Parallel Text 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, REVISED

January 26, 1993

Abstract

Much of the study of text compression attempts to maximize the compression or reduction in the size of a body of text by removing redundancy. Another focus is on speed and on performing compression rapidly. Parallelism offers a myriad of resources for meeting these goals. This paper surveys algorithms, architectures and implementations for parallel text compression. Related concepts from text compression and parallel computation are discussed and a framework for evaluating parallel compression schemes is developed. This framework delineates parallel methods that boost system speed by compressing text concurrently, and approaches that employ multiple compression techniques to improve compression. Also, 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 increase the effective density of transmitted or stored data. Traditionally, there has been a tradeoff between the benefits of employing data compression versus the computational costs required to compress and decompress data. Parallelism represents a means for increasing speed and compression effectiveness. Consequently, parallel compression is suitable for a wide range of applications including some where sequential compression does not meet performance criteria. The purpose of this paper is to present and analyze parallel text compression methods.

A common benchmark in software text compression systems is the UNIX[1] *compress* utility. *Compress* is based on a variation of Ziv-Lempel compression [ZL78] due to Welch [W84]. The UNIX *compress* utility provides compression savings of up to 80% at a relatively high input data rate 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 data rates of approximately 10 Kbytes per second on a 1 MIPS machine. In this survey, several parallel architectures and algorithms for text compression that achieve much higher data rates of 20 Mbytes, 40 Mbytes and 120 Mbytes per second are described.

The speed gained by parallelizing text compression schemes enhances their practicality and applicability. An alternative to a single parallel system is a multiprocessing approach that employs various compression methods to improve compression effectiveness. At the expense of system time and hardware resources, multiple compression algorithms, operating concurrently, can improve compression. While this use of parallelism may be less practical because of its resource demands, for certain applications the benefits of compression may outweigh the increased overhead.

[1] UNIX is a trademark of AT&T Bell Laboratories.

This survey of parallel text compression considers *lossless* compression techniques which operate under the constraint that the decompressed data must be identical to the original data stream. That is, lossless compression is completely reversible. Text compression, the focus of this survey, is usually restricted to lossless compression. Image processing is an example of an application that can tolerate inconsistencies between the original data and its decompressed 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.

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 [RS91].

Background concepts in text compression and parallel computation are provided in Section 1. Parallel compression systems based on concurrent manipulation of source data are described in Sections 2 and 3. Aggregate compression systems incorporating collections of compression methods to improve compression are discussed in Section 4. Finally, in Section 5, topics for future research and their relationship to known results are examined.

2. PRELIMINARIES

A brief introduction to parallel processing and text compression is provided in this section. The terms and assumptions necessary for a careful evaluation and comparison of parallel text compression methods are presented. For more detailed discussions of text compression in the sequential setting see [LH87], [S88A], [BCW90], and [W91]. Further insight into parallel computation is provided by [Q87], [GR88], and [L92].

2.1. Categorization of Compression Methods

The process of converting a text, such as a file on disk, a string in memory, or a stream of characters, into a compact representation is called *encoding* or *compression*. *Decoding* or *decompression* restores the original text from its compressed representation. There are a number of ways of classifying compression methods. This section describes the classifications that are relevant to parallel text compression.

Bell, Cleary and Witten draw a distinction between statistical and dictionary based compression [BCW90]. Methods where characters and phrases are compressed based on their probability or frequency are labeled *statistical* methods. Other techniques function by replacing large blocks of input with references to earlier occurrences of identical data. These *dictionary* methods, also called *textual substitution* and *Ziv-Lempel compression*, achieve compression by replacing phrases with references into some dictionary of phrases [ZL77, ZL78]. It is known that any practical dictionary compression system can be simulated by a statistical method to yield comparable compression [BCW90]. However, dictionary compression remains competitive because of its speed and simplicity [FG89].

Another classification splits compression into two separate parts: modeling and coding [BCW90]. Modeling defines the set of elements that may be considered and estimates the relative weighting or probability of each element. Coding uses these elements to produce the compressed output. Given the distinction above, statistical modeling, statistical coding, and modeling and coding for dictionary compression are further described below.

Compression schemes can be further categorized as operating off-line, on-line, or in real-time. An *off-line* model can manipulate and preprocess the entire input string prior to coding. In *on-line* models, neither the sender nor receiver can view all of the data at once; that is, data is constantly flowing through the encoder, being 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 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.

One final note on the categorization of compression methods. Robin Milner, the recipient of the 1991 Turing Award, emphasizes the importance of investigating computer science from both theoretical and practical perspectives [F93]. Many topics in parallel text compression have not been considered from both perspectives. The work in parallel statistical compression, for instance, is limited to theoretical models. A possible organization of the work in parallel text compression is to divide the area into theory and practice. Alternatively, this paper splits the area into statistical and dictionary compression approaches. This categorization is used to emphasize the missing links between practical and theoretical parallel text compression. Thus, this survey is intended not only to introduce readers to parallel text compression, but also to highlight the areas where little is known.

2.1.1. Modeling

Models can be *static, semi-adaptive*, or *adaptive*. Static modeling uses the same model for all texts. Semi-adaptive modeling employs a different model for each input file. The semi-adaptive model can be constructed during a preprocessing pass over the input and is transmitted to the decoder. Adaptive (or dynamic) modeling does not transmit a model explicitly. Rather, the adaptive model evolves and adapts to changes in the input as data is being encoded. After initializing the model, the encoder compresses one or more input characters based on the current model and subsequently updates the model. The decoder receives the input, decodes it and updates the model in the same way.

A statistical model provides the probability that a symbol or a string of symbols will occur. Context modeling, finite state models, grammar models, and recency models are the focus of current research in statistical modeling [BCW90, W91]. In dictionary compression, the dictionary model contains the set of defined strings. Typically, there is no estimation of relative probabilities.

Dictionary compression techniques are distinguished by their use and maintenance of the dictionary. The dictionary can be *static*, *semi-adaptive*, or *adaptive.* A static dictionary is created before encoding or decoding begins and must remain fixed. Semi-adaptive dictionary compression uses a different dictionary for each text encoded. Better compression is achieved by adaptive (also called dynamic) dictionaries that allow additions, deletions, and changes to the collection of referenced strings during compression.

Adaptive dictionaries are further distinguished by whether there are restrictions on which substrings can be referenced and on how far back an index can point. Variations of these restrictions yield different compression methods (see [BCW90]). Popular methods are based on the works of Ziv and Lempel [ZL77, ZL78]. Their first approach limits the pointers to a fixed-size window preceding the current position in the input and places a maximum length on dictionary entries [ZL77]. The dictionary is implicitly represented by a fixed-sized window that slides continuously over the input. Pointers are (*position, length*) pairs that indicate a substring of the window. These methods are referred to variously as LZ77, LZ1, and *sliding-window* compression.

Ziv and Lempel's second approach (often called LZ78 or LZ2 compression) parses the input into phrases and stores the dictionary explicitly. The dictionary is initially empty and new entries are derived from the longest matching dictionary entry concatenated with a character from the input. That is, for input message $w = w_1 w_2 \cdots w_n$ and longest matching prefix p of length l, the entry pw_{l+1} is added to the dictionary. The input is encoded as a dictionary index plus the extra input character [ZL78]. The UNIX *compress* utility implements a variation of LZ78 due to Welch (referred to as LZW) and initializes the dictionary with the input character set [W84]. Since pointers are not restricted to a fixed-sized window, the dictionary (and consequently the size of pointers) can continue to grow without bound. However, in practice, finite memories mandate policies for dealing with a full dictionary.

There are a number of other dictionary methods that do not have roots in the Lempel and Ziv work. These approaches include methods that use punctuation to define words or consider words of only fixed length. These have not been considered in the parallel setting. Consequently, they are not described here and the reader is referred to [LH87], [S88A], [BCW90], and [W91].

2.1.2. Coding

A statistical coder assigns a code to each string based on the probabilities given by the model. For a static or semi-adaptive model, Huffman coding, Shannon-Fano coding and arithmetic coding attempt to assign short codes to frequently occurring input strings [F49, H52, FW78]. Dynamic Huffman coding and arithmetic coding are examples of statistical coders that work in conjunction with an adaptive model [K82, V87, RL79, L84, WNC87].

Codeword-based statistical coders replace input strings by codewords to obtain a more compact representation of the input. Huffman coding and Shannon-Fano coding are codeword-based. 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 file. For codeword-based coders, let $\Sigma = \{s_1, s_2, \ldots, s_n\}$ be the *source alphabet*. A *source message* is a sequence of characters over the alphabet Σ. 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 a *prefix* code if no codeword in $\mathcal{C}$ is a prefix of another codeword.

Most coding techniques can be classified as fixed-to-fixed. fixed-to-variable, variable-to-fixed, or variable-to-variable. This categorization distinguishes the restrictions on source message and output lengths of different mappings. Huffman coding is a fixed-to-variable method since fixed-length input strings are mapped to variable-length codes. Arithmetic coding is a variable-to-variable scheme.

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.[2] 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 concept of *entropy.* Entropy is a measure of the information content of a message. For an unpredictable source, entropy (information content) is high; for an orderly source, entropy is low. Formally, for source alphabet $\Sigma = \{s_1, s_2, \ldots, s_n\}$ with probabilities of occurrence $\{p_1, p_2, \ldots, p_n\}$ and distinct code $C = \{c_1, c_2, \ldots, c_n\}$, the expression[3] $\sum_{k=1}^{n} -p_k \lg p_k$ 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 multipied by the length of the source message. That is, since the length of a coded message must be sufficient to carry the information of the corresponding source message, entropy imposes a lower bound on codeword

[2] 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.

[3] In this paper, lg denotes the base 2 logarithm.

length [LH87]. A minimum-redundancy code minimizes the difference between the average codeword length and the entropy of the source.

Huffman coding generates a minimum-redundancy prefix code. The prefix code is equivalent to a full[4] binary tree with the source symbol probabilities associated with the leaves. To construct this code tree sequentially, 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 probabilities of the two original trees. Huffman's algorithm runs in $O(n \log n)$ time, where n is the size of the source alphabet. If the symbol probabilities are presorted, Huffman's method requires only linear time.

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 [RL79, L84, WNC87]. The method uses the probabilities of the source to successively narrow the interval used to represent the input. Ultimately, the interval is narrowed sufficiently so that the interval, or any number in it, represents only the source string being compressed. 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.

In dictionary compression, the coding step maps the index of the dictionary entry matching the prefix of the source message to some shorter representation. Indices can be coded using a fixed-to-variable-length representation of the integers, including Elias codes, Fibonacci codes, and start-step-stop codes [BCW90]. It is

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

also possible to use a statistical fixed-to-variable coder to assign shorter encodings to frequently occurring indices. This cascading of a dictionary compression methods into a statistical coder is considered in Section 4.

2.2. 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 that 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). The PRAM model assumes 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 compression methods presented in this paper.

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 processor 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. Multiple-Instruction, Single-Data (MISD) computers encompass systems of several processors, each executing a different instruction on identical input. The Single-Instruction, Multiple-Data (SIMD) model consists of a number of uniform processors, an interconnection network, and an associative memory. The synchronized processing elements (PEs) of an SIMD computer simultaneously perform the same operation on different data. These machines 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 computation is the parallel random access machine (PRAM). which is in the SIMD classification [FW78, G78]. The PRAM is a SIMD model consisting of a number of identical general-purpose sequential processors. Processors 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. Instructions are specified for particular processor indices and are executed by the corresponding processors. 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. Algorithms exist for simulating the PRAM with more realistic models [L92, pp 697–710].

Several variants of the PRAM differ in their handling of simultaneous reading and writing of the global memory. The weakest of these variants, the Exclusive-Read, Exclusive-Write (EREW) PRAM, prohibits concurrent reads and writes. 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. These various PRAM models 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. A *tree-connected* network restricts data movement to links between a processor and its parent (or children). In a tree-connected system with n processors, data communication takes at most logarithmic time. More detailed descriptions of these parallel models and how they are used to perform compression are given in Section 3.

