes by $\lceil \log n \rceil$ applications of RAKE, $\lceil \log n \rceil$ COMPRESS operations on a chain reduces the tree to the empty tree. Therefore, the above algorithm computes the quantity $F(1,n)$ in $O(\log n)$ time using $O(n^3/\log n)$ processors on a CRCW PRAM. Although the paper does not mention the generation of the corresponding tree or codewords from the computed H and F values, an approach similar to Teng's (described earlier) provides these within the same resource bounds. Also, Teng proves that, for any nondecreasing sequence of probabilities $(p_1, p_2, \ldots, p_n)$, there is a left-justified Huffman tree representing a minimum-redundancy prefix code for $(p_1, p_2, \ldots, p_n)$ [T87]. Thus, utilizing a parallel dynamic programming approach, Huffman codes can be constructed for a given list of probabilities, in $O(\log n)$ time using $(n^3/\log n)$ processors. These bounds can be improved by formulating the Huffman coding problem in terms of multiplications of concave matrices. This approach is discussed in the following section.

3.2. Coding as the Multiplication of Concave Matrices

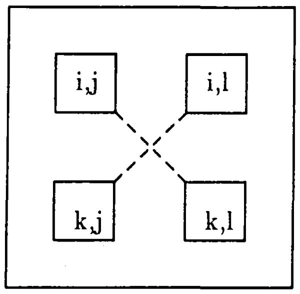
In light of the sequential $O(n \log n)$ performance of Huffman's algorithm, the parallel dynamic programming solution of the previous section is of little practical value since it requires $O(n^3)$ work. This section discusses an alternative approach due to Atallah et al. which runs in $O(\log^2 n)$ time using $n^2/\log n$ processors [AKLMT89]. The bottleneck of the dynamic programming algorithms is the n^3 processor bound that arises from multiplication of arbitrary matrices. Concave matrices are a subclass of matrices that can be multiplied more efficiently in parallel. By formulating the Huffman tree problem as a multiplication of concave matrices, the processor requirement is reduced.

A concave matrix M is a rectangular matrix that satisfies the *quadrangle condition* (see Figure 4). Specifically, for $n \times m$ matrix M, the following inequality holds for all $1 \leq i < k \leq n$, $1 \leq j < l \leq m$:

$$M[i,j] + M[k,l] \leq M[i,l] + M[k,j]$$

Atallah et al. give a recursive algorithm for multiplying concave matrices over the closed semi-ring $(min, +)$ which runs in $O(\log n \log\log n)$ time using $n^2/\log n$

Matrix M: $M[i,j] + M[k,l] \leq M[i,l] + M[k,j]$



Matrix M

1	5	9	13	17
1	4	7	10	13
1	3	5	7	9
1	2	3	4	5
1	1	1	1	1

Example

Figure 4

The quadrangle condition

processors on the CREW PRAM, and $O((\log\log n)^2)$ time using $n^2/(\log\log n)$ processors on the CRCW PRAM [AKLMT89]. By taking advantage of the more efficient concave matrix multiplication, they describe a solution to the Huffman tree construction problem that runs in $O(\log^2 n)$ time using $n^2/\log n$ processors. Their approach reduces Huffman coding to a minimum-weighted path problem for a directed graph which can be solved via parallel concave matrix multiplication. The reduction is two-fold. As mentioned earlier, for any nondecreasing sequence of probabilities $(p_1, p_2, \ldots, p_n)$ there exists a left-justified tree representing the corresponding minimum-redundancy code such that the heights of the subtrees not on the leftmost path are no more than $\lceil \log n \rceil$. The first step of the reduction builds minimum-redundancy height-limited subtrees of height at most $\lceil \log n \rceil$ for all possible subintervals $(p_i, \ldots, p_n)$. The resulting information is represented as a matrix A which is computed in $O(\log^2 n)$ time using $n^2/\log n$ processors by a reduction to recursive multiplication of concave matrices.

The matrix A generated in step 1 is augmented to form matrix M. Matrix M has no simple meaning in terms of Huffman trees. But, matrix $M^{2^{\lceil \log n \rceil}}$ gives the minimum weighted path length of the minimum-redundancy Huffman tree for probabilities $(p_1, p_2, \ldots, p_n)$ and the information needed to construct the tree. The second phase consists of the creation of matrix M and a series of $\lceil \log n \rceil$ concave matrix multiplications which can be performed in $O(\log n)$ time using $n^2/\log n$ processors. This two-phase reduction yields the Huffman tree in a total of $O(\log^2 n)$ time using $n^2/\log n$ processors on the CREW PRAM. On the CRCW PRAM, the resource bounds fall to $O(\log n(\log\log n)^2)$ time and $n^2/(\log\log n)^2$ processors.

3.3. Other Parallel Statistical Schemes

Larmore and Przytycka give a reduction of the Huffman tree problem to the Concave Least Weight Subsequence problem resulting in a new linear time sequential algorithm and a more efficient parallel algorithm [LP91]. Given a concave triangular matrix of weights $\{w(i,j) | 0 \le i < j \le n\}$, the Concave Least Weight Subsequence problem is to find a subsequence $0 = \beta_0 < \beta_1 < \cdots < \beta_m = n$ which minimizes the sum $\sum_{k=1}^{m} w(\beta_{k-1}, \beta_k)$. This subsequence can be found in sublinear time. Reducing the Huffman tree problem to the Concave Least Weight Subsequence problem results in an $O(\sqrt{n}\log n)$-time and n-processor parallel algorithm. Although the solution is not in $\mathcal{NC}$, it performs less total work than any other sublinear time parallel Huffman algorithm. Further research is needed to find an optimal sublinear time Huffman tree construction algorithm.

Generation of near-minimum-redundancy codes can be done optimally in parallel. Shannon-Fano coding is an example of a statistical coding scheme which produces a near-minimum-redundancy prefix code such that the average codeword length exceeds the minimum length by at most 1 bit. An optimal $O(\log n)$ time, $n/\log n$ processor EREW PRAM algorithm for near-optimal code construction is described by Atallah et al. [AKLMT89]. Nearly minimum-redundancy code creation is reduced to the problem of constructing a tree given a monotonic sequence of leaf levels. Initially, the input frequencies $(p_1, p_2, \ldots, p_n)$ are sorted and a sequence of lengths $(l_1, l_2, \ldots, l_n)$ are calculated such that $\log(1/p_i) \leq l_i \leq \log(1/p_i) + 1$. Next, tree T is constructed optimally by invoking the algorithm for monotonic leaf level sequences. Tree T is then compressed using parallel tree contraction resulting in a minimum-redundancy prefix codeword tree T'. Atallah et al. claim that tree T' is the Shannon-Fano tree [AKLMT89] In the same paper, a parallel algorithm for constructing almost optimal binary search trees is presented which can be used to build trees which differ from a minimum-redundancy prefix code by at most $1/n^k$ bits in $O(k \log^2 n)$ time and $n^2/\log^2 n$ processors.

Approximate solutions to the minimum-redundancy coding problem are investigated by Kirkpatrick and Przytycka [KP90]. They give an $O(\log n \log^* n)$-time, n-processor CREW algorithm for finding an approximate solution to the problem.

A variation of the Huffman tree problem is the alphabetic version which, given a sequence of probabilities $(p_1, p_2, \ldots, p_n)$, finds a binary tree of minimum weighted path length, with weight p_i assigned to the i^{th} leaf. An $\mathcal{NC}$ algorithm is given for an approximate solution to the alphabetic Huffman coding problem using a parallel implementation of the Package Merge technique due to Larmore [LP91]. The $O(\log^2 n)$ time, n processor algorithm improved an earlier approximation which required an additional factor of n processors. Since the alphabetic Huffman coding problem can be solved sequentially in $O(n \log n)$ time, further work is needed to eliminate an additional $\log n$ factor to obtain an optimal parallel solution.

Concurrency can be introduced into all phases of a compression system. That is, parallelism can speed up code creation, encoding and decoding. Once the code has been selected, the input message can be encoded and decoded in linear time by replacing each character by its corresponding code. In 1987, the decoding problem was solved optimally in parallel. Moreover, Teng and Weng give an optimal EREW PRAM algorithm for decomposing prefix-coded messages and uniquely decipherable-coded messages in $O(\log n)$ time and $O(n/\log n)$ processors

[TW87]. They reduce the decoding problem to the problems of parallel finite-state automata simulation and the evaluation of prefix sums. To complete the solution, they present an optimal parallel simulation algorithm for finite-state automata using dynamic expression evaluation and parallel tree contraction techniques. Since uniquely decipherable codes provide more compression than prefix codes and can be decompressed with no additional computational effort, they conclude that uniquely decipherable codes are superior to prefix codes [TW87].

