Dictionary Compression on the PRAM

Lynn M. Stauffer and Daniel S. Hirschberg
University of California, Irvine
Irvine, CA 92717-3425
stauffer@ics.uci.edu, dan@ics.uci.edu

Technical Report 94-7

January 18, 1994

ABSTRACT

Parallel algorithms for lossless data compression via dictionary compression using optimal, longest fragment first (LFF), and greedy parsing strategies are described. Dictionary compression removes redundancy by replacing substrings of the input by references to strings stored in a dictionary. Given a static dictionary stored as a suffix tree, we present a CREW PRAM algorithm for optimal compression which runs in $O(M + \log M \log n)$ time with $O(nM^2)$ processors, where it is assumed that M is the maximum length of any dictionary entry. Under the same model, we give an algorithm for LFF compression which runs in $O(\log^2 n)$ time with $O(n/\log n)$ processors where it is assumed that the maximum dictionary entry is of length $O(\log n)$. We also describe an $O(M + \log n)$ time and $O(n)$ processor algorithm for greedy parsing given a static or sliding-window dictionary. For sliding-window compression, a different approach finds the greedy parsing in $O(\log n)$ time using $O(nM \log M/\log n)$ processors. Our algorithms are practical in the sense that their analysis elicits small constants.

1. Introduction

We present parallel algorithms for dictionary compression using optimal, longest fragment first (LFF), and greedy parsings. Dictionary methods, also called textual substitution and Ziv-Lempel compression, achieve compression by replacing strings with references to some dictionary of strings [ZL77, ZL78]. A dictionary of characters, words, or phrases that are expected to occur frequently is maintained and a recurring substring is converted (encoded) into a compact representation by replacing the substring with the index of its corresponding dictionary entry. Compression is achieved by choosing indices so that on average they require less space than the phrases they encode. Decoding or decompression restores the original text from its compressed representation.

In this paper, we consider algorithms for dictionary compression on the parallel random access memory (PRAM) model of parallel computation [J92]. The PRAM consists of a number of identical general-purpose sequential processors that communicate through a large shared, random access memory. Variants of the PRAM model differ in their handling of simultaneous reading and writing of the global memory. The Concurrent-Read, Concurrent-Write (CRCW) model allows processors to read and write the shared memory simultaneously, while the Concurrent-Read, Exclusive-Write (CREW) model forbids simultaneous writes. CRCW models are further distinguished by their methods for handling write conflicts. The Common CRCW permits concurrent writes only when all processors are attempting to write the same value to the same block; the Arbitrary CRCW allows an arbitrary processor to succeed in a concurrent write; the Priority CRCW assumes processors are assigned unique ordered priorities and, when several processors attempt to write to a single location, the processor with the highest priority carries out its write instruction.

In the parallel environment, dictionary compression schemes have been developed using a systolic array [GS85, SR91, SH92, SH93]. A systolic array consists of a collection of linearly-connected processors where input enters at one end of the array and output exits at the other [L92]. This model requires linear time to handle input and output. These parallel compression systems are optimal for the given model in the sense that they work on-line in linear time using a linear number of processors. The PRAM models allow blocks of data to be read or written in a single time step and so sublinear time data compression is possible.

We describe our algorithms in the context of text compression. Dictionary methods can be used to compress data other than text (e.g. images) and our methods extend straightforwardly. We consider *lossless* compression which stipulates that the decompressed data must be identical to the original input stream. It is further assumed that the communication channels and storage devices are noiseless. We refer the reader to [S88], [BCW90], and [W91] for a thorough coverage of sequential text compression and to [SH93] for a survey of parallel text compression.

Dictionary compression methods can be static or adaptive. A *static* method creates a fixed dictionary before compression begins. An *adaptive* (also called *dynamic*) method allows changes to the dictionary during compression. A popular adaptive approach based on the work of Lempel and Ziv [ZL78] (often called LZ78 or LZ2 compression) parses the input into phrases which are compared to a dynamic dictionary. The UNIX *compress* utility implements a variation of LZ78 due to Welch (referred to as LZW). The discovery of $\mathcal{NC}$[1] algorithms for LZ78 is unlikely since LZ78 and two LZ78 variations are known to be P-complete[2] [D91]. The P-complete LZ78 variants implement different parsing and dictionary update strategies [W84, S88]. Ziv and Lempel's earlier approach limits references to a window of input characters preceding the input position being considered for encoding [ZL77]. The dictionary is implicitly represented by a window that slides continuously over the input. Encoding references consist of $(position, length)$ pairs that indicate a substring in the window. These methods are referred to variously as LZ77, LZ1, and sliding-window compression.

Once the dictionary has been selected, the input stream must be parsed to determine which substrings are to be replaced by dictionary indices or references. A position in the input stream is a *breakpoint* if a substring starting at that position is to be replaced by a dictionary reference. A *parsing* of the input determines the set of breakpoint positions and the corresponding sequence of substrings that will be represented by references and results in determining the sequence of dictionary references. An *optimal* parsing of the input is a parsing whose resulting sequence of dictionary references has the most succinct representation. If it is assumed that

[1] 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 input size using a polynomially-bounded number of processors [J92].