A straightforward application of parallelism in text compression partitions the input into blocks and assigns a block to each processor. In parallel, each processor compresses its block and writes its compressed output in its appropriate position in the output stream (i.e., the order of the input blocks must be maintained in the compressed output). Since the sizes of the compressed blocks may differ, additional computation is required to determine where in the output stream a processor must write its output. The decoder requires similar information to correctly partition the encoded file into blocks for assignment to different decoding processors.

Another use of parallelism divides the input into blocks and begins by assigning a processor to each input character in the first block. That block is then compressed by the processors operating in parallel and using an initial model (for later blocks, a model is developed from earlier blocks). The model is updated before moving to the next block. Again, an additional computation is required to determine where to write the output and additional information is required by the decoder to assign compressed pieces to processors.

Measure	Name
c/o	Proportion remaining
$1-(c/o)$	Proportion removed
$100(c/o)$	Percentage remaining
$100(1-c/o)$	**Percentage removed**
$8(c/o)$	Bits per instance
$o/(8c)$	Instances per bit
o/c	Compression gain

Table 1

Alternative compression measures [W91]

2.3. Evaluation of Compression Methods

Several measures are used to evaluate and compare compression methods. Some measures attempt to describe the space reduction attained by compression. The common term *compression ratio* is not clearly defined and is consequently not used in this survey. Table 1 lists possible measures where o is the size of the original data and c is the size of the compressed data [W91]. In this paper, compression is given as *percentage removed.*

In the study of parallel complexity, problems are classified according to their use of time and processor resources. The class $\mathcal{NC}$ is the collection 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 [GR88]. The class $\mathcal{P}$ is the collection of problems with polynomial-time sequential algorithms. There are problems in $\mathcal{P}$ which do not seem to be readily parallelizable. These problems form the class of P-*Complete* problems. If an $\mathcal{NC}$ algorithm for any P-complete problem could be found then all problems in $\mathcal{P}$ would have similar $\mathcal{NC}$ solutions. Although it has not been proven, it is strongly believed that $\mathcal{P} \neq \mathcal{NC}$ [GR88].

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[5] $O(\text{Seq}(\mathcal{P})/\text{P})$ time using $O(\text{P})$ processors. Therefore, the work performed by an optimal algorithm is proportional to the time required by the fastest known sequential algorithm.

2.4. Motivation

The increasing availability of massively parallel computing architectures and the large volumes of scientific data being produced motivate the study of parallel data compression. 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]. Data compression has become an essential component of data storage and communication in these high-speed applications. The speed of the communication channel also impacts the performance of distributed computing systems. Compacting messages before transmission increases the effective data rate of the communication link. 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, improved efficiency in search operations on compressed files, and reduction in backup and recovery costs in computer systems. Parallel compression schemes which compress faster than sequential methods can improve these systems.

[5] *O-notation* represents an upper bound on the asymptotic behavior of a function.

3. PARALLEL STATISTICAL COMPRESSION

Statistical compression is composed of two tasks: modeling and coding. The model estimates probabilities of input symbols and the coder uses the probabilities to code the input into a compressed form. This section examines parallel approaches to statistical compression.

Section 2.1 describes parallel construction of codewords based on a static model of the input. Dynamic programming and matrix multiplication on the PRAM are the underlying mechanisms for these approaches. Section 2.2 considers parallel encoding of input characters by codewords.

3.1. Static Code Construction

A statistical codeword-based coder assigns codes based on probabilities of individual symbols. Static Huffman compression calculates character probabilities during a preprocessing pass over the source data. This information is used to assign codewords so that short codes correspond to high-probability symbols and longer codes are given to low-probability characters. The second pass compresses the source data by replacing input characters by their codewords.

3.1.1. 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 [GR88]. Then the values of $H(i,j)$ are given by the following recurrence relation:

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

This recurrence relation is the basis of a dynamic programming algorithm for sequential code creation. 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.

A sequential algorithm, implementing the above recursive definition, for building a minimum-redundancy prefix code can be sketched as [T87]:

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.

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. 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 to improve this 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. By considering a restricted form of the RAKE operation where a leaf is removed only if its siblings are leaves, any left-justified[6] 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 [AKLMT89]. Also, each iteration of Step 2 in Teng's sequential algorithm simulates the RAKE operation and can be done in $O(\log n)$ time using $n^3/\log n$ processors on the CREW PRAM model 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 probabilities $(p_1, p_2, \ldots, p_j)$ 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:

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

[6] 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.

The following sketch of the algorithm 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 $F(1,n)$ is 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. The corresponding tree and codewords can be generated from the computed H and F values within the same resource bounds using an approach similar to Teng's. 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, 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.

3.1.2. Huffman Coding as the Multiplication of Concave Matrices

In light of the sequential $O(n \log n)$ performance of Huffman's algorithm, the parallel solutions of the previous section are far from optimal. This section discusses an alternative approach which runs in $O(\log^2 n)$ time using $n^2/\log n$

processors [AKLMT89]. Huffman code construction can be formulated as a matrix multiplication problem. 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.* Specifically, for $n \times m$ matrix M, the following inequality holds for all $1 \leq i < k \leq n$, $1 \leq j < l \leq m$:

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

A recursive algorithm for multiplying concave matrices over the closed semiring $(min, +)$ runs in $O(\log n \log\log n)$ time using $n^2/\log n$ processors on the CREW PRAM [AKLMT89]. Allowing concurrent writing reduces the running time to $O((\log\log n)^2)$ time and $n^2/(\log\log n)$ processors. By taking advantage of the more efficient concave matrix multiplication, the Huffman code construction problem can be solved in $O(\log^2 n)$ time using $n^2/\log n$ processors. The solution 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 codes 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.

3.1.3. Other Parallel Constructions of Static Codes

A reduction of the Huffman tree problem to the Concave Least Weight Subsequence problem results in a new linear time sequential algorithm for a sorted sequence of probabilities and a more efficient parallel algorithm for the general case [LP91]. Given a concave triangular matrix of weights $\{w(i,j) | 0 \leq i < j \leq n\}$, 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.

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-minimum-redundancy code construction is derived from a reduction to the problem of constructing a tree given a monotonic sequence of leaf levels [AKLMT89]. Initially, the input probabilities $(p_1, p_2, \ldots, p_n)$ are sorted and a sequence of lengths $(l_1, l_2, \ldots, l_n)$ is 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 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 [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.

3.2. Static Coding

Once the codes are constructed, a static coder compresses the input by mapping input characters to codewords. A very fast implementation uses a table to store the codes and the mapping is performed by table lookup. On the PRAM model of computation with concurrent reading and limited concurrent writing, Huffman compression with precomputed static codes can be done in parallel [HV92A]. Since both encoding and decoding are done in parallel, the location of codes in the compressed file must be computable by the encoding processor prior to writing output and by the decoding processors before reading input. Thus, the main issue in parallel compression lies in accommodating the different codeword lengths.

One simple approach assigns an input character to each processor. Each processor computes the length of the codeword corresponding to its input character and before writing output performs a calculation to determine the location of its code bits in the output file. (This calculation is called a prefix sum operation and can be performed optimally in parallel [GR88].) When all processors are finished writing, each is assigned a new input character and processing continues. This approach, however, does not lend itself to parallel decoding since the decoding processors cannot parse the codewords from the encoded file in parallel.

A second approach supports parallel decoding by interleaving the output bits. Encoding processors operate as in the first approach but instead of writing output codewords contiguously in the output stream, each processor writes a single bit of its output at each step. Since encoding processors finish writing their output at different steps, each active processor must compute (using a prefix sum operation) its write location before outputting each bit. After completing its output, a processor remains inactive until all processors are finished. Decoding can be done in parallel since each processor can determine when its input codeword is complete and the same computation as used during encoding determines the location of the next bits for active processors. The drawback of this approach is the relatively time-consuming computations to determine bit locations in the compressed stream.

A third approach allows each processor to begin compressing a new input even though other processors may not have completed writing their output. Processing proceeds in phases and at the beginning of each phase the remaining input characters are reallocated among the processors. At the start of the first phase, all inputs are allocated equally among the processors. During each phase, encoding proceeds as in the second approach. That is, each processor maps its first allocated input to its codeword and outputs a bit (interleaved with output bits of other processors) at each step. After completing its output, a processor does not become inactive but begins processing its next assigned input. When a processor finishes *all* of its inputs it signals the other processors by writing to a common register and the phase ends. The output process is interrupted for partially completed processors. The next phase begins by reallocating the remaining untouched inputs among all of the processors and compression continues. During the final phase, no inputs remain for reallocation so processors become inactive as in the second approach and active processors must compute (again using a prefix sum operation) their appropriate output locations.

Decoding of prefix-coded messages and uniquely-decipherable-coded messages can be done in $O(\log n)$ time using $O(n/\log n)$ processors [TW87]. The optimal algorithm for the EREW PRAM is based on a reduction of the decompression problem to the problems of parallel finite-state automata simulation and the evaluation of prefix sums. An optimal parallel simulation algorithm for finite-state automata using dynamic expression evaluation and parallel tree contraction techniques completes the solution.

Huffman coding is restricted to represent each input character with an integral number of bits. This restriction forces Huffman coding to perform suboptimally for many probability distributions. Arithmetic coding, a non-codeword-based method, does not have this restriction and can represent an input character with less than 1 bit. Theoretically, arithmetic coding achieves minimum-redundancy compression. In practice, arithmetic coding is excessively slow and therefore approximations are of interest. *Quasi-arithmetic coding* approximates the intermediate intervals computed by an arithmetic coder using a finite number of states [HV92A, HV92B]. Look-up tables representing state transitions are precomputed resulting in a practical approximation. The reallocation approach for parallel Huffman coding can be used to perform quasi-arithmetic coding in parallel. Since a processor may complete its processing of an input in some state other than the initial state, additional measures are taken to force the processor back to the starting state.

4. PARALLEL DICTIONARY COMPRESSION

Dictionary compression (also referred to as Ziv-Lempel compression or textual substitution) removes data redundancy by replacing repeated input substrings by references (also called indices or pointers) to earlier copies of the identical substring [RPE81, SS82, ZL77, ZL78]. A dictionary of characters, words or phrases that are expected to occur frequently is maintained and a recurring substring is encoded by the index of its corresponding dictionary entry. Compression is achieved by choosing indices so that on average they require less space than the phrase they encode. This section considers dictionary compression in the parallel setting.

The encoder and decoder work in lockstep. The encoder repeatedly detects matches between the input and the dictionary, deletes the matched characters from the input, transmits the index of the dictionary entry, and updates the dictionary. The decoder repeatedly receives an index, outputs the corresponding dictionary entry, and updates the dictionary using the same update algorithm as the encoder.

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 is used to encode the input prefix. An *optimal* parsing of the input is a shortest possible sequence of dictionary indices such that the concatenation of the corresponding dictionary entries equals the input. To determine an optimal parsing, an encoder must perform two passes over the input or be capable of looking at arbitrarily large prefixes of the input. So although a greedy parsing strategy may not yield the optimal parsing, it is widely used.

In the parallel setting, the longest match step of a greedy parsing strategy 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 $N-1$ processing elements coordinate the results via signals that propagate up and down the tree in $O(\log N)$ time.

A natural use of parallelism breaks the input into pieces and compresses selected pieces in parallel. A recursive parallel algorithm with an exponential number of processors for performing a variation of LZ78 yields logarithmic performance by breaking the input into blocks and compressing blocks in parallel. The input is iteratively split in half and each half is compressed using a dictionary constructed during a sequential compression pass over an initial portion of each block [W92]. It is also possible to elicit fine-grain parallelism from the low-level software implementations of various algorithms.