The above parallel coding methods are of theoretical interest, however they are not directly realizable in hardware. Lea reports on a hardware implementation of a text compression system based on n-gram coding [L78]. For n-gram coding, the dictionary consists of a collection words each of length exactly n. The dictionary is stored in an associative memory and parallel manipulation of the table is conducted on an associative parallel processor.[9] Two different implementations, based on fixed record length and byte-organized variable record length, significantly reduce overheads in execution time and program storage when compared to software implementations. In the next section, parallel systems for dictionary compression are examined.

4. Parallel Dictionary Compression

Dictionary or substitutional coding removes data redundancy by replacing recurrent input substrings by references to earlier copies [RPE81, SS82]. Usually, such a reference is called a pointer and the substring being referenced is called the target. Targets are maintained in a dictionary of phrases that are expected to occur frequently. Dictionary-based compression techniques are distinguished by their use and maintenance of the coding dictionary. Some methods restrict the length of dictionary entries. Within this restriction, the dictionary can be static, semi-adaptive, or adaptive. A static dictionary is created before any encoding or

[9] An associative parallel processor is a processor array with an associative memory.

decoding begins and must remain unchanged. Better compression is achieved by adaptive methods that allow additions, deletions, and changes to the collection of referenced strings during the course of encoding.

Dictionary techniques are further classified as external or internal. External dictionary or LZ2-type schemes store target phrases in a separate dictionary and the data stream is compressed by replacing occurrences of repeated substrings by indexes into the dictionary. The resulting compressed stream contains characters of the input alphabet interspersed with pointers into the dictionary. Decoding reconstructs the source string by substituting dictionary entries for pointers. Internal substitution methods (also referred to as sliding window or LZ1-type coding) do not maintain an explicit dictionary. Instead, repeated substrings are replaced by pointers to earlier occurrences of the same substring. The resulting string of characters and pointers contains the compression dictionary implicitly. Recursive schemes, implemented internally or externally, permit pointer targets to contain pointers.

Once the dictionary has been selected, the input stream must be parsed to determine which substrings are to be replaced by dictionary pointers. The most straightforward approach is *greedy* parsing where at each step the encoder finds the longest dictionary phrase that matches a prefix of the uncoded portion of the input stream. That is, the input stream is compared to each word in the dictionary and the entry corresponding to the longest prefix of the uncoded portion of the input stream is used to encode the input prefix. In the parallel setting, this longest match step can be executed concurrently by a collection of processors [BCW90]. For a dictionary of size N, $2N-1$ processors configured as a binary tree can find the longest match in $O(\log N)$ time. Each leaf processor is assigned to perform comparisons for a different dictionary entry. The remaining $N-1$ processors coordinate the results via signals that propagate up and down the tree in $O(\log N)$ time. Figure 5 is an example of the parallel match step for dictionary "abc," "acb," "bac," "bca," "cab", "cba", "aa", and "bb" and input string "baccbaacb." Processing elements 1 through 8 compare prefixes of the input stream to their corresponding dictionary entries and propagate, in the case of a match, their processor identification and match length, and a 0 otherwise. Processors 9 through 14 compare the match lengths of non-zero inputs and output the processor identification and match length of the input having the longest match. Other parallel implementations can be devised based on different processor configurations. Zito-Wolf presents a more efficient implementation using pipelined trees [Z90c]. Systolic architectures for the dictionary match step are considered later in this section.

In the parallel VLSI environment, static, semi-adaptive, and dynamic dictionary schemes have been considered using the systolic array. One advantage of the systolic implementation is that a larger pipe can be fabricated by placing a sequence of processing elements on a single chip, and then joining a series of chips on a board. Another benefit is that the length of interprocessor connections are constant and independent of the size of the array. Systolic architectures for dictionary compression reduce the computational overhead by accelerating both encoding and decoding and are therefore suitable for a larger range of applications.

4.1. Static Dictionary Compression on the Systolic Array

Gonzales-Smith and Storer give parallel algorithms for data compression using static dictionary coding [GS85, S88a]. They implement a recursive static dictionary which replaces input substrings by indices into a static table of dictionary entries, each of which may contain pointers to other indices. This allows for the representation of strings longer than the maximum-length dictionary entry and therefore may reduce both the maximum length of a dictionary entry and the size of the VLSI implementation. Also, a pointer is permitted to point to a suffix of a dictionary entry and pointers may be of variable size. The dictionary is

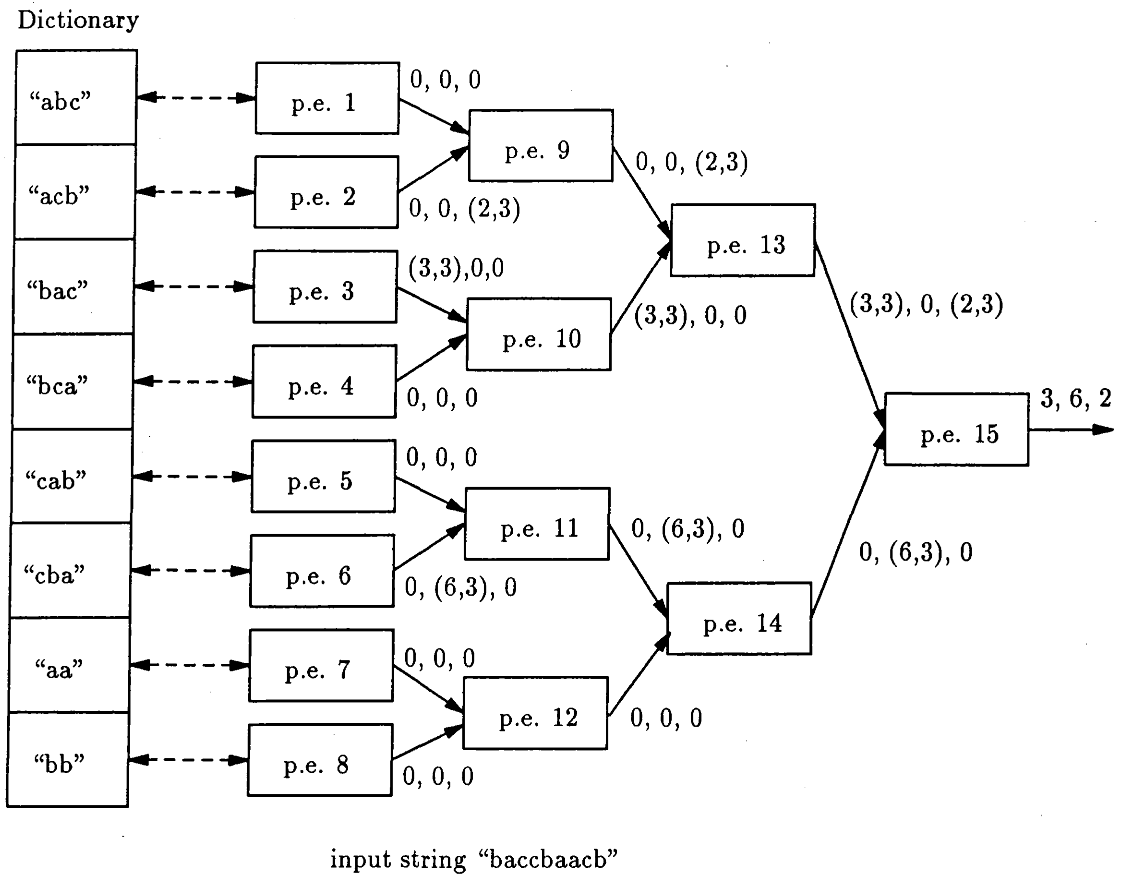


Figure 5

Parallel longest match step for dictionary of size $N = 8$

assumed to be available and details of its construction are not discussed. However, the importance and complexity of dictionary selection are emphasized in that compression performance is directly related to the "goodness" of the dictionary.

The systolic encoding/decoding pipe consists of a series of processing elements linearly connected by a two-way communication channel. A schematic of the architecture is shown in Figure 6. Each processing element stores a dictionary element. The two-way communication channel allows both the compression and expansion algorithms to use the same dictionary structure. The dictionary is constructed prior to compression and loaded into the processors. For purposes

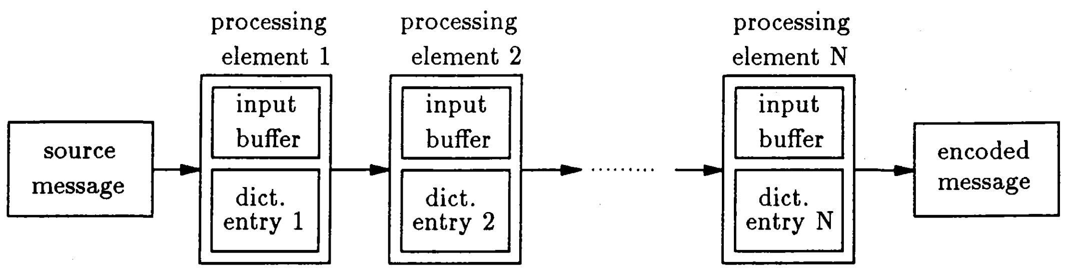


Figure 6

Systolic array for static dictionary coding

of explanation, encoding is assumed to proceed from left to right, and decoding from right to left. Encoding is performed by parallelizing a greedy algorithm for compressing substrings. Input characters are piped into a processor from the left and compared against the dictionary entry stored in that processor. If the length of the match exceeds the size of a pointer to the dictionary substring, then the matched data is replaced by the pointer. An example of the encoding process is given in Figure 7.

