Parallel Algorithms for Data Compression

M. E. GONZALEZ SMITH AND J. A. STORER

Brandeis University, Waltham, Massachusetts

Abstract. Parallel algorithms for data compression by textual substitution that are suitable for VLSI implementation are studied. Both "static" and "dynamic" dictionary schemes are considered.

Categories and Subject Descriptors: B.7.0 [**Integrated Circuits**]: General; E.4 [**Coding and Information Theory**]: *data compaction and compression*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical algorithms and problems—*computations on discrete structures, pattern matching, routing and layout*

General Terms: Algorithm

Additional Key Words and Phrases: Data compression, textual substitution, VLSI, parallel Algorithm, distributed processors.

1. *Introduction*

In distributed computing environments, it can often be the case that communication channels are relatively slow. The ability to put large amounts of processing power on a single chip promises to make sophisticated data compression algorithms truly practical. A data encoding/decoding chip can be placed at the ends of every communication channel, with no computation overhead incurred by the communicating processes.

We shall compress data by textual substitution, a technique studied by many authors (e.g., [5] and [11]). We implement our algorithms in VLSI with *systolic arrays*, a model of parallel computation studied by many authors (see [3], Section 8.3). The idea is to lay out a regular pattern of processing elements that have a simple interconnection structure; ideally, each element is connected only to adjacent elements. In contrast to this model of computation, one could simply implement a serial data compression algorithm using, for example, a microprocessor chip with some memory. However, a practical advantage of the systolic array implementation is speed, making the resulting data compression chip appropriate for a wider range of applications. In addition, the throughput of many of the systolic structures we describe are independent of their size; this is important from both a practical and theoretical standpoint.

Sections 2–4 present systolic algorithms for compressing data using the *static dictionary model*; where data is compressed by replacing substrings of text by a

Authors' present addresses: M. E. Gonzalez Smith, Department of Computer Science, Cornell University, Ithaca, NY 14850; J. A. Storer, Computer Science Department, Brandeis University, Waltham, MA 02254.

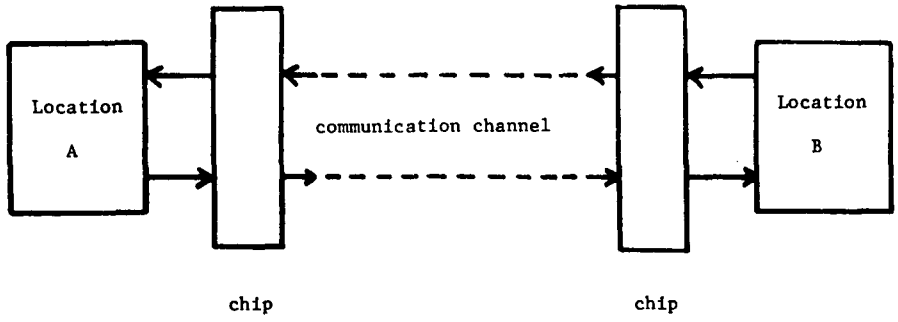


FIGURE 1

pointer to a dictionary. Sections 5–7 present systolic algorithms for the *sliding dictionary model*; where text is compressed by replacing substrings of text to an earlier occurrence in the text. Both of these models have been studied extensively in the literature and have relative merits depending on the application [11].

2. *Static Dictionary Model, Basic Definitions*

This section outlines a basic systolic structure for data compression along a communication channel, using the static dictionary model. A dictionary of strings is constructed in some fashion; presumably by either sampling the source data or by employing some statistical knowledge of the data (e.g., a dictionary of common English words[1]). We do not address the dictionary selection problem in this paper; we simply assume that a dictionary is available. Given a dictionary, data is compressed by replacing substrings of the data by pointers to matching strings in the dictionary. For example, if the dictionary consists of the two strings *ab* and *bab*, the string *ababbab* could be represented as $11b1$, $a22$, or $a2b1$, where 1 denotes a pointer to *ab* and 2 a pointer to *bab*. Selection of the shortest compressed form is the subject of Section 4. Decoding is always unique and consists of simply replacing a pointer by its target. We allow dictionary entries themselves to contain pointers to other entries (cycles are not allowed) and so decoding a pointer may involve several dictionary lookups. We refer to the string represented by a pointer, after all pointers have been expanded, as the *expanded target* of that pointer. Pointers in dictionary entries can reduce the space for the dictionary, but the major reason for such pointers is that they allow for the representation of strings longer than the maximum length of a dictionary element. This may or may not be of practical importance, but the algorithm to be presented includes this generality with no loss of performance.

Figure 1 depicts the basic situation. The two dashed lines indicate a communication channel (with one line going in each direction) and at each end of the channel are identical encoding/decoding modules. At the start of a "session," a dictionary is constructed for the body of data in question and loaded into the modules. From this point on, data is (transparently) compressed, transmitted, and decoded.