Models of parallel computation which allow blocks of data to be read and written simultaneously are necessary to break the linear time lower bound mandated for handling input and output. Sublinear time methods for dictionary compression on parallel models of computation, where data is assumed to be made available at several processors in a single time unit, are considered in Section 3.1.

In the parallel VLSI environment, static, semi-adaptive, and adaptive dictionary schemes have been considered using a systolic array. A systolic array is composed of a collection of linearly connected processing elements (PEs) where each processing element contains local units for program control and storage. It is usually assumed that the local program control is simple (i.e., consists of a few operations) and the local storage is small (i.e., a few words). Time is partitioned into steps by a global clock so that the entire array operates synchronously. At each step, each PE receives input from its neighbors, inspects its local storage, performs the computation indicated by its local control, generates output and completes the step by updating its local storage.

One advantage of a systolic implementation is that a larger array can be fabricated by placing a sequence of PEs on a single chip, and then joining a series of chips on a board. Another benefit is that the length of interprocessor connections is constant and independent of the size of the array. On-line parallel static dictionary compression on the systolic array is considered in Sections 3.2. Systolic architectures for sliding-window compression are described in Section 3.3. Section 3.4 considers the policies used by systolic architectures for handling adaptive dictionary maintenance.

4.1. Dictionary Compression on the PRAM

The PRAM model is abstracted away from issues of input/output and processor interconnection. This model allows for the assumption that a block of input data is made available to a block of processors in a single time step. Under this assumption, sublinear time parallel data compression is possible. $\mathcal{NC}$ algorithms for static and adaptive dictionary compression are described below.

Under various restrictions, $\mathcal{NC}$ algorithms for static dictionary and sliding-window compression exist. For dictionaries satisfying the prefix property[7], it is

[7] A dictionary satisfies the prefix property if all of the prefixes of each dictionary entry are also in the dictionary.

possible to compute the optimal parsing on-line [HR85]. This sequential algorithm modifies the greedy parsing strategy by considering not only the longest dictionary phrase that matches a prefix of the unencoded input but also all dictionary phrases which match the input beginning at any proper suffix of the previous match. The match extending farthest into the input is chosen. After finding the optimal parsing, the input is encoded by the corresponding dictionary indices.

$\mathcal{NC}$ algorithms for compression with a dictionary having the prefix property parallelize the sequential approach [DS92]. For a static dictionary of size N with the prefix property and with entries at most logarithmic in length, optimal compression can be computed in $O(\log N)$ time using $O(N^2)$ processors or in $O(\log^2 N)$ time and $O(N)$ processors (on the CREW PRAM). In parallel, the length of all matches beginning at positions in the suffix of the previous match are computed and stored in a matrix. Matrix entries are then manipulated to yield longest match information.

By a reduction to the shortest path problem on directed graphs, the optimal parsing with an arbitrary dictionary (and arbitrary entry length) can be found in $O(\log^2 N)$ time with $O(N^3)$ processors or in $O(\log N)$ time using $O(N^4)$ processors. For sliding-window compression, these bounds are reduced to $O(\log N)$ time and $O(N^3)$ processors [DS92].

The discovery of $\mathcal{NC}$ algorithms for some dynamic dictionary compression methods is highly unlikely. LZ78 and two variations are known to be P-complete [D91]. The variants, *first character* and *next character*, implement different parsing strategies and dictionary update procedures [W84, S88A]. The P-completeness of these three methods is established by a reduction from the circuit-value problem [L75, D91].

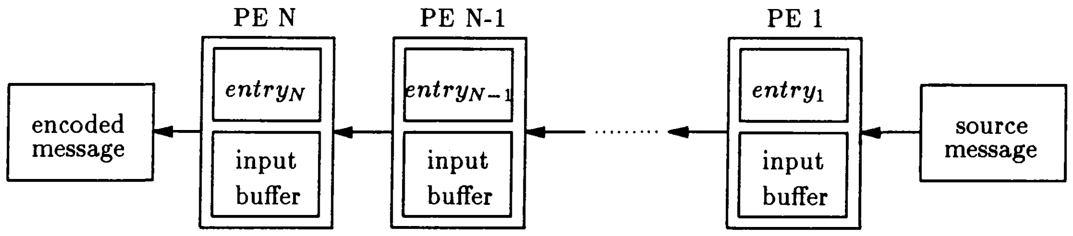

Figure 1

Systolic array for static dictionary compression

4.2. Static Dictionary Compression

Static dictionary compression on the systolic array detects matches between the input and the dictionary entries stored in processing elements (PEs) and encodes an input by the corresponding PE's identification number [GS85, S88A]. For a dictionary of size N, the systolic array consists of N PEs. Input is assumed to enter the array from the right (PE 1) and exit on the left (PE N). PE i stores static dictionary entry, $entry_i$, of length at most l. Relatively small values of l ($l \leq 6$) are required to limit hardware costs. Strings of length exceeding l can be represented by allowing dictionary entries to contain pointers to other dictionary entries. The dictionary is constructed prior to compression and is loaded or hardwired into the processors. Figure 1 depicts the systolic array.

A greedy parsing strategy is used and a search of the entire dictionary for the longest match is avoided by enforcing three conditions: 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 [GS85]. Performance of the greedy approach is unknown when these assumptions fail to hold.

At the start of the encoding clock cycle, PE i receives pair (w, p) from PE $i+1$. w is the input string to be encoded and p is the index of w (0 if not yet encoded). PE i compares $entry_i$ to w and if a match is found, w is encoded as i (p is set to i). At the end of the cycle, PE i sends (w, p) to PE $i-1$.

Decoding is similar to encoding. The compressed string is expanded by replacing each pointer by the corresponding dictionary entry. More precisely, PE i receives pair (w, p) and if $p = i$ then w is set to $entry_i$.

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. No additional characters are read until space is available in the buffer. Unfortunately, locking signals can propagate up the array, eventually locking the entrance processor. A straightforward solution is to restrict the data rate into the decoding circuit to a speed commensurate with that of the systolic array [GS85, S88A].

Static dictionary compression requires no additional overhead for maintaining the dictionary but suffers from the performance limitations of a non-adaptive technique. The key observation about this systolic array architecture is that a new character can enter the array on every clock cycle. Moreover, since each step consists of little more than a parallel comparison, a relatively short clock cycle is sufficient and each PE requires only simple hardware. Assuming an 8-bit character, the communication channel is $8l + \lg N$ bits wide to accommodate (w, p).

Another form of static dictionary compression, *N-gram* compression, has been considered in a parallel setting. *N-gram* compression maintains a dictionary of common phrases each of length exactly N. Parallel hardware implementations for *N-gram* compression which store the dictionary in an associative memory and manipulate the dictionary in parallel on an associative parallel processor have been described [L78].

4.3. Sliding-Window Compression

This section considers systolic array architectures for sliding-window (LZ77 or LZ1) dictionary compression. Recall, sliding-window compression replaces repeated phrases by pointers to positions within a fixed-sized window of the input immediately preceding the current compression position. The window implicitly represents the dictionary. Pointers consist of (*position, length*) pairs denoting the longest match between the incoming input and the previous fixed-sized window of characters. The three systolic architectures described in this section differ in the manner pointer pairs are computed.

4.3.1. The Match Tree Architecture

For a fixed-sized window of N characters, the first systolic architecture, called the *Match Tree* architecture, employs $4N - 1$ processors. The first $2N$ processors are configured as a systolic array and the remaining $2N-1$ processors are arranged as a binary tree (the *match tree*) attached to the systolic array [GS85, S88A, S92]. Input enters at the right end of the systolic array (PE $2N$), travels along the systolic array, and exits on the left (PE 1). The previous $2N$ characters processed are stored in the systolic array, one character per processing element. The character stored in PE i is referred to as $entry_i$. The match tree is used to determine match position and length information. Figure 2 depicts the Match Tree architecture for $N = 8$.

Compression is carried out in phases of N steps. At the start of each phase, PE i overwrites $entry_i$ with the character in its input buffer. The next N shifts and comparisons of the input have the effect of passing each of the N characters (which began the phase in PE's $N + 1$ through $2N$) past the N characters that preceded that character. During step k $(1 \leq k \leq N)$, PE i $(N-k+1 \leq i \leq 2N-k)$ compares its input character to $entry_i$. Notice that not all processors participate in each step. For example, PE $N-1$ makes a comparison in the first step, but only reads and sends input for the remainder of the phase. An additional special purpose

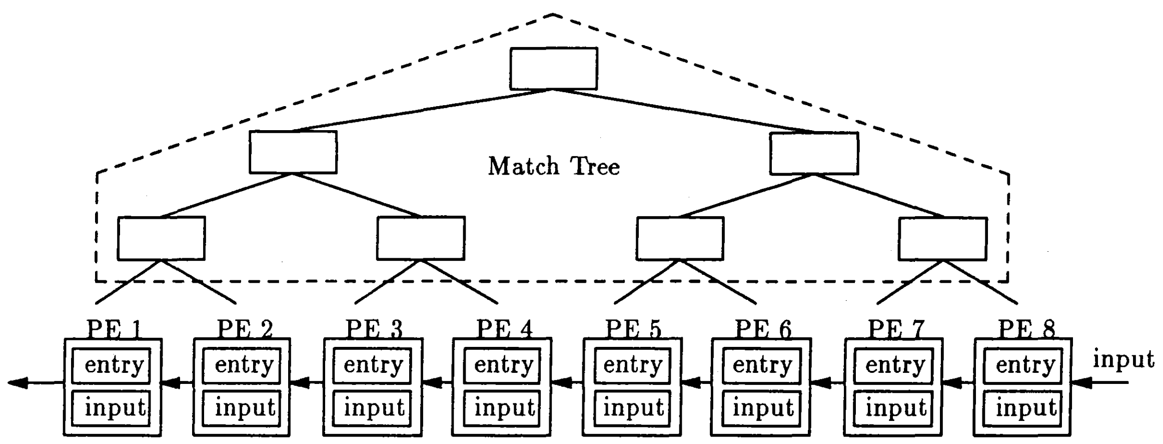

Figure 2

Match Tree Architecture

processor PE $2N$ handles 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 array are ignored. Moreover, whenever pointers overlap, the output pointer is altered to maximize compression.

As a character travels through the systolic array, it is accompanied by its longest match location and length information, pointer pair (*position, length*), which is updated whenever a longer match is found. Calculating the longest match information may require communication among non-neighboring processors. In logarithmic time, the match tree propagates information up and down the tree to determine the position and length of the longest match. If PE i detects a match, it checks with each of its neighboring processors. If PE $i-1$ did not match, then PE i sends a message to its parent processor in the match tree signaling that it has the first character of a matching string. Similarly, if PE $i+1$ did not match, PE i flags its parent processor that it is at the end of a match. These signals propagate up the tree until some processor, PE k, is able to pair up a start and an end symbol. PE k then calculates the match length and returns the information to the processor (PE j) holding the first character in the match. If the new match length exceeds the existing length of the longest match beginning at that character, the match position is assigned the processor number j and the length register is updated.

A different 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 $O(\sqrt{N})$ [GS85]. Processors are placed in an $O(\sqrt{N}) \times O(\sqrt{N})$ grid with constant length connections and additional steps 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 expands all pointers by employing a systolic array of $O(N)$ processors. Since all pointers are to locations fewer than N characters away, the N most recently decoded characters are stored in the array. The pointer (p, l) is decoded by concatenating the characters stored in processors PE p through PE $p - l + 1$. The input is augmented with two additional pointers which aid in switching from different modes in the decoding process. As in encoding, decoding proceeds in blocks of N characters. Before entering the array, the pointer (p, l) is expanded into the sequence of integers $p, p+1, \ldots, p+l-1$. The expanded encoded message enters the array on every other system step. 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.

Like the static dictionary model, the Match Tree 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 its 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. Consequently, $O(\log N)$ time is required to process an input character. If m is the maximum match length then this can be reduced to $O(\log m)$ using a collection of trees. The decoder produces an output on every system cycle. Both the encoder and decoder require the input to flow through the entire array resulting in a linear through delay[8]. The processing elements are simple and can be implemented in VLSI straightforwardly.