An optimal algorithm for computing the minimum compressed form of a substring requires access to the entire input string and may involve global data flow. These restrictions prohibit parallelism. Gonzales-Smith and Storer prove that a greedy parsing strategy is a reasonable approach whose performance is close to that of the optimal algorithm [GS85, S88A]. The greedy algorithm may also require global communication among processors in a systolic architecture if the entire dictionary must be searched to determine the longest match. This can be avoided by enforcing three conditions: the dictionary entries must be organized in order of shortest to longest strings, encoding must proceed from left to right and suffixes of dictionary elements cannot be prefixes of other dictionary entries. Performance of the greedy approach is unknown when these assumptions fail to hold.

Source alphabet: $\{a, \ldots, z, A, \ldots, Z, ., ,, ;, !, ?\}$
Input String: Data coding removes redundant information.
Dictionary:

00 – 04	ƀinfo
05 – 11	ƀcoding
12 – 18	ƀremove
19 – 22	tion
23 – 28	Data05
29 – 36	ƀredunda
37 – 44	29nt00rm
45 – 49	13s37

Initial	8	7	6	5	4	3	2	1	Processor
Configuration	49–45	44–37	36–29	28–23	22–19	18–12	11–5	4–0	Pointers
	13s37	29nt00rm	ƀredunda	Data05	tion	ƀremove	ƀcoding	ƀinfo	Dict. Entry
									⟵ Data cod. . .
After 16	8	7	6	5	4	3	2	1	
cycles	13s37	29nt00rm	ƀredunda	Data05	tion	ƀremove	ƀcoding	ƀinfo	
						Data	ƀcoding	remov	⟵ es redun. . .
After 21	8	7	6	5	4	3	2	1	
cycles	13s37	29nt00rm	ƀredunda	Data05	tion	ƀremove	ƀcoding	ƀinfo	
						Data	05remov	esƀre	⟵ dundant. . .
After 29	8	7	6	5	4	3	2	1	
cycles	13s37	29nt00rm	ƀredunda	Data05	tion	ƀremove	ƀcoding	ƀinfo	
				D	ata0	5remove	sƀredun	dantƀ	⟵ info. . .
After 33	8	7	6	5	4	3	2	1	
cycles	13s37	29nt00rm	ƀredunda	Data05	tion	ƀremove	ƀcoding	ƀinfo	
				D	ata0	513sƀre	dundant	ƀinfo	⟵ rmation. . .
After 41	8	7	6	5	4	3	2	1	
cycles	13s37	29nt00rm	ƀredunda	Data05	tion	ƀremove	ƀcoding	ƀinfo	
				Data05	13sƀ	redunda	nt00rma	tion.	
After 45	8	7	6	5	4	3	2	1	
cycles	13s37	29nt00rm	ƀredunda	Data05	tion	ƀremove	ƀcoding	ƀinfo	
				2313sƀ	redu	ndant00	rmation	.	
After 58	8	7	6	5	4	3	2	1	
cycles	13s37	29nt00rm	ƀredunda	Data05	tion	ƀremove	ƀcoding	ƀinfo	
		2313s	ƀredunda	nt00rm	atio	n.			
After 65	8	7	6	5	4	3	2	1	
cycles	13s37	29nt00rm	ƀredunda	Data05	tion	ƀremove	ƀcoding	ƀinfo	
		2313s	29nt00rm	a19.					
After 79	8	7	6	5	4	3	2	1	
cycles	13s37	29nt00rm	ƀredunda	Data05	tion	ƀremove	ƀcoding	ƀinfo	
	2313s	37a19.							
After 81	8	7	6	5	4	3	2	1	
cycles	13s37	29nt00rm	ƀredunda	Data05	tion	ƀremove	ƀcoding	ƀinfo	
23⟵	13s37	a19.							
After 84	8	7	6	5	4	3	2	1	
cycles	13s37	29nt00rm	ƀredunda	Data05	tion	ƀremove	ƀcoding	ƀinfo	
	45a19.								

COMPLETE OUTPUT: 2345a19.

Figure 7

Example of encoding using systolic implementation of a static dictionary using 8 processors

Decoding on the systolic array is similar to encoding. The compressed string enters the pipe from the right and expansion consists of replacing each pointer by its target string. In particular, processor i compares its identification to the incoming pointer and if the processor finds a match, it outputs its corresponding dictionary entry.

A difficulty in this design concerns buffer overflow errors that occur when data moves too quickly through parts of the array. A locking scheme prevents local buffer overflow. That is, no additional characters are read in until there is available space in the buffer. Unfortunately, locking signals can propagate up the pipe, eventually locking the entrance processor. The Gonzales-Smith and Storer architecture avoids this global system lock by assuming that the input data rate into the decoding circuit is commensurate with the speed of the compression chip [GS85, S88A]. As described, static dictionary compression requires no additional overhead for maintaining the dictionary but suffers from the performance limitations of a non-adaptive technique. The next section considers semi-adaptive techniques which achieve better compression performance by adapting to characteristics of the input.

4.2. Sliding Window Method on the Systolic Array

Systolic algorithms for the sliding (LZ1-type) dictionary model compress text by replacing repeated substrings by pointers to earlier occurrences of the identical substring. In this scheme, pointers denote phrases in a fixed-size window immediately preceding the current coding position. For an implicit dictionary or "sliding window" of size N, the systolic design of Gonzales-Smith and Storer stores the last $2N$ characters processed, one item per processing element [GS85, S88A]. Also, each processor has three additional registers for holding an input character and its encoding information (see Figure 8). The additional N processors form a *lookahead buffer* that is used to aid in the continuous maintenance of the semi-adaptive dictionary.

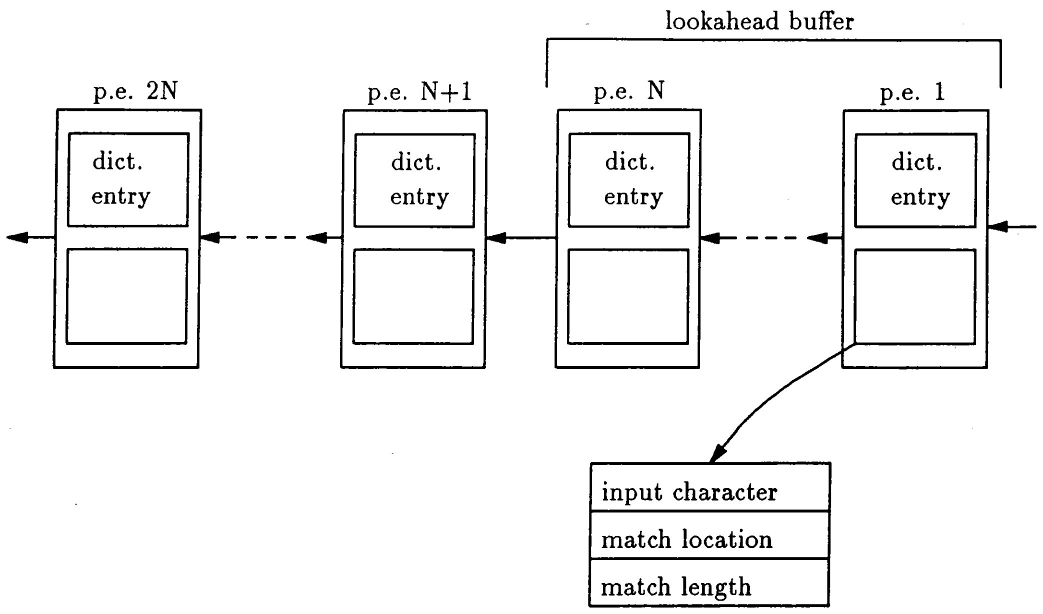


Figure 8

Systolic sliding window architecture

The systolic encoder consists of $2N$ linearly-connected processing elements. To encode the current character in the sliding dictionary model, the window is searched for the longest match with the lookahead buffer. Encoding proceeds from right to left and the dictionary is continuously updated by moving the fixed-size window over the input, removing symbols on the left and adding new characters on the right. As data is piped through the array, information is maintained for each input character on the position and length of the longest match which is encoded as a triple (*position, length, next character*). "Next character" is the first character that did not match the substring in the window. As a character travels through the pipe, it is accompanied by its longest match location and length information which is updated whenever a longer match is found.