[1] Kucera and Francis [2] present statistics for the English language, including word frequencies.

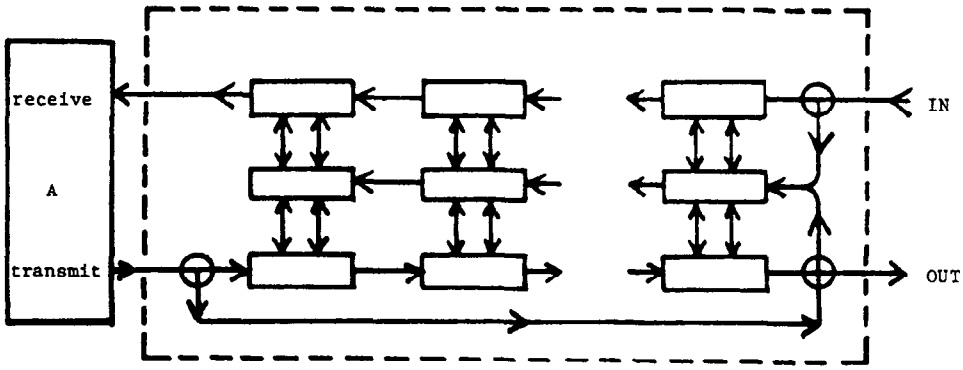


FIGURE 2

Figure 2 shows the structure of the encoding/decoding array. It is a pipe with 3 processing elements for each string of the dictionary. Referring to Figure 1, Figure 2 is shown as connected to location A on the left and the communication line on the right. For the remainder of this paper, we shall assume that the encoding/decoding array is oriented this way, and we use the terms left and right as they pertain to Figure 2. The dictionary strings themselves are stored in the middle row, one dictionary string per element. Pointers within dictionary elements will always point to (a suffix of) a dictionary element to the left. Data to be encoded enters from the left, is piped along the bottom row, and encoded as it progresses. Data to be decoded enters from the right, is piped along the top row, and decoded as it progresses. The dictionary may be loaded by either location A or location B, sending a control sequence to divert data from the normal path into the dictionary. The circle next to the transmitter for location A indicates a switch to divert data via the bypass line, to the dictionary, and to the output (to the dictionary for location B).

One advantage of the array structure of Figure 2 is that any number of these can be combined to produce a larger one.[2] Another advantage, both practical and theoretical, concerns maximum edge (wire) lengths. There has been much debate as to the right model of time for a VLSI signal to propagate along a wire [1] and some authors have argued that as much as $O(n^2)$ should be charged to an edge of length n. In Figure 2, the only edge whose size depends on the length of the pipe is the bypass line. If Figure 2, less the bypass line, is "folded" as depicted in Figure 3 (an $O(n)$-node pipe goes to an $O(\sqrt{n})$ by $O(\sqrt{n})$ rectangular region), then the entrance and exit of the structure are next to each other, and an $O(1)$ bypass line can be placed. Thus, from the standpoint of VLSI layout, the array structure depicted by Figure 2 is quite desirable:

—There are three basic elements from which the entire array is constructed.
—Each array element is connected only to its nearest neighbors in the pipe (except the bypass line).
—The structure has a linear area layout with all edges (including the bypass line) having length $O(1)$.
—The layout strategy is independent of the number of chips used.

[2] That is, they can be "stuck" together end to end. A technical point; the bypass "switch" would have to be modified to make the bypass line go from the lead chip to the end chip.

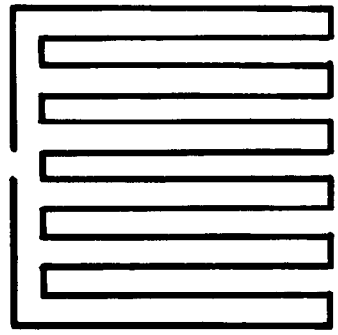


FIGURE 3

We use the following notation:

b = Number of bits per character.
ρ = Minimum size of a pointer, in units of characters ($\rho \geq 0$).
p = Maximum size of a pointer, in units of characters ($p \geq \rho$).
d = Maximum length of a dictionary element, in units of characters.

By using characters as the units for ρ and p, we are restricting the number of bits in a pointer to be an integer multiple of b. None of what we do requires this, but it simplifies our discussion. In particular, a pointer can be viewed as a sequence of *pseudocharacters*, each b bits long. We define an *item* to be either a character or a pseudocharacter. Given this definition, a compressed form is simply a sequence of items. For a string S of items $|S|$ denotes its length (in units of b). If q is a pointer to a target t, it is always assumed that $|q| < |t|$.

It will be useful to allow pointers to indicate any suffix of a dictionary element. This can be done by letting a pointer with integer value i denote the string consisting of the ith character of the dictionary through the end of the dictionary element containing the ith character. We assume that pointers to suffixes of the same dictionary element have the same length. This assumption is mainly for simplicity, but also has technical importance in Section 4.