4.3.2. Broadcast/Reduce Architecture

A second design for sliding-window compression, the ***Broadcast/Reduce*** architecture, is also built from a systolic array and binary trees of processors [Z90A, S92]. In this architecture, the data stream and dictionary are separated and longest match decisions are made by tree processors. This is unlike the Match tree design in which the data, dictionary, and output all flow through the systolic array. The longest match pointer information for each character is computed by making each character simultaneously available to every processor via a binary tree of processors, and identifying the largest match at each cycle using another tree-connected collection of processors.

For a dictionary of size N, the architecture consists of $3N - 2$ processors. N processors are arranged in a systolic array and the remaining processors are configured as two binary trees (each containing $N - 1$ processors) synchronized with the systolic array (see Figure 3). One of the binary trees (the ***broadcast tree***) is placed on top of the systolic array. On each cycle, input enters at the root of the broadcast tree and data in the tree is propagated down one level. After a delay of $\log N$ steps, the same input character enters the systolic array at PE N. Consequently, the input enters the systolic array one step after the copy sent via the broadcast tree reaches processors in the array. The input character leaves the broadcast tree and is simultaneously compared to the N preceding characters, each

[8] Through delay for an input character is the interval between its entering and exiting the systolic array.

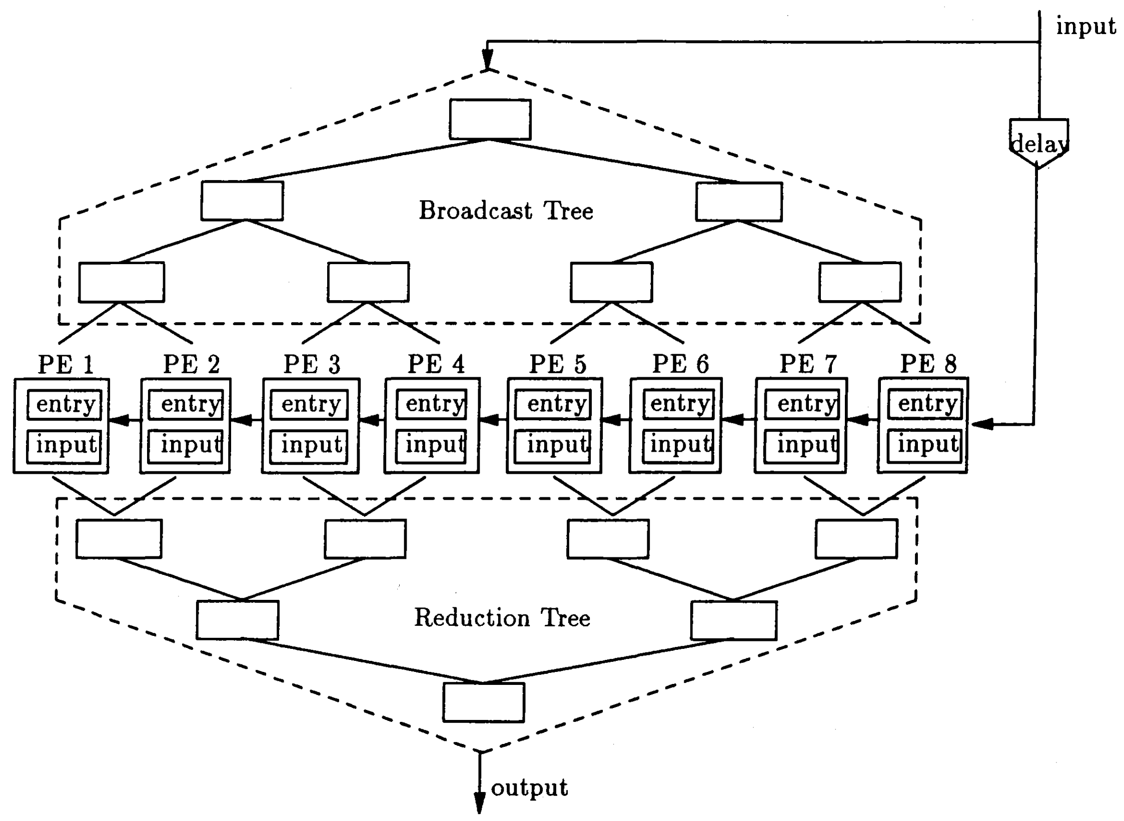

Figure 3

Broadcast/reduce architecture

available in one of the N PEs of the array. The other binary tree (the *reduction tree*) is placed below the systolic array and combines match information. Ultimately, the (*position, length*) pair representing the longest match *ending* (previous approaches calculate the longest match *beginning* at a particular symbol) at that character is output by the root of the reduction tree. As in the Match Tree design, a special purpose processor at the root of the reduction tree combines pointer pairs.

Since global communication among processors is not required, the match length computations of the Match Tree design are avoided. All steps take unit time and system speed is unaffected by match length. Also, the linear through delay of previous architectures is reduced to $2\log N$ at the expense of additional hardware. The decoding architecture is described in [Z90A]. The designs of the next section eliminate the trees of processors.

4.3.3. Wrap Architecture

More recent architectures for sliding-window compression on the systolic array attempt to remove the trees introduced in previous designs. The *Wrap* architecture consists of a bi-directional systolic array [Z90B, S92]. Input is encoded by maximal (*position, length*) pointers as it travels through the array. After being encoded, characters wrap back and flow through the array as dictionary entries. The movement of data alternates with the shifting of the dictionary (sliding-window) and pointer information is updated without the need for a tree of processors.

For a dictionary of size N, the Wrap architecture consists of a systolic array of $N/2$ processing elements (PEs) connected by a two-way communication channel. PE i has an input register and stores a character of the sliding-window in $entry_i$. Input enters on the right (PE $N/2$) and travels through the array and exits on the left (PE 1). After remaining latched for one system step, the character is output and a copy re-enters the array as $entry_1$ at PE 1. This copy moves right through the entry registers. The Wrap architecture is depicted in Figure 4. Input enters on even numbered cycles and characters in the entry registers move right on odd numbered cycles. As an input character shifts left from PE $N/2$ to PE 1, it is compared to the N characters that preceded it. At PE i, if the input character matches $entry_i$ then the (*position, length*) pair is updated. When the input character leaves PE 1 and is latched, it is accompanied by a (*position, length*) pair representing the longest match that *ends* with this character. As with previous designs, the output is then manipulated by a special purpose processor to yield the final encoding.

The Wrap encoding architecture allows input on every other cycle. This restriction can be eliminated by having PE i compare $entry_i$ to both its input

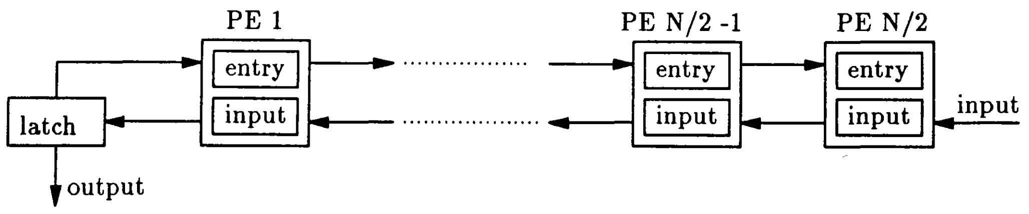

Figure 4

Wrap Architecture

register and the input register of PE $i-1$. However, processors and data paths of this design are more complicated and increase the system cycle and hardware costs.

An architecture similar to the Wrap design sends two copies of each input into the systolic array [HR90, RH91, RH92]. The second copy enters the array N steps after the first. Processing elements require relatively complicated hardware to determine in which steps to participate.

4.4. Adaptive Dictionary Compression

Adaptive dictionary compression utilizes an evolving dictionary that adapts to changes in the input characteristics. Usually, adaptive approaches achieve superior compression results over static and semi-adaptive methods. There are a number of different adaptive approaches, all of which must include strategies for match selection and dictionary updating. Adaptive dictionary compression on the systolic array is the focus of this section.

An architecture for adaptive dictionary compression, referred to here as the *Pair* architecture, is based on a pair forest representation of the dictionary [S88B]. In a pair forest representation, each character of the input alphabet is present in the dictionary and larger entries consist of a pair of pointers to other entries. The Pair architecture stores the dictionary in a systolic array, one entry in each processing element. The dictionary is updated by adding entries derived from the concatenation of the previous match with the current match. Thus, each processing element stores an entry consisting of two pointers. Compression is achieved by replacing input which matches the pair of pointers stored in a processing element by the single processor number. When the dictionary becomes full, the Pair architecture switches control to a second dictionary. To accommodate this scheme, two separate dictionary arrays 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.

For a dictionary of size N, encoding is performed on a systolic array consisting of N processing elements (PEs), numbered 1 through N from left to right. A second identical systolic array is used for dictionary swapping. PEs at the start of the array store the characters of the input alphabet and later PEs store a pair of pointers. That is, for input alphabet Σ, PEs 1 through $|\Sigma|$ store one input character each. Each processor stores a flag bit, $flag_i$, used to delimit the current dictionary. Initially, only $flag_{|\Sigma|+1}$ is set. If $flag_i$ is set then PE i is designated to "learn" the next new dictionary entry. All PEs to the left of the learning processor contain dictionary entries, while PEs to the right are empty. Input enters from the left (PE 1) and leaves on the right (PE N). A greedy parsing strategy is used and whenever a prefix of the input 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. If $flag_i$ is set then PE i must allow one pointer to pass through before adopting its pair entry in order to avoid duplicating the entry in PE $i-1$. After PE N learns its entry, a 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.

Decoding utilizes a systolic array of size N with bi-directional communication paths. PEs are numbered N to 1, with PE 1 being the rightmost processor (this is the reverse of the encoding array). Data enters from the left (PE N) and exits on the right (PE 1). Processors to the right of PE $|\Sigma|$ are initialized to contain the source alphabet and PE $|\Sigma|+1$ starts out as the learning processor. Each processor has a flag bit and a dictionary entry. Upon adopting an entry, the learning processor passes the flag upstream to the processor on its left. As described, the decoding dictionary will fail to adopt entries that appear in the encoding dictionary. To see this, consider the four pointers p_1, p_2, p_3, and p_4 (p_1 then p_2 then p_3 then p_4) traveling down the encoding array. Pointers p_1 and p_2 reach the learning processor PE j and are adopted. Later, PE $i+1$ adopts pointers p_2 and p_3. And finally, PE $i+2$ adopts p_3 and p_4. In the decoder, learning processor PE k adopts p_1 and p_2 and PE $k+1$ later adopts p_3 and p_4. Pointer pair (p_2, p_3) is not entered into the decoding dictionary. To remedy this, nil pointers are interspersed between every input to the decoder. This ensures that the encoder and decoder learn the same dictionary. Decoding mirrors encoding; that is, whenever an input substring arrives at a processor with index equal to the input, it is replaced by the entry stored in the processor. As in previous systolic architectures, additional measures are needed to avoid buffer overflows which may occur when compressed inputs are expanded. The Pair architecture uses a stop bit to signal the processor to the left to wait to send additional data.

As described, the Pair design stores each character of the input alphabet in the PEs at the start of the array (or at the end in the case of decoding). This requirement is not necessary since the input characters can be handled as pointers concatenation of the previous match with the current match. Thus, each processing element stores an entry consisting of two pointers. Compression is achieved by replacing input which matches the pair of pointers stored in a processing element by the single processor number. When the dictionary becomes full, the Pair architecture switches control to a second dictionary. To accommodate this scheme, two separate dictionary arrays 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.