Gonzales-Smith and Storer describe an encoding method that performs comparisons on blocks of N input characters at a time. During processing of these characters, N new characters are read in, and N previously coded symbols are output. In particular, let $a_{N+1}, \ldots, a_{2N}$ be the sequence of characters read in during the processing of characters $a_1, \ldots, a_N$. Since a block of size N has been coded, each processor updates its local dictionary element by replacing it by the current contents of its input register and, after an additional left shift, encoding continues. For the next N cycles, characters $a_{N+1}, \ldots, a_{2N}$ are compared against each character in the correct dictionary of input characters $a_1, \ldots, a_N$. Notice that not all processors participate in each system cycle. By knowing their own processor identification and the number of system cycles, each processor can determine which comparisons to perform. Processor $2N$ has the additional function of handling the longest match position and length information and, when the length l exceeds the size of a pointer, the pointer is output and the next l characters in the pipe are ignored. Moreover, whenever pointers overlap, processor $2N$ alters the output pointer to maximize compression.

Calculating the longest match information requires communication among non-neighboring processors. Gonzales-Smith and Storer investigated two schemes for updating match position and length figures [GS85, S88A]. The first design stacks a binary tree-connected collection of $2N - 1$ processors on top of the systolic architecture. In logarithmic time, information is propagated up and down the tree to determine the position and length of the longest match. More precisely, if processor i detects a match, it checks with each of its neighboring processors. If the succeeding processor $i + 1$ did not match, then processor i sends a message to its parent processor in the binary tree signaling that it has the first character of a matching string. Similarly, if processor i's preceding neighbor did not match, i flags its parent processor that it is at the end of a match. These signals propagate up the tree until some processor k is able to pair up a start and an end symbol. Processor k then calculates the match length and returns the information to the processor (processor j) holding the first character in the match. If the new match length exceeds the existing longest match beginning at that character, the match position is assigned the processor number j and the length register is updated. An encoding example based on the tree-connected support structure is given in Figure 9.

The second match position and length updating scheme avoids some of the VLSI layout concerns, such as long edge lengths, at the expense of an increased logic delay of $O(\sqrt{N})$. Processors are placed in an $O(\sqrt{N}) \times O(\sqrt{N})$ grid with constant length connections and additional system cycles are introduced to spread information among non-neighboring processors. If the maximum length of a target string is limited to some constant k, the logical delay can be bounded by k.

Decoding of the systolic dictionary model expands all pointers by employing a series of $O(N)$ processors. Since all pointers are to locations less than N characters away, the N most recently decoded characters are stored in the pipe. The pointer (*position, length*)=$(p,\ l)$ is decoded by concatenating the characters stored in processor p to processor $p - l + 1$. The array is augmented with two additional pointers which aid in switching from different modes in the decoding process. Similar to encoding, decoding proceeds in blocks of N characters. Before entering the pipe, the pointer (*position, length*)=$(p,\ l)$ is expanded into the sequence of integers $p, p+1, \ldots, p+l-1$. The expanded encoded message consisting of characters and digits is fed into the pipe on every other system cycle. After each cycle, the input shifts left and each processor compares its identification number to the input. If the input item is an integer equal to the processor number, the processor replaces the integer by the contents of its dictionary entry. After N cycles, each processor replaces its dictionary entry with the contents of its input register.

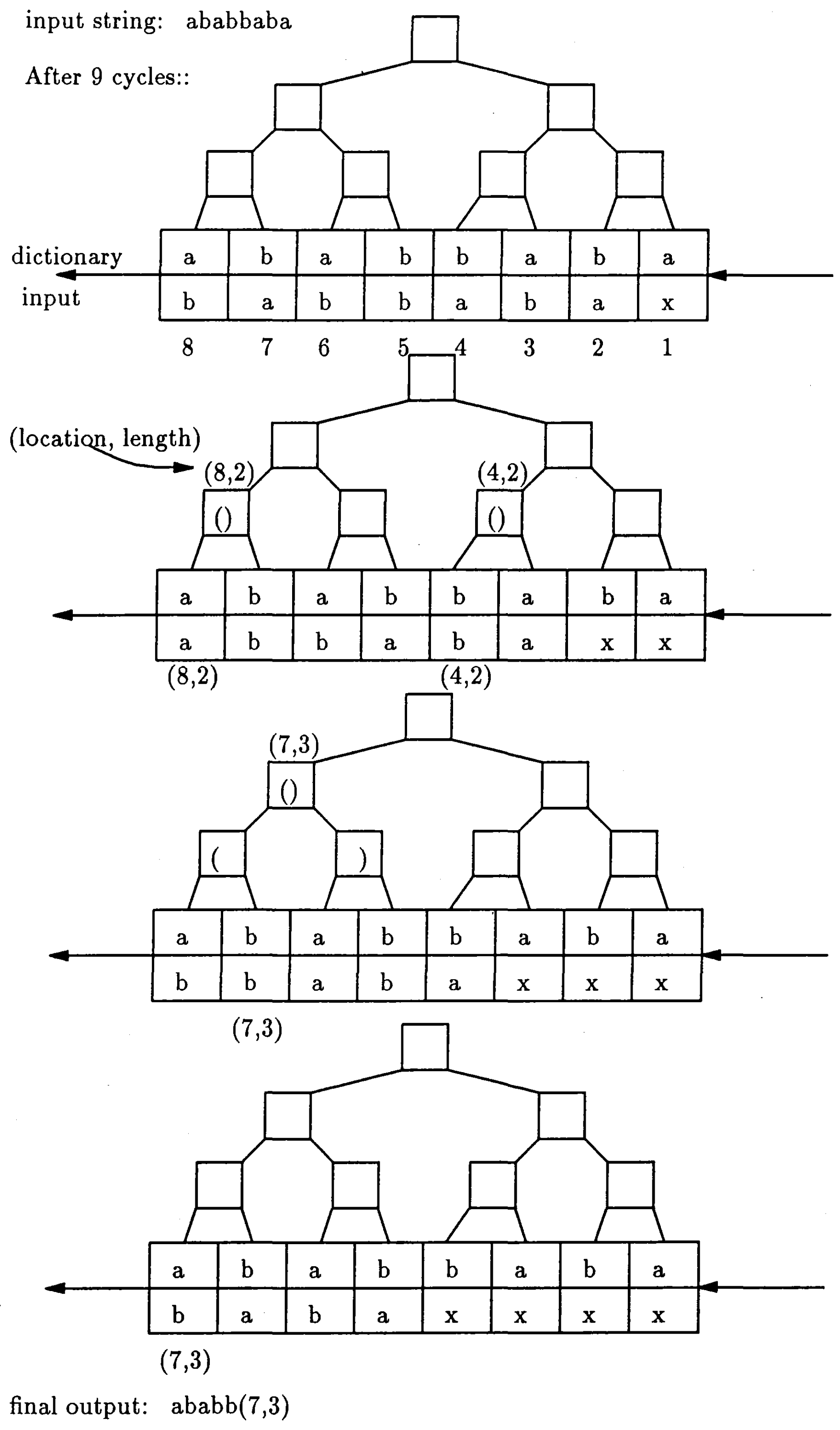


Figure 9

Encoding example on a tree-connected systolic sliding-window design

Like the static dictionary model, the Gonzales-Smith and Storer architecture for the sliding window model requires that the speed of the chip and the rate of the communication channel guarantee that additional data does not arrive at a processing element prior to it having available space. Hence, any improvements to the system performance that impact the data transfer rate may force the redesign of many system components. Another disadvantage is the communication and logical delays associated with the maintenance of match information. This design can, however, be implemented in VLSI straightforwardly, but details of the appropriate system parameters, such as array dictionary size specifications, are not discussed.

A different systolic data compression design built from a systolic array and binary trees is described by Zito-Wolf [Z90A]. In this design, unlike the Gonzales-Smith and Storer designs in which the data, dictionary, and output all flow through the systolic array, the data stream and dictionary are separated and longest match decisions are made by the tree processors. The dictionary is stored in the systolic array and compression is performed in two steps. First, the maximal match *ending* at each character is computed by making each input character simultaneously available to every processor via a broadcast tree of logarithmic depth, and identifying the largest match at each cycle using another tree connected collection of processors. This is in contrast to previous approaches which calculate the longest match *beginning* at a particular symbol. By not requiring global processor communication, all steps take unit time and system speed is unaffected by match length.