[2] 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 $\mathcal{P} \neq \mathcal{NC}$ [J92].

Input: compression ratios measure compression

Dictionary D = {press,
comp, pres, sion,
asu, com, eas, ure,
io, me, on, ra,
a, c, e, i, m, n, o, p, r, s, t, u, <blank>}

Greedy: comp/r/e/s/sion/ /ra/t/io/s/ /me/a/s/ure/ /comp/r/e/s/sion

LFF: com/press/i/on/ /ra/t/io/s/ /m/eas/ure/ /com/press/i/on

Optimal: com/pres/sion/ /ra/t/io/s/ /m/eas/ure/ /com/pres/sion

Figure 1

An example of the greedy, LFF and optimal parsings

dictionary references are of fixed-length then an optimal parsing is one with the fewest number of references. For variable-length references, an optimal parsing is not necessarily one with the fewest number of references. A more straightforward approach is *greedy* parsing where iteratively the encoder finds the longest dictionary phrase that matches a prefix of the uncoded portion of the input stream and the index of that dictionary entry is used to encode the input prefix. The *longest fragment first*, or LFF, algorithm parses the input by repeatedly locating the longest substring of the uncoded portion of the input which matches a dictionary entry and replacing it with the corresponding dictionary reference. This process continues until the input is completely replaced by references.

Greedy, LFF, and optimal parsing with fixed-length references are illustrated in Figure 1. In general, the compression performance of LFF lies between greedy and optimal parsing [SH73]. However, to determine an optimal or LFF parsing, a sequential encoder must perform two passes over the input or be capable of looking at arbitrarily large prefixes of the input. Consequently, greedy parsing is widely used in sequential compression systems since it requires only limited look-ahead and is computed on-line. In the parallel environment, the practicality of sequential greedy parsing is no longer relevant. The PRAM model lifts the performance limitations of optimal and LFF parsing by making the entire input string available for computation in a single step.

In this paper, we describe parallel algorithms for static dictionary compression subsequent to the selection of strings represented in the dictionary (see [CL82] and [SH73] for discussions of the dictionary creation problem). We assume that the dictionary contains d entries, each of length less than or equal to some maximum length M. We also assume that the input string is processed in blocks of n characters. To guarantee a feasible compressed representation, the dictionary is required to include all members of the input character set.

The remainder of the paper is organized as follows. In Section 2, we review previous parallel work on greedy, LFF and optimal parsing for dictionary compression. Section 3 presents an $O(M + \log M \log n)$ time and $O(nM^2)$ processor algorithm for optimal dictionary compression with respect to a static dictionary satisfying the assumptions stated above. An $O(M \log n)$ time and $O(n/\log n)$ processor algorithm for LFF based dictionary compression with respect to a static dictionary is given in Section 4. When $M = O(\log n)$ this algorithm runs in $O(\log^2 n)$ time. Greedy parsing is described in Section 5, for both static and sliding-window dictionary compression. We conclude the paper with a discussion of other aspects of parallel compression and possible extensions to this work.

2. Previous work

Table 1 summarizes previously known results and those presented in this paper. Some algorithms assume that the dictionary satisfies the prefix property (notated "Pref" in Table 1), i.e., all prefixes of each dictionary entry are present in the dictionary. Many of the results assume that the dictionary is represented as a suffix tree. (These results are notated by "*" in Table 1.) A suffix tree for a dictionary D is a trie composed of all the strings in D. Every string in D corresponds to a path in the suffix tree in a natural way. Associated with each node in the tree is the substring (not necessarily in D) consisting of the concatenation of the substrings of the edges along the path from the root to the node. For our purposes, we require that a node corresponding to a string in D stores a pointer into a table of references. This information can be easily added in the suffix tree construction stage which builds the suffix tree representing the static dictionary in $O(\log(dM))$ time using $O(dM)$ processors on the Arbitrary CRCW [AILSV88].

Notation:
d = number of strings in the dictionary
M = maximum length of any dictionary entry
n = input size
Pref = prefix property
* = requires suffix tree dictionary
exp. = expected time complexity (randomized algorithm)

Parsing	Model/Assumptions	Time	Processors	Reference
STATIC DICTIONARY				
Optimal	Common CRCW, Pref, *	$O(M+\log n)$	$O(n^2)$	[DS92]
		$O(M+\log^2 n)$	$O(n)$	[DS92]
	Common CRCW, Pref	$O(\log n)$	$O(n^2 d)$	[DS93]
	CREW, *	$O(M+\log^2 n)$	$O(n^3)$	[DS92]
		$O(M+\log n)$	$O(n^4)$	[DS92]
		$O(M+\log M \log n)$	$O(nM^2)$	[§2]
	Arbitrary CRCW, Pref	exp. $O(\log n)$	$O(dM+n)$	[DS93]
Greedy	CREW	$O(M+\log n)$	$O(n)$	[§4.1]
	Arbitrary CRCW, *	exp. $O(\log n)$	$O(dM+n)$	[DS93]
LFF	CREW,*	$O(M \log n)$	$O(n/\log n)$	[§3.2]
UNRESTRICTED SLIDING-WINDOW				
Greedy	CREW	$O(M+\log n)$	$O(n)$	[§4.2]
		$O(\log n)$	$O(n^2 \log M/\log n)$	[§4.3]
		$O(M+\log n)$	$O(n^2)$	[DS92]
		exp. $O(\log n)$	$O(n)$	[N91]
	Priority CRCW	$O(M+\log n)$	$O(n)$	[DS93]
FIXED-SIZE SLIDING-WINDOW				
Greedy	CREW	$O(\log n)$	$O(nM \log M/\log n)$	[§4.3]