For a dictionary of size N, encoding is performed on a systolic array consisting of N processing elements (PEs), numbered 1 through N from left to right. A second identical systolic array is used for dictionary swapping. PEs at the start of the array store the characters of the input alphabet and later PEs store a pair of pointers. That is, for input alphabet Σ, PEs 1 through $|\Sigma|$ store one input character each. Each processor stores a flag bit, $flag_i$, used to delimit the current dictionary. Initially, only $flag_{|\Sigma|+1}$ is set. If $flag_i$ is set then PE i is designated to "learn" the next new dictionary entry. All PEs to the left of the learning processor contain dictionary entries, while PEs to the right are empty. Input enters from the left (PE 1) and leaves on the right (PE N). A greedy parsing strategy is used and whenever a prefix of the input 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. If $flag_i$ is set then PE i must allow one pointer to pass through before adopting its pair entry in order to avoid duplicating the entry in PE $i-1$. After PE N learns its entry, a 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.

Decoding utilizes a systolic array of size N with bi-directional communication paths. PEs are numbered N to 1, with PE 1 being the rightmost processor (this is the reverse of the encoding array). Data enters from the left (PE N) and exits on the right (PE 1). Processors to the right of PE $|\Sigma|$ are initialized to contain the source alphabet and PE $|\Sigma|+1$ starts out as the learning processor. Each processor has a flag bit and a dictionary entry. Upon adopting an entry, the learning processor passes the flag upstream to the processor on its left. As described, the decoding dictionary will fail to adopt entries that appear in the encoding dictionary. To see this, consider the four pointers p_1, p_2, p_3, and p_4 (p_1 then p_2 then p_3 then p_4) traveling down the encoding array. Pointers p_1 and p_2 reach the learning processor PE j and are adopted. Later, PE $i+1$ adopts pointers p_2 and p_3. And finally, PE $i+2$ adopts p_3 and p_4. In the decoder, learning processor PE k adopts p_1 and p_2 and PE $k+1$ later adopts p_3 and p_4. Pointer pair (p_2, p_3) is not entered into the decoding dictionary. To remedy this, nil pointers are interspersed between every input to the decoder. This ensures that the encoder and decoder learn the same dictionary. Decoding mirrors encoding; that is, whenever an input substring arrives at a processor with index equal to the input, it is replaced by the entry stored in the processor. As in previous systolic architectures, additional measures are needed to avoid buffer overflows which may occur when compressed inputs are expanded. The Pair architecture uses a stop bit to signal the processor to the left to wait to send additional data.

As described, the Pair design stores each character of the input alphabet in the PEs at the start of the array (or at the end in the case of decoding). This requirement is not necessary since the input characters can be handled as pointers into the dictionary (by padding them to length $\log N$) without having to store them explicitly in the array.

Storer and Reif present a systolic real-time architecture based on a modified version of the update heuristic used in the Pair architecture [SR90]. The dictionary update heuristic, instead of entering the concatenation of two previous matches (after allowing one pointer to pass through), adds the concatenation of two pointers only if neither pointer was adopted by the preceding processor. A prototype VLSI chip for this design was built using a systolic array of $3,839$ PEs ($4,096$ minus the 257 which correspond to characters of the input and a special nil value). The implementation runs on a 40 Mhz clock and yields a data rate of 40 Mbytes per second. Section 3.6 describes some additional details of the hardware.

A variant of the previous architectures stores several dictionary entries in each processing element [SRM90]. In each clock cycle, each PE performs a parallel match between its entries and the contents of its input buffer. This yields a more compact design requiring significantly less hardware that operates at commensurate speeds. A similar variation uses Content Addressable Memories in each PE to perform parallel matches between the input and several dictionary entries withing a single clock cycle [MRS92].

A different use of parallelism for adaptive dictionary compression employs a tagged trie data structure for dictionary management [BB92]. The tags are used to determine what strings are to be deleted. Two parallel processes carry out compression and dictionary updating. With little demand on current technology, hardware implementations achieve data rates ranging from 13.6 to 20 Mbytes per second.

4.5. Other Dictionary Methods

This section considers a class of parallel compression algorithms which implement a sequential dictionary of words using a self-organizing list. Self-organizing lists permute the order of list entries after an entry is accessed, attempting to place more frequently requested entries closer to the front of the list. To distinguish this collection of compression techniques from other dynamic dictionary schemes, we refer to them collectively as *list compression* methods under a particular update heuristic. Parallel list compression schemes operate at much higher data rates of 40Mbytes per second (assuming a 40 MHz clock and 1 byte per cycle) in comparison to the data rates of sequential compression systems that range from 10 to 320 Kbytes per second.

A list compression method uses a self-organizing data structure to maintain a list of source messages and an encoding of the integers to compress list indices [BSTW86, E87, R87, HC87]. To compress a word, it is located in the dictionary and encoded by its list position. After a word has been referenced, the list is reorganized appropriately. After an input string has been replaced by its matching list index, it can be further compressed by encoding list positions. A variable-length encoding of the integers, such as Elias codes, Fibonacci codes or start-step-stop codes, or a non-codeword based method such as arithmetic coding can be used to compress list positions [BCW90].

Move-to-front and transpose are two update heuristics used in parallel list compression (see [HH85] for a survey of self-organizing linear search). The move-to-front heuristic moves the accessed string to the front of the list, shifting all records previously ahead of it back one position. The transpose heuristic permutes the list by exchanging the accessed entry with the one immediately before it in the list. After reaching a steady state, where many further search requests are not expected to significantly impact the expected search time, the expected access cost is less for transpose than for move-to-front, but the convergence time or number of accesses required to reach a steady state is greater for transpose than for move-to-front [HH85]. There are applications for which each of move-to-front and transpose outperforms the other. For any particular application, simulations are necessary to determine the superior heuristic.

VLSI implementation issues regarding input/output pin requirements have forced the investigation of two major algorithmic variants for list compression. The simpler procedure permutes a fixed-length list of symbols and the other approach permits arbitrary-length list entries. A byte-level fixed-length list 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 of 30% to 40% on text files [TW89]. Methods which allow arbitrary-length dictionary entries provide higher compression of 48% to 75%. For parallel models, however, arbitrary-length list entries that must travel between processing elements force an arbitrary number of input/output pins on each PE. Designs for the simpler fixed-length list entries are described below. Approximations for the more general list of variable-length entries are addressed in Section 3.5.4.

4.5.1. Move-to-Front List Compression on the Systolic Array

Assuming a dictionary of size N, the first systolic array architecture for list compression under the move-to-front update heuristic and based on the fixed-length scheme employs two systolic arrays of N PEs each [TW89]. One array is used for encoding and the other for decoding. Input enters the systolic array from the right (PE 1) and leaves to the left (PE N). As input flows through the array, matches are detected between the input and the characters stored in the PEs.

To encode fixed-length word w using the move-to-front heuristic, w is compared to the list entries of successive processors. PE i has an input buffer and stores dictionary entry $entry_i$. At PE 1 (the front of the list) w overwrites the current list entry which is transmitted to PE 2. Next, the list entry of PE 2 is overwritten by the previous entry of PE 1. List entries continue to cascade down the list until w matches the list entry and the final entry is updated. Upon matching, w is encoded by the identification number of the matching PE. By depositing the input character in the first processor as it enters the array and then cascading previous processing element contents down the array, the move-to-front behavior is realized. The output of the array, consisting of a sequence of 8-bit list indices, is fed into a fixed-to-variable length coding processor.

At the beginning of the clock cycle, PE i receives 4-tuple (w, e, p, m) from PE $i-1$. w is the word to be encoded, e is the word being moved down in the list (i.e., e is the current contents of $entry_i$ and is nil if w is already encoded), p is the list position of w (0 if no match found yet) and m is a flag bit which is set if the list needs further updating. If m is set then PE i compares $entry_i$ to w. If the words match, PE i overwrites $entry_i$ with e, sets m to 0 and transmits (w, e, i, m) to PE $i+1$. If w differs from $entry_i$ then PE i sends (w, $entry_i$, p, m) to PE $i+1$ and copies e into $entry_i$. Otherwise the input passes through PE i unchanged. If words are limited to a single byte, this scheme achieves compression savings of 19% to 38% at a data rate of 40 Mbytes per second (assuming a 40 MHz clock) [TW89].

A string of $\lg N$-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. Unfortunately, no high-data rate fixed-to-variable-length coder is known. This bottleneck dictates the data rate of the entire systolic system. Using a table lookup approach, Thomborson and Wei experimented with various tail-end encoders [TW89]. Their empirical findings suggest that their fixed-to-variable code converter gives rather poor compression (11% to 22%) but can perform at a data rate commensurate with the systolic array. Huffman coding provides better compression (19% to 38%) but operates at a limited data rate. Dynamic Huffman coding and arithmetic coding yield far better compression but even the most sophisticated implementations operate at data rates below 15 KBytes per second [MP88, K82, V87].

The systolic decoder for move-to-front list compression cannot simply mirror the encoder since it is impossible to update the dictionary until after the input has passed PE 1 (except if the input is '1') and has been decoded at some processor later in the array. By using a two-way communication channel and having input enter the array on the left (PE N) and exit on the right (PE 1), it may be possible to mirror the encoder.

Alternatively, instead of storing the i^{th} word in the dictionary, PE i reserves pos_i which is the list position of the alphabet symbol with index representation i. For ASCII codes, pos_i is the table index of the entry with ASCII value i. At the onset of the decoding clock cycle, PE i receives input (w, p) from PE $i-1$. As in the encoder, p is the encoded list position and w is the decoded word occurring in position p of the list. If $p = pos_i$ then pos_i is set to '1' and w is assigned i. That is, the symbol with representation i is moved to the first list entry and w is decoded as i. If $p > pos_i$ then pos_i is incremented to reflect the movement of character representation i deeper into the list.

4.5.2. Transpose List Compression on the Systolic Array

Parallel transpose list compression is described by the following general paradigm. Encoder and decoder maintain identical word lists using the transpose heuristic. Namely, after a word is used it is exchanged with the word stored in the position immediately preceding its original position. This section describes two designs for parallel list compression under the transpose heuristic. The first approach is similar to the move-to-front designs and uses a systolic array. However, input is restricted to every other system step (the reasons for this are discussed later). The second design combines a systolic array with trees to improve the linear through delay of the first design.

For the first design, to transmit word w on the systolic array, w is compared to the list entries of successive processors. If w matches the list entry in PE i, it is encoded as i. The encoder then updates the list by transposing the list entry (w) of PE i with the list entry in PE $i-1$. When the decoder array receives list index i, it decodes it as the list word stored in PE i (w) and then updates the list by exchanging w with the previous list entry stored in PE $i-1$. Since several matches can be detected in parallel the list update procedure needs additional consideration.

In the sequential setting, a sequence of words that match the list structure in successive entries are handled in the same way as other matches. However, in the systolic environment, matches corresponding to successive entries in the array impose additional constraints when the list of words is being manipulated in parallel. That is, simultaneous matches occurring in different locations in the array may force global communication among the processors to determine the contents of the updated list. To illustrate this difficulty, consider the input string "abcdefgh" and the word list " h, g, f, e, d, c, b, a". Sequential transpose list compression outputs the sequence of positions 8, 8, 6, 6, 4, 4, 2, 2 and the final word list is identical to the original. In a sense, each pair of matches causes updates that cancel each other. On the systolic array, all eight matches are detected simultaneously. Handling the subsequent update may require global communication.

For a list of length N, the systolic array encoder consists of N processing elements linearly connected by a two-way communication channel [SH92A]. PE i stores the list entry which is currently in position i in the list and a copy of the input word PE i considered on the previous clock cycle. The list entry will be referred to as $entry_i$ and the prior input word as $oldw_i$. The input stream enters the array from the right (PE 1) and the encoded message exits at the left (PE N).

In order to prevent a contiguous sequence of matches from occurring concurrently, input packets enter the array only on every other clock cycle. The word list is updated at the start of each encoding cycle. Later in the cycle, word matches are detected and encoded.