More recent architectures for sliding dictionary compression on the systolic array have attempted to remove the propagation delay introduced in previous designs. To be practical, data compression components must operate on-line and at high input bandwidths. On-line compression of an unbounded input requires time proportional to the input length and space proportional to the size of the dictionary. Henriques and Ranganathan investigated VLSI implementations, using CMOS technology, of a systolic architecture for sliding window dictionary compression [HR90]. They describe an on-line linear time and linear size systolic compression system. Moreover, they argue that a buffer size of 256 is most reasonable for VLSI implementation. This is based on experimental observations and the fact that the buffer size determines the pointer length which impacts the codeword size and ultimately the compression ratio. Although the overall system time is linear, additional clock cycles are used to propagate the length and match information among the processors. Also, for a system of N processors, only a single maximum length match is calculated for each block of N characters. How this impacts the compression effectiveness is not discussed, but it seems that this limited use of substring replacement may have a negative impact of compression. Unfortunately, no empirical results are given.

Zito-Wolf describes a bi-directional real-time systolic architecture for sliding window data coding that processes a character on every system cycle [Z90B]. Encoding is performed in two stages. The first stage is conducted on a systolic array which transforms the input into a stream of maximal matches. That is, for every input character a pair (*location, length*) is computed, identifying the longest match *ending* at that character. During the second stage, the array output is directed to a serial processor which extracts a sequence of matches that cover the input. The compression time is not only linear in the size of the input but also requires only a single clock cycle to process each character. More importantly, unlike the Gonzales-Smith architecture, the clock cycle is bounded and independent of the dictionary size. Using a 40Mhz clock, the system processes at a high-bandwidth of 300 million bits per second. As with other systolic implementations, the architecture is modularly expandable which allows for larger applications.

Like the static dictionary model, the Gonzales-Smith and Storer architecture for the sliding window model requires that the speed of the chip and the rate of the communication channel guarantee that additional data does not arrive at a processing element prior to it having available space. Hence, any improvements to the system performance that impact the data transfer rate may force the redesign of many system components. Another disadvantage is the communication and logical delays associated with the maintenance of match information. This design can, however, be implemented in VLSI straightforwardly, but details of the appropriate system parameters, such as array dictionary size specifications, are not discussed.

A different systolic data compression design built from a systolic array and binary trees is described by Zito-Wolf [Z90A]. In this design, unlike the Gonzales-Smith and Storer designs in which the data, dictionary, and output all flow through the systolic array, the data stream and dictionary are separated and longest match decisions are made by the tree processors. The dictionary is stored in the systolic array and compression is performed in two steps. First, the maximal match *ending* at each character is computed by making each input character simultaneously available to every processor via a broadcast tree of logarithmic depth, and identifying the largest match at each cycle using another tree connected collection of processors. This is in contrast to previous approaches which calculate the longest match *beginning* at a particular symbol. By not requiring global processor communication, all steps take unit time and system speed is unaffected by match length.

More recent architectures for sliding dictionary compression on the systolic array have attempted to remove the propagation delay introduced in previous designs. To be practical, data compression components must operate on-line and at high input bandwidths. On-line compression of an unbounded input requires time proportional to the input length and space proportional to the size of the dictionary. Henriques and Ranganathan investigated VLSI implementations, using CMOS technology, of a systolic architecture for sliding window dictionary compression [HR90]. They describe an on-line linear time and linear size systolic compression system. Moreover, they argue that a buffer size of 256 is most reasonable for VLSI implementation. This is based on experimental observations and the fact that the buffer size determines the pointer length which impacts the codeword size and ultimately the compression ratio. Although the overall system time is linear, additional clock cycles are used to propagate the length and match information among the processors. Also, for a system of N processors, only a single maximum length match is calculated for each block of N characters. How this impacts the compression effectiveness is not discussed, but it seems that this limited use of substring replacement may have a negative impact of compression. Unfortunately, no empirical results are given.

Zito-Wolf describes a bi-directional real-time systolic architecture for sliding window data coding that processes a character on every system cycle [Z90B]. Encoding is performed in two stages. The first stage is conducted on a systolic array which transforms the input into a stream of maximal matches. That is, for every input character a pair (*location, length*) is computed, identifying the longest match *ending* at that character. During the second stage, the array output is directed to a serial processor which extracts a sequence of matches that cover the input. The compression time is not only linear in the size of the input but also requires only a single clock cycle to process each character. More importantly, unlike the Gonzales-Smith architecture, the clock cycle is bounded and independent of the dictionary size. Using a 40Mhz clock, the system processes at a high-bandwidth of 300 million bits per second. As with other systolic implementations, the architecture is modularly expandable which allows for larger applications.

4.3. Dynamic Dictionary Coding on the Systolic Array

Dynamic dictionary compression systems utilize an evolving dictionary that adapts to changes in the input characteristics. Usually, dynamic approaches achieve superior compression results over static and semi-adaptive methods. There are a number of different dynamic approaches, all of which must include two basic strategies: match selection and dictionary update procedures. Dynamic dictionary compression implemented on the systolic array is the focus of this section.

4.3.1. Identity Heuristic and Systolic Dynamic Dictionary Coding

In 1988, Storer introduced the first systolic implementation for the dynamic dictionary model [S88A, S88B]. The approach is similar to the Gonzales-Smith and Storer implementation for the static dictionary model with additional specifications for updating the dictionary. As in the static design, a greedy approach is used for match selection. Dictionary maintenance involves the determination of strings to be inserted or possibly deleted. Candidate strings for insertion are derived from the concatenation of two previous matches (this is referred to as the identity update heuristic). Two separate dictionary pipes are employed, each initialized to contain the coding alphabet with one character per processor. Initially, compression begins using one of the two dictionaries and once the current dictionary becomes full, additional space is made available by swapping in the other (empty) dictionary. Later, when the dictionary again becomes full, the roles of the two dictionaries are reversed.

If N is the size of the dictionary, encoding is performed on a systolic pipe consisting of N processors, numbered 0 through $N-1$ from left to right. If A is the size of the input alphabet, processors 0 through $A-1$ are assigned the characters of the source alphabet and processors A through $N-1$ are capable of storing a pair of pointers. A flag bit in each processor is used to delimit the current dictionary. Initially, the flag bit in processor 1 is the only one set. The processor holding the flag is designated to "learn" the next new dictionary entry. All of the processors to the left of the learning processor contain dictionary entries, while processors to the right are empty. The input stream enters from the left and, as in the static dictionary systolic implementation, whenever a prefix of the input stream matches the contents of a processor, the string is replaced by the processor's number. The dictionary is updated by assigning the first pair of pointers to enter the learning processor to the dictionary entry stored in the learning processor and then passing the flag to the next processor. When processor $N-1$ receives a dictionary item, the signal is sent indicating that the dictionary is full. At this point, control is shifted to the empty dictionary and the current dictionary is flushed out.

As for encoding, decoding utilizes a pipe of size N and data enters the pipe from the left and exits on the right. However, the processors are numbered $N-1$ to 0, with 0 being the rightmost processor. Processors 0 through $A-1$ are initialized to contain the source alphabet and processor A starts out as the learning processor. Expansion is carried out as in the static dictionary system; that is, whenever an input substring arrives at a processor with index equal to the input, it is replaced by the word stored in the processor.

The identity heuristic for updating the systolic dictionary is closely related to a serial update heuristic which augments the dictionary with the concatenation of the previous match and the current longest match. The Storer implementation builds larger dictionary strings from two smaller ones. Storer compares the performance of the serial and systolic designs and finds that the difference in compression effectiveness is insignificant [S88A, S88B]. Storer hypothesizes that the systolic learning of the dictionary is superior to serial learning when compression is performed on a systolic array. This conjecture is based on an experiment which constructed a dictionary using the serial identity heuristic and then compressed a number of files using a serial algorithm and a systolic static dictionary simulator.

In many cases, the compression savings achieved by the parallel version was 10 to 15 percent more than that achieved by the serial algorithm.

Storer and Reif present a systolic real-time architecture based on a modified version of the identity heuristic of earlier designs [SR90]. The dictionary update heuristic, instead of entering the concatenation of two previous matches, adds the concatenation of two strings only if neither pointer was adopted by the preceding processor. A prototype VLSI chip for their design was built using a systolic coding pipe of 4,096 processing elements and a 25Mhz clock capable of operating at 300 million bits per second.