Table 1

Parallel parsing results for static and sliding-window compression

3. Optimal Parsing

In this section, we describe PRAM algorithms for optimal compression. We assume that the static dictionary is arbitrary and is represented as a suffix tree.

Computing the optimal parsing can be transformed into the problem of finding a shortest path in a graph [BCW90]. The optimal parsing can be computed in $O(M+\log^2 n)$ time using $O(n^3)$ processors or in $O(M+\log n)$ time using $O(n^4)$ processors using that reduction [DS92]. For input $X = x_1 \cdots x_n$, the transformation views each character as a vertex in a graph and a directed edge from x_i to x_j is in the graph if the string $x_i \cdots x_j$ is in the dictionary. We show how

to obtain improved complexity bounds. We first consider fixed-length dictionary references and the shortest path is in terms of the number of edges that have to be traversed. In the case of variable-length references, edges are assigned weights equal to the length of the corresponding dictionary reference and the shortest path is in terms of the sum of the edge weights of the edges traversed.

The input is divided into n/M blocks, $B_1, \ldots, B_{n/M}$. Block B_l consists of M characters, $x_{M(l-1)+1} \cdots x_{Ml}$. The first step of the algorithm determines the matches between strings beginning at each input position and the dictionary. This is equivalent to adding the directed edges to the associated graph. This can be done in $O(M)$ time using $O(n)$ processors by assigning a processor to each position and stepping sequentially down the suffix tree. Define C to be an $n \times n$ matrix which is updated in a sequence of $O(\log(n/M))$ iterations. C is initialized using the dictionary match information. That is, if $x_i \cdots x_j$ is in the dictionary, then $C[i,j] = 1$. Otherwise, $C[i,j] = 0$. Next, the shortest path is computed for all pairs of positions belonging to the same or adjacent blocks. For all pairs (i,j), within one block or one pair of blocks, this can be done in $O(\log^2 M)$ time using $O(M^3)$ processors. For all n/M groups, this computation can be done in $O(\log^2 M)$ time using $O(nM^2)$ processors.

At this point, $C[i,j]$ is the length of the shortest path between x_i and x_j for $1 \leq i \leq n$ and $i \leq j \leq \min\{i+M, n\}$. The remainder of the algorithm consists of an iterated step. Within each iteration, the search for paths is extended to consider all paths between positions with up to twice the number of intermediate blocks of the previous iteration. In the first iteration the shortest path between vertices in blocks B_l and B_{l+2} is computed. The maximum dictionary entry length, M, forces the shortest path from any position in B_l to pass through a vertex in B_{l+1}. Since the shortest path between every position in B_{l+1} and every position in B_{l+2} is known, the shortest path between any x_i in B_l and x_j in B_{l+2} is computed considering all positions in B_{l+1} as intermediate vertices and taking the minimum over the resulting paths. More formally,

$$C[i,j] = \min_{t \in B_{l+1}} \{C[i,t] + C[t,j]\}$$

In iteration k, the shortest paths between blocks B_l and B_{l+2^k} passing through block $B_{l+2^{k-1}}$ are computed and added to matrix C. In each iteration, the minimum computation can be completed in $O(\log M)$ time using $O(M^3)$ processors for each

group. Totaling over all n/M groups, each iteration runs in $O(\log M)$ time using $O(nM^2)$ processors. All $O(\log(n/M))$ iterations require $O(\log M \log n)$ time.

Combining this with the initialization phase, the shortest path or, equivalently, the optimal parsing, can be computed in in $O(M + \log M \log n)$ time using $O(nM^2)$ processors.

The algorithm for fixed-length dictionary references, described above, essentially computes the shortest path in terms of the number of edges. In the case of variable-length references, the edge joining x_i to x_j is assigned a weight equal to the length of the dictionary reference for dictionary entry $x_i \cdots x_j$. The algorithm above can be generalized to compute the shortest path in terms of the sum of the weights of the edges traversed. Thus, the optimal parsing with variable-length references can be computed in $O(M + \log M \log n)$ time using $O(nM^2)$ processors.

4. LFF Compression

In this section, we describe $\mathcal{NC}$ algorithms for dictionary compression based on the LFF parsing heuristic. We assume that the static dictionary is stored as a suffix tree and that dictionary references are fixed in length. Interval graphs are introduced in Section 4.1 for use in the formulation of later algorithms.