At the beginning of the clock cycle, PE i receives triple (w, e, p) from PE $i-1$ and bit m_{i+1} from PE $i+1$. As in the move-to-front systolic architecture, w is a word to be matched, e is the current contents of $entry_{i-1}$ that may be needed for a transpose update, p is the list position of w (0 is no match found yet) and m_{i+1} is a bit flag which is set if PE $i+1$ detected a match in the previous clock cycle. If m_{i+1} is set then PE i overwrites $entry_i$ with $oldw_i$. If w matches $entry_i$ then PE i carries out three tasks. Namely, PE i sets $p = i$, assigns $m_i = 1$, and (if $i > 1$) overwrites $entry_i$ with input e (equivalently $entry_{i-1}$ obtained from PE $i-1$).

At the close of the clock cycle, PE i overwrites $oldw_i$ with w and transmits (w, $entry_i$, p) to PE $i+1$ and (m_i) to PE $i-1$. Contention is avoided as a result of restricting input to every other cycle.

A systolic transpose decoder which mirrors the encoder also allows input to enter the array on every other clock cycle. At the outset of the cycle, the word list is updated. Unlike the encoder, where only a single bit is passed from PE i back to PE $i-1$ to facilitate updating, the decoder requires $entry_i$ be transmitted along with the single match bit. Following the list updating, list indices are replaced by word list entries. An alternative systolic decoder similar to the move-to-front decoder of the previous section, processes packets on every step. See [SH92A] for details.

For a fixed-length list, such as the 256 different 8-bit ASCII characters, each processor is initialized to contain one of the 8-bit bytes. Alternatively, new words can be added to the list until the list becomes full. Moreover, if an input word w is not in the current list of size K ($1 \leq K \leq N$) the word is encoded by the index $K+1$ followed by the word w and the list is updated by transposing w with the list entry K. If $K = N$ (i.e., the list is full), word w replaces the last list entry. The decoder "learns" the word list in a similar fashion. In the systolic array, an additional flag bit in each processor is used to delimit the current list. The processor holding the flag is designated as the first empty list entry. Initially, the flag bit in PE 1 is set.

The linear through delay of the previous systolic array transpose designs is determined by the passage of the input from PE 1 through to PE N. An architecture, similar to the Broadcast/Reduce design in Section 3.3.2, combines a systolic array with trees to reduce the through delay to logarithmic at the expense of additional hardware. The trees broadcast the input to the systolic array and reduce the simultaneous outputs of the processors. In addition to decreased through delay, the restriction allowing data to enter the array only on every other system step is eliminated.

For a list of size N, the architecture consists of $3N-2$ processors. N processors are arranged in a systolic array and the remaining processors are configured as two binary trees (the broadcast and reduction trees). Input enters at the root of the broadcast tree and is propagated down the tree to each array processor. Results of the processors are reduced to a single non-null output via the propagation toward the root of the reduction tree. The tree interconnect provides total through delay of $2 \log N$.

Processor PE i in the systolic array stores the list entry which is currently in position i in the list. This entry is referred to as $entry_i$. PE i receives input from the broadcast tree and from PE $i-1$ and PE $i+1$. PE i transmits $entry_i$ to PE $i-1$ and PE $i+1$ and outputs match information to the reduction tree. Processors in the broadcast tree simply pass their input to their outputs. Reduction processors receive two inputs at least one of which is zero. When both inputs are zero the reduction tree outputs zero. Otherwise, the reduction processor transmits the non-zero input.

After the input has propagated down the broadcast tree to the array processors, encoding proceeds as follows. At the beginning of the clock cycle, PE i receives (w) from the broadcast tree, $(entry_{i-1})$ from PE $i-1$, and $(entry_{i+1})$ from PE $i+1$. w is the word to be encoded. If w matches $entry_i$ then w is set to i and $entry_{i-1}$ is written into $entry_i$. If w matches $entry_{i+1}$ (received from PE $i+1$ at the start of the cycle) then PE i overwrites $entry_i$ with $entry_{i+1}$ and sets w to 0. Otherwise, PE i sets w to 0. At the close of the clock cycle, PE i transmits $entry_i$ to PE $i+1$ and PE $i-1$ and sends w to its neighboring processor in the reduction tree. Thus, at the end of each clock cycle, exactly one processor (the one which matched the input symbol) outputs a non-zero value into the base of the broadcast tree. This non-zero value is propagated to the root of the reduction tree and finally output. Decoding mirrors encoding [SH92A].

4.5.3. List Compression on the Xnet

The Xnet interconnection network provides a modified mesh-connected structure suitable for rapid communication among neighboring processing elements. Switches, located between each pair of PEs, permit vertical, horizontal, or diagonal connections. The connections wrap around, meaning that switches are located between the top and bottom row of the mesh and between the extreme left and right columns. During each step, the switches are identically set throughout the system and computation proceeds synchronously. Each PE is connected to two of its nearest neighbors and communication proceeds in a single direction. For example, when the switches are set to provide North-to-South connections, a single PE receives input from the processor above it in the mesh and transmits data to the processor below it. Each PE has its own local memory and is assigned a unique identification number. High-speed Xnet architectures for list compression under the move-to-front and transpose permutation heuristics are described below.

List compression on the Xnet is similar to the systolic array designs of the previous sections. Additional phases are introduced to accommodate the switching connections [SH92B].

For a list of length N^2 the Xnet implementations for maintaining a self-organizing list of fixed-length entries under move-to-front consists of N^2 PEs. PE i stores the list entry which is currently in position i in the list (referred to as $entry_i$). The input enters the Xnet at PE 1 and exits at PE N^2.

Each step of consists of updating the list and checking for matches between the input and the stored entries. Each step is carried out in 2 phases. During Phase 1 the Xnet switches are set to connect NorthWest-to-SouthEast (NW-SE) PEs. PE kN, where $k \geq 1$, sends 4-tuple (w, e, p, f) to PE $kN + 1$. w is the input being accessed in the list, e is the current contents of $entry_i$, p is the list position of w (p is used to perform data compression) and f is a bit flag which is set if $entry_i$ is to be cascaded to PE $i + 1$. In Phase 2, the Xnet connections are changed to link processors East-to-West (E-W). PE i, where i is not a multiple of N, transmits 4-tuple (w, e, p, f) to PE $i + 1$. In either phase, processors receiving input update their entry and then compare it to the input character. That is, if $f = 1$, PE i exchanges $entry_i$ and e. Then, if w matches the new entry stored in e (equivalently the previous contents of $entry_i$) then PE i sets $f = 0$ and $p = i$. To realize the move-to-front update, if the input w is not already at the front of the list, PE 1 writes $entry_1$ into e, stores w in $entry_1$ and set $f = 1$.

For a list of length N^2, the Xnet design for list compression under the transpose update heuristic consists of N^2 PEs. PE i stores the i^{th} list record and a copy of the input word PE i considered on the previous step. The list entry is referred to as $entry_i$ and the prior input word as $oldw_i$. Since several matches can be detected simultaneously, the Xnet design for transpose allows input to enter the mesh on every other step. The dictionary is updated and, later in the cycle, word matches are detected.

In Phase 1, the Xnet switches are set NW-SE. PE kN, where $k \geq 1$, sends 3-tuple (w, e, p) to PE $kN+1$. As in the Xnet move-to-front design, w is the input being accessed in the list, e is $entry_i$ that may be needed for transpose update, and p is the list index of w used for performing data compression. Switches are changed to reverse the diagonal connections of Phase 1 to SE-NW links in Phase 2. PE $kN+1$, $k \geq 1$ sends bit m_{kN+1} to PE kN. m_{kN+1} is a bit flag which is set if PE $kN+1$ detected a match in the previous step. At the start of Phase 3, the Xnet alters the connections to W-E links. PE i, where i is not a multiple of N, sends 3-tuple (w, e, p) to PE $i+1$. In Phase 4, the switches link E-W and PE $i+1$, where i is not a multiple of N, transmits m_{i+1} back to PE i. If m_{i+1} is set then PE i (for all i) overwrites $entry_i$ with $oldw_i$. If w matches $entry_i$ then PE i sets $m_i = 1$ and (if $i > 1$) overwrites $entry_i$ with input e. Finally, PE i overwrites $oldw_i$ with w. Contention is avoided as a result of restricting input to every other cycle.

4.5.4. Parallel List Compression for Arbitrary-length Entries

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 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 a 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.

As pointed out earlier, a parallel list compression architecture may have difficulty allowing non-fixed length words to travel between processing elements because of the potentially unreasonable pin requirements. For the arbitrary-length scheme, placing a limit on the length of words and hashing approximate the general paradigm [TW89].

One simple solution to the problem of unbounded pin requirements is to place a bound on the maximum allowable word length and use the fixed-length systems of the previous sections [TW89]. This bound enforces a limit on the pin requirements at the risk of deteriorating compression and speed. The appropriate maximum word length is dependent on the application.

In order to avoid the potential VLSI issues, an approximation using a hard-wired hash table to map arbitrary words onto an 8-bit byte has been considered [TW89]. These 8-bit codes are entered into a 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-data rate 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 ranging from 25% to 65% and an input data rate of 120 Mbytes per second running on a 40 MHz word stream clock. This is considerably lower than the compression savings of 30% to 75% obtained by BSTW.

4.6. Evaluation

Preliminary designs for encoding and decoding chips for byte-level move-to-front list compression on the systolic array have critical paths of 25 nanoseconds (assuming 1986 CMOS standard logic) and can operate at a data rate of 40Mbytes per second [TW89]. The data rate of the design is dependent on the behavior of the front-end variable-to-fixed decoder. This is an advantage over the Match tree design whose data transfer rate is determined by the maximum match length [GS85]. For these systems, any improvement in the data rate may require changes to many system components. The hashing scheme, described above, for list compression of arbitrary-length entries yields a data rate of 120 Mbytes per second assuming a 40 MHz word stream clock. One drawback of the list compression methods is that a variable-length code is output which requires a 360 MHz queue at the end of the systolic array to handle bit-serial outputs. This requires more advanced implementation technology than is currently available in CMOS [BB92].

Of the Match Tree, Broadcast/Reduce, and the Wrap architectures for sliding-window compression, the simpler Wrap system which processes input on every step is superior. However, if through delay is critical, the logarithmic through delay of the Broadcast/Reduce architecture is favored. In practice, binary trees of processors are often readily available in hardware. In these cases, the architectures require additional evaluation [S92]. However, the Match Tree architecture remains less desirable since its data rate is dependent of the size of the dictionary.

Using a systolic array of 4,096 processing elements and assuming 40 MHz systolic hardware, the Match Tree architecture achieves a data rate of 40 Mbytes per second [GS85, S88A]. Preliminary estimates for the Wrap and Broadcast/Reduce architectures specify similar data rates [Z90A, Z90B]. A prototype VLSI chip for sliding-window compression (similar to Wrap approach) implementing nine processing elements was fabricated using CMOS 2-micron technology [S92]. Assuming a 40 MHz clock, the chip yields a data rate of 20 Mbytes per second. A systolic array for adaptive dictionary compression using approximately 4,000 processing elements on 30 custom chips has been fabricated using 1.2-micron CMOS and can operate at 40Mbytes per second [SR91]. A custom VLSI chip for the adaptive dictionary compression architecture of the last section built using 1.0 micron CMOS technology operates with a 20 MHz clock, consumes 8 bits per cycle, and achieves a data rate of 20 Mbytes per second [SR91, S92]. Each chip houses 128 processors. A complete systolic array of 3,839 processors is obtained by chaining 30 chips together. Based on simulations, a clock rate of 75 MHz is feasible, yielding a data rate of 75 Mbytes per second. A new design, incorporating techniques for isolating pad delays, is estimated to run at 40 Mbytes per second and has 256 processing elements per chip [S92].

5. MULTIPLE COMPRESSION

The parallel systems of Sections 2 and 3 utilize a single compression method that manipulates the data in parallel to improve compression speed. An alternative application of parallelism combines multiple compression techniques operating simultaneously to obtain greater compression. Current approaches to this form of parallel compression are pipelining and competitive processing.