4.3.2. Move-to-Front Compression on the Systolic Array

Thomborson and Wei investigate systolic implementations of dynamic move-to-front coding algorithms [TW89]. In general, a move-to-front compression scheme maintains a self-organizing list of target strings (applying the move-to-front list maintenance heuristic) and encodes the table indexes using a statistical code. Huffman and arithmetic compression for index coding assign short codewords to positions near the front of the list. When a symbol is transmitted, the code corresponding to its current table position is output and the symbol is moved to the front of the list. Currently, move-to-front compression consists of two major algorithmic variants. The simpler procedure permutes a byte-level fixed-length list of symbols and the other defined-word approach divides the input stream into "words" and transmits words by a move-to-front code [BSTW86, E87]. For example, a byte-level move-to-front code might maintain a target list of 256 entries corresponding to the 256 possible values of an 8-bit ASCII byte. Such a system achieves compression savings of 30% to 40% on text files [TW89]. Defined-word methods often provide higher compression savings of 48% to 75%.

Systolic implementations of a fixed-table-size move-to-front system are composed of two separate chains of processors, one for encoding and the other for decoding. The sequential encoder permutes a table of fixed-length 2^k. A systolic byte-level encoder uses a linear array of 2^k processing elements. The i^{th} processing element, $1 \leq i \leq 2^k$, stores the target symbol t_i which is currently in the i^{th} position of the fixed-length table. The input data stream enters the array at processing element 1, flows through the array, and is output by processing element 2^k.

Encoding of source character a proceeds as follows. Symbol a is input to processing element 1. If a matches t_1 then the codevalue '1' is passed to processing element 2 signifying that a appears in the first position in the list. Eventually the codevalue '1' is output as the encoding of a. If a is not equal to t_1, a is copied into register t_1 and the previous contents of t_1 are transmitted to processing element 2 for deposit in location t_2. That is, a is placed at the front of the list, and the remainder of the list is bumped back. General processing element i receives a 4-tuple $(a, u, p, flag)$ from its neighboring processor $i - 1$, where a is the source character, u is the table symbol being moved down in the list, p is a's list index, and *flag* is set when the list needs further updating. If *flag* is set and a differs from t_i, u is copied into t_i and $(a, t_i, p, TRUE)$ is passed to processing element $i + 1$. If *flag* is set and a matches t_i, u is copied into t_i and $(a, u, i, FALSE)$ is transmitted. Otherwise, input $(a, u, p, flag)$ passes through processing unit i, unchanged. Figure 10 depicts the encoding of the string "architecture" for a systolic encoder consisting of 8 processing elements.

A string of k-bit codes, corresponding to the list positions of the input characters, is output by the encoding array and fed into a fixed-to-variable-length coding system. List positions near the head of the list are assigned short codewords. Thomborson and Wei experimented with various tail-end encoders and conclude that higher compression ratios can be obtained by using a dynamic fixed-to-variable-length encoder sensitive to changes in locality of reference in the source

Initial	8	7	6	5	4	3	2	1	Proc. Elem. ID
Configuration	e	i	r	a	u	t	h	c	Dict. Entry
									⟵ architecture
After 1	8	7	6	5	4	3	2	1	
cycle	e	i	r	a	u	t	h	a	
								(a,c,0,T)	⟵ rchitecture
After 2	8	7	6	5	4	3	2	1	
cycles	e	i	r	a	u	t	c	r	
							(a,h,0,T)	(r,a,0,T)	⟵ chitecture
After 3	8	7	6	5	4	3	2	1	
cycles	e	i	r	a	u	h	a	c	
						(a,t,0,T)	(r,c,0,T)	(c,r,0,T)	⟵ hitecture
After 4	8	7	6	5	4	3	2	1	
cycles	e	i	r	a	t	c	r	h	
(c matches pe 3)					(a,u,0,T)	(r,h,0,T)	(c,a,0,T)	(h,c,0,T)	⟵ itecture
(a matches pe 5)									
After 5	8	7	6	5	4	3	2	1	
cycles	e	i	r	u	h	a	c	i	
				(a,a,5,F)	(r,t,0,T)	(c,c,3,F)	(h,r,0,T)	(i,h,0,T)	⟵ tecture
After 9	8	7	6	5	4	3	2	1	
cycles	e	i	u	a	c	i	e	t	
5 ⟵	(r,r,6,F)	(c,c,3,F)	(h,h,4,F)	(i,t,0,T)	(t,r,0,T)	(e,h,0,T)	(c,t,0,T)	(t,c,0,T)	⟵ ture
After 16	8	7	6	5	4	3	2	1	
cycles	u	r	i	c	t	u	r	e	
3 ⟵	(u,a,0,T)	(r,h,0,T)	(e,e,5,F)						
After 17	8	7	6	5	4	3	2	1	
cycles	a	h	i	c	t	u	r	e	
8 ⟵	(r,r,7,F)	(e,e,5,F)							
After 18	8	7	6	5	4	3	2	1	
cycles	a	h	i	c	t	u	r	e	
7 ⟵	(e,e,5,F)								

COMPLETE OUTPUT: 5,6,3,4,7,6,8,5,3,8,7,5

Figure 10

A fixed-length systolic encoder

[TW89]. Their empirical findings suggest that their fixed-to-variable code converter gives rather poor compression savings (11% to 22%) but can perform at a high bandwidth. Dynamic Huffman coding provides better compression savings (19% to 38%) but operates at a limited bandwidth.

A systolic byte-level decoder also requires a linear array of 2^k processing elements. Instead of storing the i^{th} element in the list, processing unit i reserves the k-bit index q_i, $1 \leq i \leq 2^k$, corresponding to the list position of the character with representation i. For ASCII codes, q_i is the table index of the entry with ASCII value i. During decoding of symbol b, processing element i receives input $(b,\ r)$ from processing element $i-1$, where r is the value of the decoded symbol. If b equals q_i then q_i is set to '1' and r is assigned i. That is, the symbol with representation i is moved to the first list entry and b is decoded as i. If b is greater than q_i then q_i is incremented by one to reflect the movement of character representation i deeper into the list. Figure 11 depicts the behavior of the systolic move-to-front decoder.

Both the encoder and decoder are systolic arrays composed of simple processing elements and can be implemented fairly straightforwardly. Thomborson and Wei describe encoder and decoder chips that can operate at an input bandwidth of 40 Mbytes per second [TW89]. The bandwidth of their design is dependent on the behavior of the front-end variable-to-fixed decoder. This is an advantage over the Gonzales-Smith and Storer design whose data transfer rate is determined by the systolic clock. For these systems, any improvement in the bandwidth may require changes to many system components. Also, Thomborson and Wei report that their systolic implementation requires fewer processing elements than the Gonzales-Smith and Storer VLSI implementation and improves the compression speed by a factor of three.

As mentioned earlier, defined-word schemes provide better compression than byte-level methods. The most notable scheme, BSTW compression, is due to Bentley, Sleator, Tarjan and Wei [BSTW86]. Initially, the encoder list of the

Encoded message: 5,6,3,4,7,6,8,5,3,8,7,5

Dictionary:

Character	Representation
c	1
h	2
t	3
u	4
a	5
r	6
i	7
e	8

Initial	8	7	6	5	4	3	2	1	Processor
Configuration	8	7	6	5	4	3	2	1	List Position of character with rep.= proc. ID
After 1	8	7	6	5	4	3	2	1	
cycle	8	7	6	5	4	3	2	1	
								(5,♭)	← 6,3,4,. . .
After 3	8	7	6	5	4	3	2	1	
cycles	8	7	6	5	4	3	3	3	
						(5,♭)	(6,♭)	(3,♭)	← 4,7,6,. . .
After 4	8	7	6	5	4	3	2	1	
cycles	8	7	6	5	4	4	4	1	
					(5,♭)	(6,♭)	(3,1)	(4,♭)	← 7,6,8,. . .
After 6	8	7	6	5	4	3	2	1	
cycles	8	7	6	1	6	5	1	3	
			(5,5)	(6,♭)	(3,1)	(4,2)	(7,♭)	(6,♭)	← 8,5,3,. . .
After 7	8	7	6	5	4	3	2	1	
cycles	8	7	6	2	6	5	2	4	
		(5,5)	(6,♭)	(3,1)	(4,2)	(7,♭)	(6,♭)	(8,♭)	← 5,3,8,. . .
After 8	8	7	6	5	4	3	2	1	
cycles	8	7	1	3	6	6	3	5	
	(5,5)	(6,6)	(3,1)	(4,2)	(7,♭)	(6,♭)	(8,♭)	(5,♭)	← 3,8,7,. . .
After 9	8	7	6	5	4	3	2	1	
cycles	8	7	2	4	7	1	4	1	
5←	(6,6)	(3,1)	(4,2)	(7,♭)	(6,3)	(8,♭)	(5,1)	(3,♭)	← 8,7,5,. . .
After 12	8	7	6	5	4	3	2	1	
cycles	8	1	5	7	8	1	6	4	
2←	(7,7)	(6,3)	(8,♭)	(5,1)	(3,3)	(8,♭)	(7,♭)	(5,♭)	
After 14	8	7	6	5	4	3	2	1	
cycles	8	3	6	7	1	3	7	5	
3←	(8,♭)	(5,1)	(3,3)	(8,4)	(7,♭)	(5,♭)			
After 17	8	7	6	5	4	3	2	1	
cycles	3	5	1	8	3	4	7	5	
3←	(8,4)	(7,6)	(5,♭)						
After 18	8	7	6	5	4	3	2	1	
cycles	4	6	2	8	3	4	7	5	
4←	(7,6)	(5,♭)							

COMPLETE OUTPUT: 5,6,1,2,7,3,8,1,3,4,6,8 = architecture

Figure 11

A fixed-length systolic decoder

BSTW algorithm is empty. The first time a word is encountered, an *escape code* is transmitted followed by the word in cleartext. The word is entered into the move-to-front table. Subsequent occurrences of the word are encoded by the word's list position. The BSTW scheme compresses the cleartext and list indexes applying two separate codes.