4.1. Interval Graphs

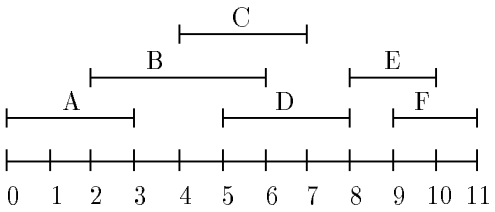
Interval graphs are a useful discrete mathematical structure for modeling many problems with restrictions that are linear in nature. Scheduling, VLSI layout and biology are among the myriad of applications modeled by interval graphs. Interval graphs consist of a set of vertices associated with intervals of a linearly ordered set, such as the real line. Edges join vertices whose corresponding intervals intersect or overlap.

More formally, an interval graph G=(V,E), associated with the set of intervals $\mathrm{I} = \{\mathrm{I}_i = [a_i, b_i] \mid a_i \leq b_i, 1 \leq i \leq n\}$, consists of vertex set $\mathrm{V} = \{\mathrm{I}_i\}$ and edge set $\mathrm{E} = \{(\mathrm{I}_i, \mathrm{I}_j) \mid \mathrm{I}_i \cap \mathrm{I}_j \neq \emptyset\}$ [G80]. Interval $[a, b]$ is said to have *left* endpoint a and *right* endpoint b. For the set I, define *first*(I) as the interval whose right endpoint is furthest to the left. Namely, *first*(I) $= \mathrm{I}_k$ such that $b_k = \min_i\{b_i\}$. By finding the minimum of the right endpoints, *first*(I) can be found in optimal $O(\log n)$ time using $O(n/\log n)$ processors [BB87]. For each interval I_i define the next non-overlapping interval *next*(I_i) to be the interval ending furthest to the left among the intervals beginning after interval I_i. That is, *next*(I_i) $= \mathrm{I}_k$ if $b_k = \min_j\{b_j \mid b_i < a_j\}$.

I = {A[0,3], B[2,6], C[4,7], D[5,8], E[8,10], F[9,11]}

first(I) = A

next(A) = C, next(B) = E, next(C) = E



G = (V, E)

V = {A, B, C, D, E, F}

E = {(A, B), (B, C), (B, D), (C, D), (E,F)}

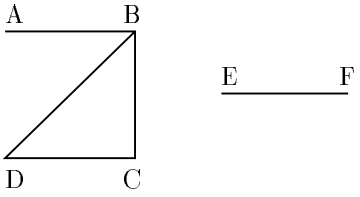


Figure 2

Example of corresponding interval and graph models

The parallel computation of $next(\mathrm{I}_i)$, for all i, begins by sorting the $2n$ endpoints into descending order. This initial step requires $O(\log n)$ time and $O(n)$ processors [BB87, J92]. The remainder of the *next* computation involves a prefix minimum operation and a few comparisons. Thus, the sorting step dictates the $O(\log n)$ time and $O(n)$ processor bounds for finding $next(\mathrm{I})$. An interval graph G(I) and its corresponding interval model I(G) are pictured in Figure 2. We shall describe some results in terms of the interval model. Moreover, our algorithms instantiate intervals that contain their endpoints and that do not share any common endpoint.

Many standard graph-theoretic problems known to be NP-hard for general graphs can be solved in polynomial time on interval graphs. An *independent set* in a graph consists of a set of vertices, no two of which are adjacent. For an interval graph, an independent set corresponds to a collection of non-intersecting intervals. In parallel on the EREW PRAM, the *maximum size* independent set (or equivalently a *largest cardinality* set of non-overlapping intervals) can be computed in $O(\log n)$ time and $O(n)$ processors [OSZ92]. This solution builds on optimal computations of *first* and *next* as can be seen in the outline of the parallel maximum independent set algorithm in Figure 3.

4.2. LFF Parsing

Figure 4 gives a high-level description of our parallel LFF parsing algorithm. The algorithm begins by determining the lists of lengths of matches between the dictionary and the string beginning at each position of the input. Next, the length, M^*, of the longest dictionary entry appearing in the input is calculated. The

Find Maximum Independent Set

1. Compute *first*(I)
2. Compute *next*(I_i) for all i
3. Considering *next*(I_i) as a pointer, mark all nodes in the path from *first*(I) to the root of the tree containing *first*(I)
4. Return set of marked nodes

Figure 3

Outline of the Maximum Size Independent Set algorithm

LFF parsing

1. Compute lists of dictionary match lengths at each position of input X
2. Compute maximum match length, M^*
3. For each match length $l = M^*$ downto 1 do
   a. Find a maximum collection C of non-overlapping matches of length l
   b. Update match length at input positions overlapping C

Figure 4

Sketch of parallel LFF parsing algorithm

LFF parsing is then found by repeatedly locating a maximum collection of longest fragments and updating the match information for positions which intersect the current collection. First, we give an $O(M(\log M + \log n))$ time and $O(n)$ processor algorithm. We then improve the processor requirement to $O(n/\log n)$.