Pipelined compression systems combine two or more compression methods in succession to compress the input more effectively than the individual methods operating in isolation. Communication applications are an example of systems for which the additional time used to carry out the sequence of methods is insignificant in view of the relatively slow communication speed [PMK91]. Research has focused on the selection of appropriate methods and their optimal positioning in the pipeline.

Statistical methods take advantage of character redundancy whereas dictionary methods profit from string repetition. This distinction is the basis of current pipelined compression systems [BM90, PM90, PMK91]. Dictionary methods maintain a table of strings and compression is carried out by replacing repeated strings by references (or indices) into the table. The fixed-length indices that are produced by the dictionary compressor may contain repeated copies of the same index. This corresponds to strings in the table appearing more than once in the input. By treating the dictionary indices as the input alphabet to a statistical compressor, these systems, any improvement in the data rate may require changes to many system components. The hashing scheme, described above, for list compression of arbitrary-length entries yields a data rate of 120 Mbytes per second assuming a 40 MHz word stream clock. One drawback of the list compression methods is that a variable-length code is output which requires a 360 MHz queue at the end of the systolic array to handle bit-serial outputs. This requires more advanced implementation technology than is currently available in CMOS [BB92].

Of the Match Tree, Broadcast/Reduce, and the Wrap architectures for sliding-window compression, the simpler Wrap system which processes input on every step is superior. However, if through delay is critical, the logarithmic through delay of the Broadcast/Reduce architecture is favored. In practice, binary trees of processors are often readily available in hardware. In these cases, the architectures require additional evaluation [S92]. However, the Match Tree architecture remains less desirable since its data rate is dependent of the size of the dictionary.

Using a systolic array of 4,096 processing elements and assuming 40 MHz systolic hardware, the Match Tree architecture achieves a data rate of 40 Mbytes per second [GS85, S88A]. Preliminary estimates for the Wrap and Broadcast/Reduce architectures specify similar data rates [Z90A, Z90B]. A prototype VLSI chip for sliding-window compression (similar to Wrap approach) implementing nine processing elements was fabricated using CMOS 2-micron technology [S92]. Assuming a 40 MHz clock, the chip yields a data rate of 20 Mbytes per second. A systolic array for adaptive dictionary compression using approximately 4,000 processing elements on 30 custom chips has been fabricated using 1.2-micron CMOS and can operate at 40Mbytes per second [SR91]. A custom VLSI chip for the adaptive dictionary compression architecture of the last section built using 1.0 micron CMOS technology operates with a 20 MHz clock, consumes 8 bits per cycle, and achieves a data rate of 20 Mbytes per second [SR91, S92]. Each chip houses 128 processors. A complete systolic array of 3,839 processors is obtained by chaining 30 chips together. Based on simulations, a clock rate of 75 MHz is feasible, yielding a data rate of 75 Mbytes per second. A new design, incorporating techniques for isolating pad delays, is estimated to run at 40 Mbytes per second and has 256 processing elements per chip [S92].

5. MULTIPLE COMPRESSION

The parallel systems of Sections 2 and 3 utilize a single compression method that manipulates the data in parallel to improve compression speed. An alternative application of parallelism combines multiple compression techniques operating simultaneously to obtain greater compression. Current approaches to this form of parallel compression are pipelining and competitive processing.

Pipelined compression systems combine two or more compression methods in succession to compress the input more effectively than the individual methods operating in isolation. Communication applications are an example of systems for which the additional time used to carry out the sequence of methods is insignificant in view of the relatively slow communication speed [PMK91]. Research has focused on the selection of appropriate methods and their optimal positioning in the pipeline.

Statistical methods take advantage of character redundancy whereas dictionary methods profit from string repetition. This distinction is the basis of current pipelined compression systems [BM90, PM90, PMK91]. Dictionary methods maintain a table of strings and compression is carried out by replacing repeated strings by references (or indices) into the table. The fixed-length indices that are produced by the dictionary compressor may contain repeated copies of the same index. This corresponds to strings in the table appearing more than once in the input. By treating the dictionary indices as the input alphabet to a statistical compressor,

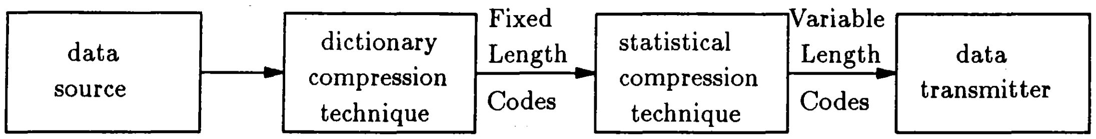

Figure 5

Example of a pipelined compression scheme

additional compression may be gained by replacing frequently occurring indices with shorter variable-length representations. Figure 5 illustrates this pipelined approach.

Several dictionary-statistical-pipelined compression systems have been investigated. Reported methods have paired LZW, a variant of Ziv-Lempel compression and the basis of the UNIX *compress* utility, with statistical compressors based on splay trees, Huffman coding, and arithmetic coding [W84, J88, BM90, PM90, PMK91]. Bailey and Mukkamala conclude that the LZW-Splay method provides better compression of 6% on average than either method in isolation when applied to their test files.

The LZW-Arithmetic coding combination works by accumulating frequencies of LZW dictionary entries and later using these frequencies in the arithmetic coding phase to further compress the representation. Table 2 reports compression findings for LZW-Arithmetic coding. UNIX *compress* version 4.0, paired with the arithmetic coding implementation due to Bell, Cleary and Witten [BCW90], is referred to as *Compress*-AC in the table. The test files used belong to the Calgary/Canterbury text compression corpus [BCW90]. The best compression for each file is shown in bold type. Except for the two program files progl and progp, the pipelined method *Compress*-AC provided slightly better compression than *compress*. Significantly better results are reported for fine-tuned versions of pipelined LZW-Arithmetic coding. By utilizing various LZW dictionary sizes and

Input File	Size (Bytes)	*compress*	Arithmetic Coding	*Compress*-AC	Improvement of *Compress-AC* over *compress*
bib	111261	58.18	34.58	**58.28**	.1
book1	768771	56.81	43.17	**57.04**	.23
book2	610856	58.95	40.29	**59.20**	.25
geo	102400	24.05	**29.30**	27.72	3.67
news	377109	51.71	35.17	**52.12**	.41
obj1	21504	34.67	25.42	**36.97**	2.3
obj2	246814	47.87	24.12	**49.01**	1.14
paper1	53161	52.83	37.70	**52.90**	.07
paper2	82199	56.01	42.17	**56.10**	.09
paper3	46526	52.36	41.12	**52.47**	.11
pic	513216	87.88	85.42	**87.96**	.08
progc	39611	51.67	34.56	**51.78**	.11
progl	71646	**62.11**	40.51	62.08	<.03>
progp	49379	**61.10**	38.82	61.04	<.06>
trans	93695	59.19	31.35	**59.28**	.09

Table 2

Listing of compression delivered by *Compress* and Arithmetic coding

applying different methods of maintaining the cumulative frequency table, Perl, Maram and Kadakuntla report compression increases of as much as 21% on their test files over LZW or arithmetic coding in isolation [PMK91]. They conclude that the pipelined version yields greater compression for smaller dictionary sizes and is especially effective in compressing binary text for which LZW achieves only small compression.

Competitive-parallel processing employs several processors, each concurrently executing a different compression method [C90]. In this MISD system, the output stream of the processor achieving the best compression is selected by the referee processor and transmitted (see Figure 6). Information is relayed with the compressed input to enable the appropriate decompression processor to reconstruct the original data. Competitive-parallel processing is used in file archiving. The

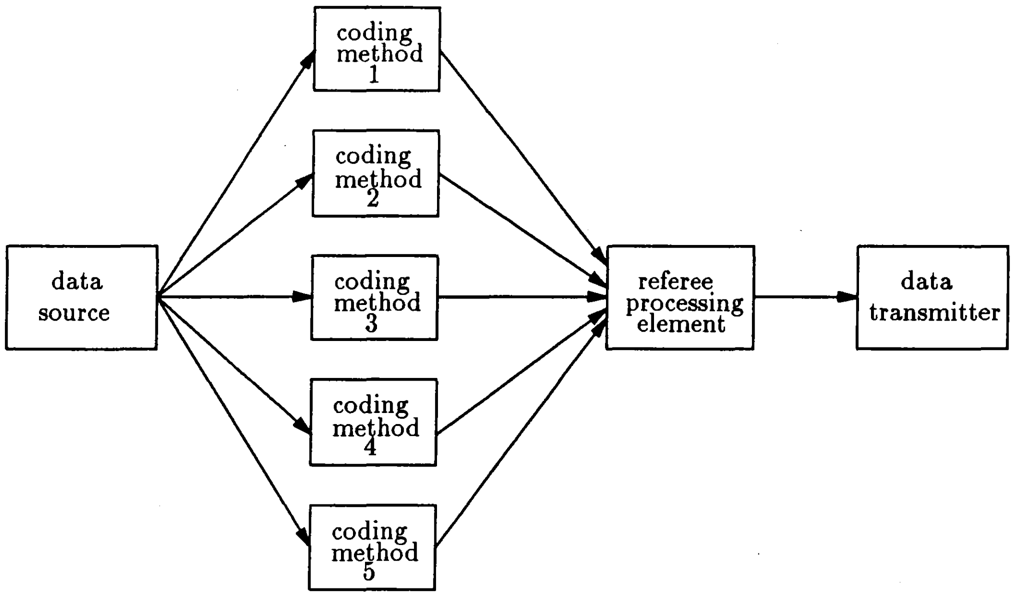

Figure 6

Multiple compression techniques competing for best compression performance

program ARC, a popular archiver for personal computers, analyzes the compression effectiveness of four compression schemes before compressing a file using the best method [BCW90, pp 17–18].

6. FUTURE RESEARCH

In this survey, the use of parallelism in text compression has been described and compared. The distinct goals of speed and compression effectiveness have yielded different applications of parallelism. To create faster systems, parallelism has been applied to the modeling and coding tasks. To improve compression effectiveness, work has focused on compression systems using multiple methods operating in parallel. In this section, we highlight a number of important problems that remain unanswered in the area of parallel text compression.

Empirical evaluation is a fundamental component of parallel text compression research just as it is for sequential compression. Unfortunately, existing evaluations are not based on uniform criteria. Fabricating chips, implementing algorithms on existing parallel machines, and building special purpose hardware are, however, formidable and often prohibitive tasks. Aside from actual implementation, simulations provide meaningful evaluation. Also, detailed designs of processor components and timing requirements are informative. To facilitate comparative studies of speed, compression, and other application objectives, a common corpus of test files must be established. Augmenting the Calgary/Canterbury compression corpus [BCW90] with a collection of multi-megabyte files is a reasonable starting point for standardizing the evaluation of practical parallel algorithms and architectures.

Statistical compression involves the tasks of modeling and coding. Parallel statistical modeling has not been investigated. Context modeling, for example, 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 relatively small. However, if the preceding character is a "q" the probability of the next letter being a "u" is quite high. Perhaps parallelism can improve the models by considering several contexts simultaneously.

For statistical coding, 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 [LH90]. Parallel construction of length-limited Huffman codes remains an open problem.