The implementation of the byte-level encoder and decoder systolic arrays can be modified to allow for word-based compression by storing words rather than single characters in the processing elements. With a 3-bit count field, each variable-length word of 7 or fewer ASCII characters is encoded in 59 bits. However, this design requires 127 bits to represent the 4-tuples (2 words at 59 bits each, 8-bit table index, and 1-bit flag) processed by the encoder and therefore forces an unreasonably high number of 254 input/output pins per processing element. In addition, this approach suffers from an increased clock cycle to perform word comparisons. By reducing the maximum word-length, the pin requirements are lessened but at the expense of decreased compression effectiveness.

Thomborson and Wei investigate an alternative system which approximates the BSTW procedure on the systolic array [TW89]. The idea is to map variable-length words to an 8-bit hashcode using a hardwired hash table. These 8-bit codes are entered into the move-to-front list of target strings and manipulated as in the byte-level systolic encoder and decoder arrays. A closed hashing scheme with no collision resolution is used to obtain a high-speed, high-bandwidth design. These performance improvements, however, come at the expense of poorer compression performance. Unlike the BSTW algorithm in which the least-recently-used target word "falls" off of the end of the list, the hashing approach randomly eliminates list words. This random behavior of the systolic design yields compression savings ranging from 25% to 65%.

A systolic encoder, based on an 8-bit hashing scheme, uses an array of 239 processing elements.[10] Each processing element i stores a word w_i and a hashcode h_i. A word W is initially parsed from the source stream and hashed to an 8-bit hashcode w. If the hash table entry with index w is different from word W, the entry is overwritten by word W and an escape code is output. If w matches the entry, the hash component outputs the hashcode w. The hash indexes and escape codes are input into the move-to-front systolic encoder. Encoding and list evolution are identical to the byte-level encoder. Two independent fixed-to-variable code converters are used to compress the pipeline output, one for clear text and the other for fixed-to-variable index coding. Decoding of hash indexes closely mimics the byte-level decoder with the addition of hash table manipulation and hash index to source word conversion.

5. Multiple Data Compression

The static and dynamic data compression methods in Sections 3 and 4 employ a single coding scheme that manipulates the data in parallel. The systolic array implementations pipeline the coding table to decrease compression overhead. Parallel code construction speeds up codeword creation by building the codeword tree in parallel. Each of these methods has its advantages and disadvantages and is designed to improve compression speed. Alternatively, combining multiple data compression techniques works to obtain greater compression savings. Competitive parallel processing and pipelining of compression algorithms apply parallelism to data compression by utilizing multiple compression methods operating in parallel.

A pipelining data compression system combines two or more coding algorithms to compress data more effectively than the individual methods performing in isolation. The approach is to use a succession of compression techniques to

[10] Using an 8-bit code and allowing for cleartext escape codes for each possible word length requires a table of size $2^8 - 2^3 = 248$. To improve hash function performance, 239, the largest prime less than 248, is chosen.

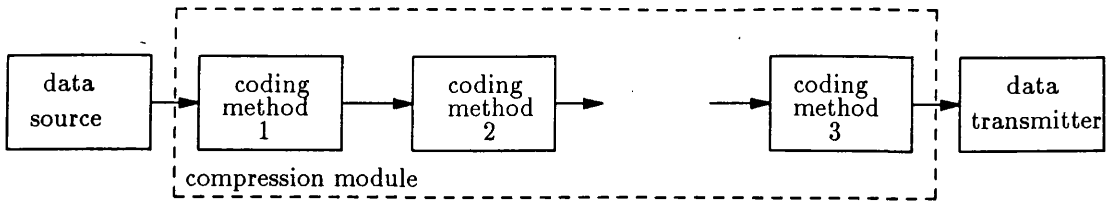


Figure 12

Pipeline of compression techniques to improve compression ratio

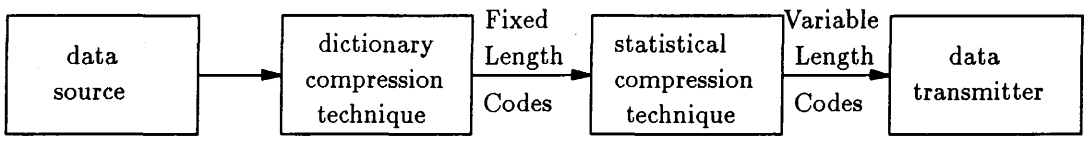


Figure 13

Example of a pipelined compression scheme

improve the compression ratio (see Figure 12). The selection of appropriate coding methods and their optimal positioning in the compression pipeline influences the system performance. Some data compression methods, such as Huffman coding, take advantage of character redundancy while others, including dictionary methods, profit from string repetition. Bailey and Mukkamala observe that if the input contains string redundancy, it also exhibits character redundancy [BM90]. They also note that the fixed length codes that are produced by a string parsing algorithm may contain multiple copies of the same codeword. By directing the fixed length output of the dictionary compression algorithm into a prefix coding element, additional compression is achieved (see Figure 13). Based on empirical investigations of 2-stage methods, pipelined data compression algorithms significantly increase compression savings. The major disadvantage of pipelining methods is the overhead required to run a sequence of sequential methods.

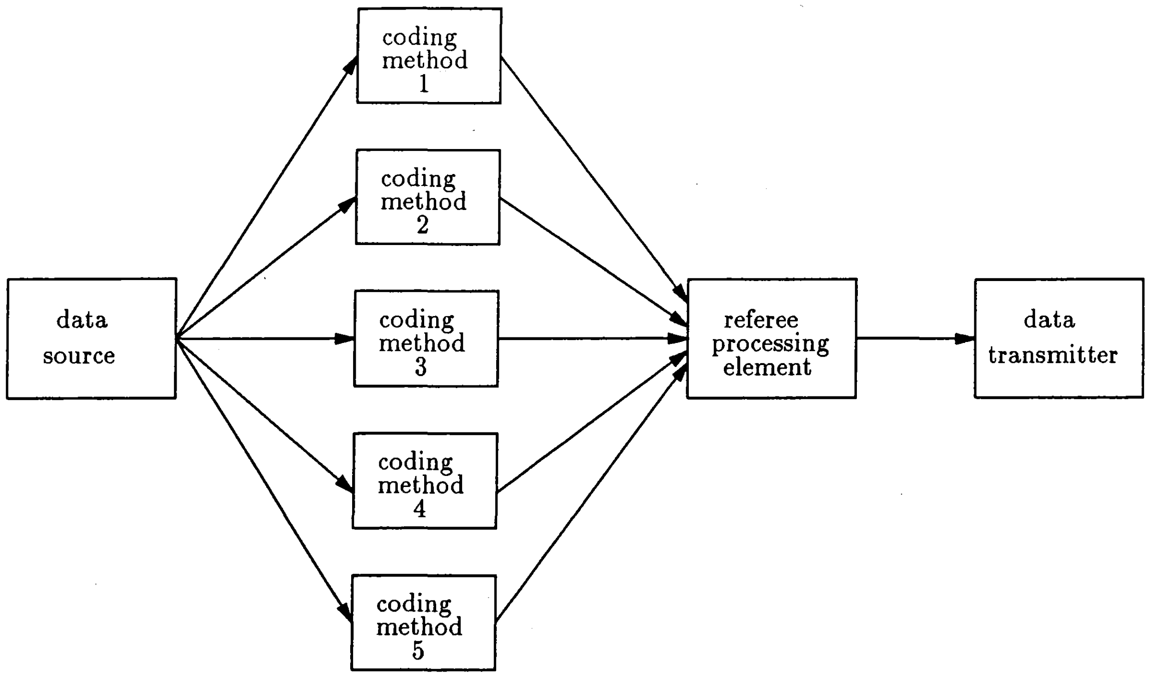


Figure 14

Multiple compression techniques competing for best compression performance

Competitive parallel processing for data compression employs several processors, each simultaneously executing a different data compression method [C90]. In this MISD parallel system, the output stream of the processor achieving the best compression savings is selected by the referee processor and transmitted. This is illustrated in Figure 14. Information is relayed with the coded package to enable the appropriate decompression processor.