A processor is assigned to each position of the input string $X = x_1 x_2 \cdots x_n$ at the start of Step 1. Each processor computes the list m of the lengths of matches between the dictionary and the input beginning at its assigned position. That is, processor P_i computes list m_i by comparing successive strings beginning at x_i to the dictionary. This list is stored in an array of length M. Initially, $m_i[k]$ is set to zero, for $k = 1 \ldots M$. For $j = 1$ to M, P_i determines if string $x_i \cdots x_{i+j-1}$ is in the dictionary by iteratively going down one level in the suffix tree. If $x_i \cdots x_{i+j-1}$ is in the dictionary, length j is added to list m_i by setting $m_i[j] = 1$. In addition, P_i initializes a_i, the length of the longest match permitted at position i, by setting a_i to the length of the longest match occurring at position i. Since the maximum

length of any dictionary entry is $O(M)$, Step 1 runs in $O(M)$ time using $O(n)$ processors.

M^* is the maximum of the a_i values. Thus, Step 2 is completed by a standard maximum computation in $O(\log n)$ time and $O(n/\log n)$ processors [J92].

Observe that by treating matches as intervals, the computation in Step 3a is equivalent to determining a maximum independent set in an interval graph. At the start of each iteration, the maximum collection of non-overlapping matches C is initialized. On a particular iteration of the loop, if list m_i contains l then P_i participates in the maximum independent set computation. List m_i contains l if $l \leq a_i$ and $m_i[l] = 1$. Using the algorithms described in Section 4.1, Step 3a computes C in $O(\log n)$ time using $O(n)$ processors. In Step 3b, match information is updated in $O(\log n)$ time as follows. Each processor P_j whose length l match is in C signals the trailing positions $j+1, \ldots, j+l-1$ covered by its match to become inactive and also signals the $l-1$ preceding positions, $j-l+1 \ldots j-1$, to update their match list m as described below. This signaling can be done in $O(\log M)$ time using pointer jumping [J92]. Inactive processors do not participate in the remainder of the LFF computation. If P_k is signaled by P_j to update its list m_k then P_k sets $a_k = j - k$, essentially eliminating all lengths exceeding $j - k$ from m_k. Step 3b requires $O(\log M)$ time and uses $O(n)$ processors. To complete its $O(M)$ iterations, Step 3 requires a total of $O(M(\log M + \log n))$ time and $O(n)$ processors.

Combining the performance of the separate steps, this first LFF algorithm runs in $O(M(\log M + \log n))$ time using $O(n)$ processors and $O(nM)$ space. Since processors access the dictionary simultaneously, the algorithm uses the CREW PRAM model.

To illustrate our algorithm, consider the dictionary and input string in Figure 1. For dictionary D, initially, $m_i = \{5, 4, 1\}$ for $i = 4, 31$, $m_i = \{4, 3, 1\}$ for $i = 1, 28$, $m_i = \{4, 1\}$ for $i = 8, 35$, $m_i = \{3, 1\}$ for $i = 21, 22, 24$, $m_i = \{2, 1\}$ for $i = 9, 10, 13, 16, 20, 36, 37$ and $m_i = \{1\}$ for all other values of i. The longest match length $M^* = 5$ and for each i, a_i is initialized to the largest value in list m_i. In the first pass of Step 3, $l = 5$ and matches at positions 4 and 31 are included in the LFF parsing. Processors 5, 6, 7, 8, 32, 33, 34, and 35 become inactive and processors 1 and 28 update their match lists (i.e., $m_1 = \{3, 1\}$, $a_1 = 3$, $m_{28} = \{3, 1\}$ and $a_{28} = 3$). In the next pass, $l = 4$ and processing skips to $l = 3$ since there are no active processors with matches of length 4. For $l = 3$, positions

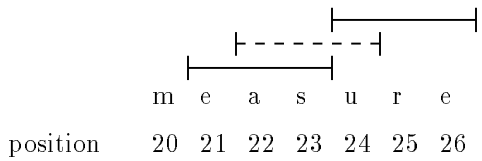


Figure 5

Example of LFF parsing algorithm using dictionary and input of Figure 1

21, 22, and 24 are candidates for C. The maximum independent set C consists of matches at 21 and 24 (see Figure 5 where the solid intervals form the maximum size independent set). Positions 22, 23, 25, and 26 become inactive and processor 20 recomputes $m_{20} = \{1\}$. This process continues for $l = 2$ and $l = 1$ to yield the LFF parsing in Figure 1.

To improve on this approach, consider the processor requirements of the maximum independent set computation in Step 3a (refer to Figures 3 and 4). As mentioned earlier, the parallel maximum independent set algorithm derives $next(\mathrm{I}_i)$ for all intervals I_i, $1 \leq i \leq n$, by initially sorting the interval endpoints. In order to achieve a logarithmic time bound, a linear number of processors are required for the sorting step. However, in our application, this sorting step is not necessary since the intervals are provided in sorted order as a consequence of the problem set up. The processor bound for the *next* computation can be reduced to $O(n/\log n)$. The final marking step in the maximum independent set algorithm (Step 3 in Figure 3) can be done in logarithmic time using $O(n/\log n)$ processors [AM91]. Thus, the maximum independent set calculation in the LFF algorithm can be performed in $O(\log n)$ time with $O(n/\log n)$ processors.