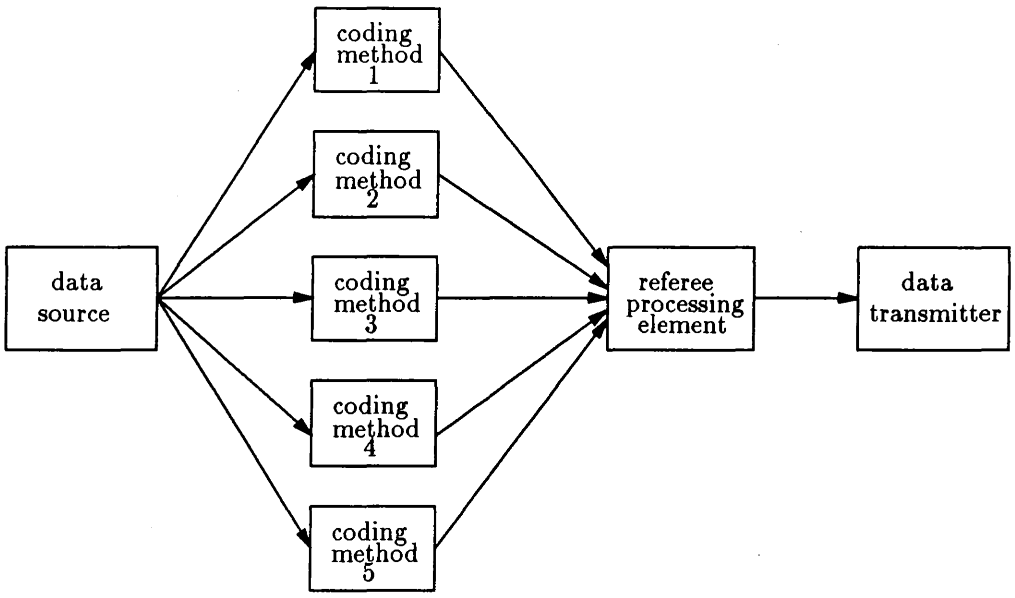

Figure 6

Multiple compression techniques competing for best compression performance

program ARC, a popular archiver for personal computers, analyzes the compression effectiveness of four compression schemes before compressing a file using the best method [BCW90, pp 17–18].

6. FUTURE RESEARCH

In this survey, the use of parallelism in text compression has been described and compared. The distinct goals of speed and compression effectiveness have yielded different applications of parallelism. To create faster systems, parallelism has been applied to the modeling and coding tasks. To improve compression effectiveness, work has focused on compression systems using multiple methods operating in parallel. In this section, we highlight a number of important problems that remain unanswered in the area of parallel text compression.

Empirical evaluation is a fundamental component of parallel text compression research just as it is for sequential compression. Unfortunately, existing evaluations are not based on uniform criteria. Fabricating chips, implementing algorithms on existing parallel machines, and building special purpose hardware are, however, formidable and often prohibitive tasks. Aside from actual implementation, simulations provide meaningful evaluation. Also, detailed designs of processor components and timing requirements are informative. To facilitate comparative studies of speed, compression, and other application objectives, a common corpus of test files must be established. Augmenting the Calgary/Canterbury compression corpus [BCW90] with a collection of multi-megabyte files is a reasonable starting point for standardizing the evaluation of practical parallel algorithms and architectures.

Statistical compression involves the tasks of modeling and coding. Parallel statistical modeling has not been investigated. Context modeling, for example, 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 relatively small. However, if the preceding character is a "q" the probability of the next letter being a "u" is quite high. Perhaps parallelism can improve the models by considering several contexts simultaneously.

For statistical coding, 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 [LH90]. Parallel construction of length-limited Huffman codes remains an open problem.

Most of the work in statistical coding is under the PRAM model of parallel computation and focuses on the parallel construction of codes. More practical approaches are of interest. For example, can a minimum-redundancy prefix code be constructed efficiently using a systolic architecture? Also, there are no known parallel methods for adaptive statistical coding methods, such as dynamic Huffman coding and arithmetic coding [K82, V87, RL79, L84, WNC87]. Also, an adaptive coding system capable of operating at a data rate that is commensurate with the systolic list compression systems can be used to compress the output of dictionary compression methods.

The static and adaptive models for dictionary compression use primarily the systolic array as the model of computation. Dictionary compression systems need to be developed for alternative parallel models, such as the hypercube. In the theoretical setting, sublinear time parallel algorithms are possible on models allowing more than one character to be read and written per unit time. Since dynamic dictionary compression is P-complete, it is highly unlikely that parallel methods belonging to the class $\mathcal{NC}$ exist. Instead, it may be possible to find $\mathcal{NC}$ solutions for methods which approximate dynamic dictionary compression. Also, the PRAM algorithms for static and sliding-window dictionary compression are not optimal.

The discussion in Section 4 illustrates the impact of pipelined and competitive parallel systems on compression. A different approach to utilizing a number of different compression algorithms involves switching from one method to another when system performance begins to deteriorate. For example, move-to-front list compression performs well on small files and transpose compression is superior for large files [SH92A, SH92B, TW89]. This suggests the examination of a hybrid scheme combining move-to-front and transpose. Research into the dynamic evaluation of data and the use of the evaluation to determine the compression method that gives the best system performance may provide a valuable system that is capable of achieving enhanced compression.

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., 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.

[BB92] Bunton, S. and Borriello, G. Practical dictionary management for hardware data compression. *Commun. ACM 35*, 1 (Jan., 1992), 95–104.

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

[D91] De Agostino, S. P-Complete problems in data compression. Tech. Rep. URLS-DM/NS-90/001(INFO). Dept. of Mathematics, University of Rome "La Sapienza", Italy.

[DS92] De Agostino, S. and Storer, J. A. Parallel algorithms for optimal compression using dictionaries with the prefix property. In *Proceedings IEEE Data Compression Conference*, Snowbird, Utah, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 52–61.

[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.

[FG89] Fiala, E. R. and Greene, D. H. Data compression with finite windows. *Commun. ACM 32*, 4 (Apr., 1989), 490–505.

[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 Tenth Annual ACM Symposium on Theory of Computing*, 1978, pp. 114–118.

[F93] Frenkel, K. A. An interview with Robin Milner. *Commun. ACM 36*, 1 (Jan., 1993), 90–97.

[GR88] Gibbons, A. M. and Rytter, W. *Efficient Parallel Algorithms*, Cambridge University Press, Cambridge, 1988.

[G78] Goldschlager, L. M. A unified approach to models of synchronous parallel machines. In *Proceedings Tenth Annual ACM Symposium on Theory of Computing*, 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.

[HR85] Hartman, A. and Rodeh, M. Optimal parsing of strings. In *Combinatorial Algorithms on Words*, A. Apostolico and Z. Galil, Ed., Springer-Verlag, pp. 155–167.

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

[HH85] Hester, J. H. and Hirschberg, D. S. Self-organizing linear search. *ACM Comp. Sur. 17*, 3 (Sep., 1985), 295–311.

[HC87] Horspool, R. N. and Cormack, G. V. A locally adaptive data compression scheme. Technical Correspondence. *Commun. ACM 30*, 9 (Sept., 1987), 792–794.

[HV92a] Howard, P. G. and Vitter, J. S. Parallel lossless image compression using Huffman and arithmetic coding. In *Proceedings IEEE Data Compression Conference*, Snowbird, Utah, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 299–308.

[HV92b] Howard, P. G. and Vitter, J. S. Practical implementations of arithmetic coding. In *Image and Text Compression*, Storer, J. A., Ed., Kluwer Academic Publishers, Norwell, MA, 1992, pp. 85–112.

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

[J88] Jones, D. W. Application of splay trees to data compression. *Commun. ACM 31*, 8 (Aug., 1988), 996–1007.

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

[K82] Knuth, D. E. Dynamic Huffman coding. *J. Algorithms 6* (1982), 163–180.

[L75] Ladner, R. E. The circuit value problem is log-space complete for P. *SIGACT News 7* (1975), 18–20.

[L84] Langdon, G. G. An introduction to arithmetic coding. *IBM J. Research and Development 28*, 2 (Mar., 1984), 135–149.

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

[LH90] Larmore, L. L. and Hirschberg, D. S. A fast algorithm for optimal length-limited Huffman codes. *J. ACM 37*, 3 (July, 1990), 464–473.

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

[L92] Leighton, F. T. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann Publishers, San Mateo, CA, 1992.

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

[MRS92] Markas, T., Reif, J. and Storer, J. A. On parallel implementations and experimentations of lossless data compression algorithms. In *Proceedings IEEE Data Compression Conference*, Snowbird, Utah, IEEE Computer Society Press, Los Alamitos, CA, 1992, p. 425.

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

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

[MP88] Mitchell, H. L. and Pennebaker, W. B. Optimal hardware and software arithmetic coding procedures for the Q-coder. *IBM J. Research and Development 32*, 6 (Nov., 1988), 727–736.

[PMK91] Perl, Y., Maram, V., and Kadakuntla, N. The cascading of the LZW compression algorithm with arithmetic coding. In *Proceedings IEEE Data Compression Conference*, Snowbird, Utah, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 277–286.

[PM90] Perl, Y. and Mehta, A. Cascading LZW algorithm with Huffman coding method: a variable to variable length compression algorithm. In *Proceedings First Great Lakes Computer Science Conference*, Kalamazoo, 1990.

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

[RH91] Ranganathan, N. and Henriques, S. A systolic architecture for LZ based decompression. In *Proceedings IEEE Data Compression Conference*, Snowbird, Utah, IEEE Computer Society Press, Los Alamitos, CA, 1991, p. 450.

[RH92] Ranganathan, N. and Henriques, S. High speed VLSI designs for Lempel-Ziv compression. to appear. *IEEE Trans. on Circuits and Systems.*

[RS91] Reif, J. H. and Storer, J. A. Adaptive lossless data compression over a noisy channel. In *Proceedings Communication Security, and Sequences Conference*, Positano, Italy, 1991.

[RL79] Rissanen, J. J. and Langdon, G. G. Arithmetic coding. *IBM J. Research and Development 23*, 2 (Mar., 1979), 149–162.

[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.

[R87] Ryabko, B. Y. A locally adaptive data compression scheme. Technical Correspondence. *Commun. ACM 30*, 9 (Sept., 1987), p. 792.

[SH92A] Stauffer, L. M. and Hirschberg, D. S. Transpose coding on the systolic array. In *Proceedings IEEE Data Compression Conference*, Snowbird, Utah, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 162–171.

[SH92B] Stauffer, L. M. and Hirschberg, D. S. Self-organizing lists on the Xnet. Tech. Rep. 92-81. Info. and Comp. Sci. Department, University of California, Irvine.

[S92] Storer, J. A. Massively parallel systolic algorithms for real-time dictionary-based text compression. In *Image and Text Compression*, Storer, J. A., Ed., Kluwer Academic Publishers, Norwell, MA, 1992, pp. 159–178.

[S88A] Storer, J. A. *Data Compression Methods and Theory*, Computer Science Press, Rockville, MD, 1988a.

[S88B] Storer, J. A. Parallel algorithms for on-line dynamic data compression. In *Proceedings IEEE International Conference on Communications: Digital Technology – Spanning the Universe*, 1988b, pp. 385–389.

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

[SR91] Storer, J. A. and Reif, J. H. A parallel architecture for high-speed data compression. *J. Parallel and Distr. Comp. 13* (1991), 222–227.

[SRM90] Storer, J. A., Reif, J. H., and Markas, T. A massively parallel VLSI design for data compression using a compact dynamic dictionary. In *Proceedings IEEE VLSI Signal Processing Conference*, San Diego, CA, 1990.

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

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

[TW87] Teng, S. H. and Wang, B. Parallel algorithms for message decomposition. *J. 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.

[V87] VITTER, J. S. Design and analysis of dynamic Huffman codes. *J. ACM 34*, 4, Oct., 825–845.

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

[W91] WILLIAMS, R. N. *Adaptive Data Compression*, Kluwer Academic Publishers, Norwell, MA, 1991.

[W92] WILLIAMS, R. N.. Comp.compression.research bulletin board posting

[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 Second IEEE Symposium on Parallel and Distributed Processing*, Dallas, TX, 1990a, pp. 174–181.

[Z90B] ZITO-WOLF, R. J. A systolic architecture for sliding-window data compression. In *Proceedings IEEE VLSI Signal Processing Conference*, San Diego, CA, 1990b, pp. 339–351.

[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.