6. Future Research

There are a number of important questions that remain unanswered in the area of parallel data compression. In this section, we suggest possibilities for future investigation.

As this survey has outlined, known results can be divided into the broad classes of statistical and dictionary-based compression. Most of the work in statistical methods is under the PRAM model of parallel computation and focuses on the parallel construction of trees and related issues. More practical approaches in other parallel models are necessary. For example, can a minimum-redundancy prefix code be found efficiently using a systolic architecture? Also, there are no known parallel designs for adaptive statistical coding methods, such as adaptive Huffman coding and arithmetic coding. Given the better compression results of dynamic methods, these are important areas needing attention.

The work in parallel dictionary coding has been limited primarily to the systolic array. Dictionary compression systems need to be developed for alternative parallel models, such as the hypercube.

Teng suggests further investigation of randomized and probabilistic algorithms for minimum-redundancy prefix coding [T87]. Also, optimal solutions for the general and alphabetic versions of the Huffman coding problem are not known and warrant further research. Another unanswered question is whether there exists a poly-logarithmic time, sub-quadratic processor algorithm for the Huffman tree problem. A variation of Huffman coding is the length-limited Huffman coding problem which creates a code from a sequence of probabilities restricted to some maximum code length. Parallel construction of length-limited Huffman codes remains an open problem.

Thomborson and Wei give a systolic move-to-front (see Section 4.3.2) compression system using a multiple-mode fixed-to-variable tail-end code converter that operates at a high input bandwidth [TW89]. Unfortunately, the overall compression is significantly worse than Huffman encoding. They suggest that the development of a high-bandwidth Huffman encoder is an interesting area for future research.

The discussion in Section 5 illustrates the impact of multiple parallel schemes, such as a pipelined succession of data compression systems or a competitive collection of methods operating simultaneously, on the overall compression ratio. The enhanced compression comes at the expense of additional hardware expenditures. Future work addressing issues of performance, feasibility, and suitability of these aggregate designs is needed.

Context modeling is a promising new approach to data compression which uses the preceding few characters of the input to predict and therefore estimate the probability of the next input character [BCW90]. For instance, in isolation, the probability of the letter "u" occurring is very low. However, if the preceding character is a "q" the probability of the next letter being a "u" approaches 1. Context-modeling has not been addressed in the parallel setting.

References

[AKLMT89] Atallah, M. J., Kosaraju, S. R., Larmore, L. L., Miller, G. L., and Teng, S.-H. Constructing trees in parallel. In *Proceedings 1989 ACM Symposium on Parallel Algorithms and Architectures*, Sante Fe, New Mex., ACM, New York, 1989, pp. 283–290.

[BM90] Bailey, R. L. and Mukkamala R. Pipelining data compression algorithms. *The Computer Journal 33*, 4 (1990), 308–313.

[BCW90] Bell, T. C., Cleary, J. G., and Witten, I. H. *Text Compression*, Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[BSTW86] Bentley, J. L., Sleator, D. D., Tarjan, R. E., and Wei, V. K. A locally adaptive data compression scheme. *Commun. ACM 29*, 4 (April, 1986), 320-330.

[C90] Competitive parallel processing for compression of data. *NASA Tech Briefs 14*, 2 (Feb., 1990), 32–33.

[E87] Elias, P. Interval and recency rank source coding: two on-line adaptive variable-length schemes. *IEEE Trans. Inf. Theory IT-33*, 1 (Jan., 1987), 3–10.

[F49] Fano, R. M. *Transmission of Information*, M.I.T. Press, Cambridge, Mass., 1949.

[F66] Flynn, M. J. Very high-speed computing systems. In *Proceedings IEEE*, Vol. *54*, 1966, pp. 1901–1909.

[FW78] Fortune, S. and Wyllie, J. Parallelism in random access machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, ACM, New York, 1978, pp. 114–118.

[G78] Goldschlager, L. M. A unified approach to models of synchronous parallel machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, ACM, New York, 1978, pp. 89–94.

[GS85] Gonzalez-Smith, M. E. and Storer, J. A. Parallel algorithms for data compression. *J. ACM 32*, 2 (Apr., 1985), 344–373.

[HR90] Henriques, S. and Ranganathan, N. A parallel architecture for data compression. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, Dallas, Texas, 1990.

[H52] Huffman, D. A. A method for the construction of minimum-redundancy codes. *Proceedings IRE 40*, 9 (Sept., 1952), 1098–1101.

[KP90] Kirkpatrick, D. G. and Przytycka, T. Parallel construction of near optimal binary trees. In *Proceedings 1990 ACM Symposium on Parallel Algorithms and Architectures*, Crete, Greece, 1990.

[LP91] Larmore, L. L. and Przytycka, T. Personal communication, 1991.

[L78] Lea, R. M. Text compression with an associative parallel processor. *Computer J. 21*, 1 (Jan., 1978), 45–56.

[LH87] Lelewer, D. A. and Hirschberg, D. S. Data compression. *ACM Comp. Sur. 19*, 3 (Sep., 1987), 261-296.

[LL85] Lipton, R. J. and Lopresti, D. A systolic array for rapid string comparison. In *Proceedings Chapel Hill Conference on VLSI*, 1985.

[MRK85] Miller, G. L., Ramachandran, V., and Kaltofen, E. Efficient parallel evaluation of straight-line code and arithmetic circuits. Technical Report. University of Southern California (1985).

[MR85] Miller, G. L. and Reif, J. H. Parallel tree contraction and its application. In *Proceedings of the Twenty-Sixth Annual Symposium on Foundations of Computer Science*, IEEE, Portland, Oregon, 1985, pp. 478–489.

[M88] Milutinovic, V. M., Ed. *Computer Architecture: Concepts and Systems*, North-Holland, New York, 1988.

[Q87] Quinn, M. J. *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, New York, 1987.

[RPE81] Rodeh, M., Pratt, V. R. and Even, S. Linear algorithm for data compression via string matching. *J. ACM 28*, 1 (Jan., 1981), 16–24.

[SR90] Storer, J. A. and Reif, J. H. A parallel architecture for high speed data compression. In *Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation*, Fairfax, Vir., IEEE Computing Society Press, Washington, D. C., 1990.

[S88a] Storer, J. A. *Data Compression Methods and Theory*, Computer Science Press, Rockville, Maryland, 1988a.

[S88b] Storer, J. A. Parallel algorithms for on-line dynamic data compression. In *Proceedings of the IEEE International Conference on Communications: Digital Technology - Spanning the Universe*, IEEE Publishing, New York, 1988b, pp. 385–389.

[SS82] Storer, J. A. and Szymanski, T. G. Data compression in textual substitution. *J. ACM 29*, 4 (1982), 928–951.

[T87] Teng, S.-H. The construction of Huffman-equivalent prefix code in NC. *ACM SIGACT J. 18*, 4 (May, 1987), 54–61.

[TW87] Teng, S.-H. and Wang, B. Parallel algorithms for message decomposition. *J. of Parallel and Distr. Comp. 4* (1987), 231–249.

[TW89] Thomborson, C. D. and Wei, Belle W.-Y. Systolic implementations of a move-to-front text compressor. In *Proceedings 1989 ACM Symposium on Parallel Algorithms and Architectures*, Sante Fe, New Mex., ACM, New York, 1989, pp. 283–290.

[W84] Welch, T. A. A technique for high-performance data compression. *Computer 17*, 6 (June, 1984), 8–19.

[WNC87] Witten, I.H., Neal, R. M., and Cleary, J. G. Arithmetic coding for data compression. *Commun. ACM 30*, 6 (June, 1987), 520–540.

[Z90a] Zito-Wolf, R. J. A broadcast/reduce architecture for high-speed data compression. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, Dallas, Texas, 1990a.

[Z90b] Zito-Wolf, R. J. A systolic architecture for sliding-window data compression. In *Proceedings of the IEEE Workshop on VLSI Signal Processing*, 1990b.

[Z90c] Zito-Wolf, R. J. VLSI architectures for high-speed sliding dictionary data compression. Technical Report Number CS-90-149. Computer Science Department, Brandeis University, MA (1990c).

[ZL77] Ziv, J. and Lempel, A. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory 23*, 3 (1977), 337–343.

[ZL78] Ziv, J. and Lempel, A. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory 24*, 5 (1978), 530–536.