To complete the updates in Step 3b, divide the input into $O(n/\log n)$ groups and assign a processor to each group. The idea is to have each processor deactivate and update the lists of the positions in its block that are affected by the match intervals selected in Step 3a. Positions can be affected by a match in C in two ways. If position j is in C (i.e., string $x_j \cdots x_{j+l-1}$ is a match in the maximum independent set), then positions $j+1, \ldots, j+l-1$ must be deactivated and positions $j-l+1, \ldots, j-1$ must change their match list information. So, in $O(\log n)$ time each processor determines the locations of matches within its assigned block. Step 3b continues with each processor notifying positions affected by matches within its block.

When $l \leq \log n$, any match can affect positions in at most 3 blocks since a match can straddle at most two blocks and a third block of lowered number positions may need to update list information. In parallel, each processor sweeps through positions affected by matches occurring within its block in $O(\log n)$ time. Processor P_i begins at position $(i-1)\log n + 1$ and moves toward higher indexed positions. Upon encountering position j in C, P_i deactivates the next $l-1$ positions, $j+1, \ldots, j+l-1$. When P_i reaches the right end of its block at $x_{i \log n}$, it terminates the sweep if it has not detected a match in positions $i \log n - l, \ldots, i \log n$. If a match was detected in positions $i \log n - l, \ldots, i \log n$ then P_i continues into the right neighboring block until it completes the deactivation of affected positions. Upon completing this lower to higher indices sweep of the input, P_i sweeps from position $i \log n$ to lower indexed positions updating the largest allowable match lengths (a). If P_i detects a match at position j, then P_i sets $a_{j-1} = 1$, $a_{j-2} = 2$, ..., $a_{j-l+1} = l - 1$ and continues the sweep at position $j - l$.

When $l > \log n$, each block can contain the beginning position of at most one match interval and a match interval can span several input blocks. Suppose position j in block i is in C. Using pointer jumping, processor P_i notifies the blocks affected by the match interval $x_j \cdots x_{j+l-1}$ in $O(\log M)$ time. Now, in $O(logn)$ time, each affected processor deactivates positions and updates lists as detailed above.

Finally, consider the processor requirements of Step 1 (Step 2 is already computed optimally with $O(n/\log n)$ processors). Using a grouping approach similar to that described for Step 3b, the input is divided into blocks of $O(\log n)$ positions and a processor is assigned to each block. In parallel, each processor computes the match lists of its assigned positions. This gives an $O(M \log n)$ time and $O(n/\log n)$ processor version of Step 1.

Combining these improvements, our algorithm computes the LFF parsing in $O(M(\log M + \log n))$ time using $O(n/\log n)$ processors. Consider the performance of our algorithm when M is $O(\log n)$. In practice, this is a reasonable assumption since matches in text rarely exceed several characters. For example, when compressing English text, the average match length is roughly 5 characters. Thus, using a dictionary of 64K entries and a maximum entry length of 16 characters does not significantly impact compression. When M is $O(\log n)$, the LFF parsing is computed in $O(\log^2 n)$ time using $O(n/\log n)$ processors. This approach is simple and practical since it has reasonable constant factors. Namely, the optimal parallel

list ranking used in several steps of the algorithm can be done using the algorithm of Anderson and Miller which elicits a reasonable constant [AM91].

5. Greedy Parsing

Greedy parsing sequentially scans the input, locating and removing the longest dictionary entry occurring as a prefix of the uncoded portion of the input. In this section, we consider both static and sliding-window dictionaries. We describe an algorithm for determining the greedy parsing given a static dictionary in $O(M + \log n)$ time using $O(n)$ processors. Later in this section we describe two algorithms for sliding-window compression. Using an approach similar to the one taken for the static dictionary model, the first approach yields an $O(M + \log n)$ time and $O(n)$ processor algorithm on the CREW. The second algorithm runs in $O(\log n)$ time using $O(nM \log M / \log n)$ processors on the CREW.

5.1. Static Dictionary

For greedy parsing, we must determine the longest match between the input beginning at each position and the dictionary. We assume that the dictionary is in the form of a suffix tree. Using $O(M)$ time, a processor assigned to each input position can determine the longest match information by stepping down the suffix tree, beginning at the root.

By assigning a processor to each input position, the breakpoint positions in the parsing are marked as follows. Position 1 is automatically a breakpoint and, by viewing the match lengths as pointers (i.e., if the match length at position i is j, processor P_i is pointing at position $i+j$), the standard pointer doubling technique is used to mark all breakpoints on the path from position 1 to position n in logarithmic time [J92]. Thus, the greedy parsing can be computed in $O(M + \log n)$ time using $O(n)$ processors.

5.2. Sliding-window Dictionary with unrestricted window

In sliding-window compression, the dictionary is implicitly represented for each input position by the characters preceding the position. Many variations of sliding-window compression are possible by imposing restrictions on the size of the window and the set of allowable dictionary strings. When the size of the window is unrestricted, better compression is possible but often at the expense of increased complexity. We describe an algorithm for sliding-window compression with an

unrestricted window. This algorithm uses a suffix tree representation of the input string to carry out compression. In Section 5.3, we describe an algorithm that obtains largest match information based on a fixed-sized window consisting of the previous M characters.

Our algorithm for sliding-window compression with an unrestricted window uses an approach similar to the one described above for static dictionary compression. Here, a suffix tree representation of the *input string* is used to compute the longest match information. The suffix tree is augmented to store the index at each node of the earliest occurrence of the node's corresponding substring[3]. It is not necessary to store the indices of each occurrence of the node's corresponding substring since the earliest occurrence is included in the window of each input string occurring after it. As the longest match computation for a particular input position proceeds, only nodes storing indices less than the input position are considered. Using the augmented suffix tree, the longest match between an input position and the strings in its window can be computed in $O(M)$ time. Thus, all positions can determine their longest match in $O(M)$ time using $O(n)$ processors.

At this point, the greedy parsing is obtained by marking the input positions in the path from position 1 to n (as described earlier). This marking requires $O(\log n)$ time. So, the greedy parsing is computed in $O(M+\log n)$ time using $O(n)$ processors. Using the method shown in Section 5.3, the greedy parsing for an unrestricted sliding window can be computed in time $O(\log n)$ using $O(n^2 \log M/\log n)$ processors.

5.3. Sliding-window Dictionary with fixed-sized window

For input string $X = x_1 \cdots x_n$ and a fixed-sized window of size M, the dictionary at character x_i consists of the previous M characters, $x_{i-M} \cdots x_{i-1}$. Our algorithm for greedy parsing using a fixed-sized sliding-window proceeds in 3 phases. In Phase 1, the length of the longest match between the string beginning at x_i and the strings beginning at each position in x_i's window is computed for all i, $1 \leq i \leq n$. Observe that the matches for x_i are required to begin in x_i's window but may overlap a prefix of the string beginning at x_i. Define L to be an $n \times M$ table which is updated in a series of $\log M$ rounds to compute the longest match information. At the end of round k, $1 \leq k \leq \log M$, $L_k[i,j]$ stores the length of the longest match between the prefixes of strings $x_i \cdots x_{i+2^k-1}$ and $x_{i-j} \cdots x_{i-j+2^k-1}$.

[3] Naor describes this augmentation in [N91].

Initially, $k = 0$ and each input character is compared to each character in its window. In constant time, processors $P_{i,1}, \ldots, P_{i,M}$ compare x_i to each character in x_i's window. If $x_i = x_{i-j}$ then $P_{i,j}$ sets $L_0[i,j] = 1$. Otherwise, $P_{i,j}$ sets $L_0[i,j] = 0$. Using nM processors, table L is initialized for all pairs (i,j) in $O(1)$ time. In round k, $1 \leq k \leq \log M$, matches of length up to 2^k are built from matches of lengths up to 2^{k-1}. For x_i, if $L_{k-1}[i,j] = 2^{k-1}$ then the match between $x_i \cdots x_{i+2^{k-1}-1}$ and $x_{i-j} \cdots x_{i-j+2^{k-1}-1}$ is concatenated with the match between the string beginning at $x_{i+2^{k-1}}$ and the string in x_i's window beginning at $x_{i-j+2^{k-1}}$. Observe that $x_{i-j+2^{k-1}}$ is the jth character in $x_{i+2^{k-1}}$'s window. Otherwise, if $L_{k-1}[i,j] < 2^{k-1}$ then the maximum match between the string beginning at x_i and the string in x_i's window beginning at x_{i-j} cannot be extended. Thus, the following recurrence represents the computation in round k:

$$L_k[i,j] = \begin{cases} L_{k-1}[i,j] + L_{k-1}[i+2^{k-1}, j], & \text{if} \quad L_{k-1}[i,j] = 2^{k-1} \\ L_{k-1}[i,j], & \text{if} \quad L_{k-1}[i,j] < 2^{k-1} \end{cases}$$

and each round can be completed in constant time by assigning a processor to each (i,j) pair. So, Phase 1 requires $O(\log M)$ time and nM processors.

In Phase 2, the longest match of the string starting at x_i among all strings beginning in its sliding-window can be computed in $O(\log M)$ time using $O(M)$ processors by a standard maximum computation of the values in the ith row of table L. Using $O(nM)$ processors, the longest match information can be computed for all positions $x_1, \ldots, x_n$.

In the final phase, standard pointer doubling techniques (as mentioned above) are used to determine the parsing containing the set of longest matches that begins at location 1. Phase 3 can be completed in $O(\log n)$ time using $O(n)$ processors.

Thus, using the sliding-window, we can find the greedy parsing in time $O(\log M + \log n)$ using $O(nM)$ processors. When $M = O(\log n)$, this solution takes $O(\log n)$ time and $O(n \log n)$ processors.

It is possible to reduce the number of processors by spending additional time in Phases 1 and 2. By Brent's Theorem [GR88], increasing the time in Phases 1 and 2 to $O(\log n)$ reduces the processor requirements to $O(nM \log M / \log n)$.

Using this method, the greedy parsing for an unrestricted sliding window can be computed in time $O(\log n)$ using $O(n^2 \log M / \log n)$ processors by noting the following. Phase 1 can be performed in $\log M$ time using n^2 processors or, by Brent's Theorem, in the claimed time-processor bounds. Phase 2 requires a prefix

maximum computation (as opposed to a maximum over a fixed-size window) which can be done in time $O(\log n)$ using $O(n)$ processors.

6. Other Considerations

In this section, we describe how to produce the compressed output from the parsing, decompression, and possibilities for further research.

We have addressed the parsing problem in dictionary compression. After the input string has been parsed, the compressed output consists of the concatenation of the dictionary references corresponding to the substrings in the parsing. A processor, assigned to the start position of each substring in the parsing (i.e., each breakpoint), can determine where in the output stream to write its corresponding dictionary reference by performing a prefix sum computation. This can be done optimally in parallel in time $O(\log n)$ using $O(n/\log n)$ processors [GR88].

Decompression replaces each dictionary reference in the compressed string by its corresponding dictionary entry. If dictionary references are all of fixed-length, the parsing of the compressed stream into individual dictionary references is straightforward (the ith reference starts after stream byte $c(i-1)$, where each reference is c bytes in length). If P_i is assigned to the ith reference, it can determine in time $O(M)$ the length of its represented string and, using a prefix sum computation, the location for each represented string in the original input stream can be computed in $O(\log n)$ time. In $O(M)$ time, each processor writes its represented string to the recovered original stream. For variable-length dictionary references, assigning processors to dictionary references in the output stream, requires additional work. Let r be the maximum length of any dictionary reference. Assign a processor to each output byte and in $O(r)$ time each processor can determine its complete dictionary reference in a way similar to recognition of references in sequential decompression. By treating the reference length information as pointers, the parsing of the output stream into dictionary references can be done using pointer doubling in $O(\log n)$ time and $O(n)$ processors. Now, the decompression problem is the same as for fixed-length references and the method above combined with the preprocessing yields the recovered original stream in $O(r + M + \log n)$ time using $O(n)$ processors.

Our $\mathcal{NC}$ algorithms for LFF parsing work for static dictionaries with a logarithmic bound on the length of dictionary entries and fixed-length dictionary

references. $\mathcal{NC}$ algorithms for LFF parsing with arbitrary length entries and references are a natural extension to investigate.

Since dynamic dictionary compression of the Lempel and Ziv variety is P-complete, dynamic methods differing from this approach are of interest. List compression, where a dictionary of words is maintained using a self-organizing heuristic, seems amenable to parallelization. Also, parallel parsing strategies other than those used for sequential compression may lead to improved parallel systems.

References

[AILSV88] Apostolico, A., Iliopoulos, C., Landau, G. M., Schieber, B., and Vishkin, U. Parallel construction of a suffix tree with applications. *Algorithmica 3*, 3 (1988), 347–365.

[AM91] Anderson, R. J. and Miller, G. L. Deterministic parallel list ranking. *Algorithmica 6* (1991), 859–868.

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

[BB87] Bertossi, A. A. and Bonuccelli, M. A. Some parallel algorithms on interval graphs. *Discrete Appl. Math. 16* (1987), 101–111.

[CL82] Cooper, D. and Lynch, M. F. Text compression using variable- to fixed-length encodings. *J. Amer. Society for Information Science* (Jan., 1982), 18–31.

[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 (1991).

[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, 1992, pp. 52–61.

[DS93] De Agostino, S. and Storer, J. A. Parallel algorithms for optimal compression using dictionaries with the prefix property. Manuscript (1993).

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

[G80] Golumbic, M. C. *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, 1980.

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

[J92] JaJa, J. *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992.

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

[N91] Naor, M. String matching with preprocessing of text and pattern. In *Proceedings ICALP, LNCS 510*, Springer-Verlag, 1991, pp. 739–750.

[OSZ92] Olariu, S., Schwing, J. L., and Zhang, J. Optimal parallel algorithms for problems modeled by a family of intervals. *IEEE Trans. Parallel and Distributed Systems 3*, 3 (May, 1992), 364–374.

[SH73] Schuegraf, E. J. and Heaps, H. S. Selection of equifrequent word fragments for information retrieval. *Inform. Stor. Retr. 9* (1973), 697–711.

[SH92] 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, 1992, pp. 162–171.

[SH93] Stauffer, L. M. and Hirschberg, D. S. Parallel text compression. Technical Report 91-44, Revised. Info. and Comp. Sci. Department, University of California, Irvine (1993).

[S88] Storer, J. A. *Data Compression Methods and Theory*, Computer Science Press, Rockville, MD, 1988.

[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.

[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.

[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.