Example 1. Consider a dictionary of $2^{12} = 4096$ elements of at least 8 characters ($d = 8$). The input to the encoder is a sequence of 8-bit character codes ($b = 8$). The output is a sequence of 8- or 16-bit pointers ($p = 2$). A pointer is either a 1 followed by a 7-bit integer pointing to elements 1 through 128 of the dictionary or a 0 followed by a 15-bit integer i pointing to the ith character of the dictionary (the dictionary can have at most 2^{15} characters); that is, short pointers indicate one of the first 128 elements of the dictionary and long pointers indicate any suffix of any element of the dictionary. Every character of the source is itself an element of the dictionary. For English text, for example, the characters a through z, along with common strings such as "the," might be in the first 128 dictionary elements whereas less common characters such as # and less common words such as "aardvark" might be in the higher locations.

3. *Static Dictionary Model, Decoding*

This section gives a systolic algorithm for decoding using the static dictionary model; that is, the algorithm to be used by each processing element along the top

row in Figure 2. For a given one of these decoding processing elements, we refer to the corresponding dictionary element as *dictelement*. The representation of a pointer is not important; we assume only that a pointer uniquely specifies a suffix of a dictionary element (which itself may contain pointers). In addition, there must be some way to distinguish the item starting a pointer from a character and, given the start of a pointer in a stream of items, it must be possible to determine where it ends. It could be, as in Example 1, that the input stream can contain only pointers (and distinguishing the start of a pointer from a character is a moot question), and in this case, there is a minor technical problem of how to distinguish decoded characters from pointers. This is easily handled by adding an extra bit for each input item of b bits, initially 0. When characters are substituted for pointers, they are coded as 1 followed by the character; the 1's can be stripped off as the characters leave the decoding pipe. In any case, we henceforth assume that characters (whether part of the input stream or produced while decoding) are always distinguishable from the first item of a pointer, and we make no more mention of these details.

Algorithm 1 (Figure 4) is the basic decoding algorithm for an element. Each decoding element has a buffer with $p + 1$ *cells*, each capable of holding an item. The buffer behaves like a queue, where items enter from the right and leave to the left. We assume two global alternating time intervals (e.g., a two-phase nonoverlapping clock) called *phase* 1 and *phase* 2. During *phase* 1, the algorithm does the following: First, the contents of the buffer are shifted as far to the left as possible (so that all empty locations are at the right of the buffer). The second step is to check if *count* = 0 and if there is a complete pointer starting at the left of the buffer. The (local) variable *count* is used to keep track of how much of a pointer has been transmitted. The test *count* = 0 ensures that, if the leftmost item in the buffer is part of a pointer, then it is the beginning of the pointer.[3] Given that *count* = 0 and a complete pointer is leftmost in the buffer, there are two possibilities, either the pointer points to a suffix of *dictelement* or it is some other pointer. The variable *count* is used differently depending on which of these cases holds. In the first case, *count* is set to the length of the target of the pointer, the pointer is deleted, and *count* will be used to count down as the target is transmitted one character at a time. In the second case, *count* is set to minus the length of this pointer and will be used to count up as this pointer is passed down the pipe one item at a time. Thus, the first two cases can be distinguished by testing the sign of *count*.[4] The third and final step of *phase* 1 is to prevent buffer overflow. The problem is that, as we transmit a target of a pointer that was deleted from the buffer, more characters could be coming into the buffer (which has only $p + 1$ cells) from the right. To take care of this, the decoding element has a (local) variable *lock* that has the purpose of blocking new characters from entering on the right when it is set to 1. The decoding element is capable of reading the value of *lock* for the element ahead of it (to the left) in the pipe; we refer to this value as *nextlock*. The last step of *phase* 1 sets *lock* = 1 only if the buffer contains p items and either *nextlock* = 1 or *count* > 0 (*count* > 0 means that we will be transmitting characters from *dictelement*, not from the buffer).

[3] That is, we want to determine the cell containing the end of the pointer. This may not be possible starting from a cell in the middle of the pointer.

[4] We have done this mainly for convenience. Alternately, *count* could be positive in the second case and a bit could be set to distinguish between these two cases.

```
during phase 1 do
  begin
  shift items in buffer to the left (so that all empty cells are at the right)
  if (count = 0) & (leftmost item starts a complete pointer q) then begin
    if q points to a suffix of dictelement then begin
      count = ⟨length of target of q⟩
      remove q from buffer
      end
    else count = −⟨length of q⟩
    end
  if (buffer has ≥ p items) & (nextlock = 1 or count > 0) then lock = 1 else
  lock = 0
  end
during phase 2 do if (buffer not empty) & (nextlock = 0) then
  begin
  if (count > 0) then begin
    count = count − 1
    transmit the d − count item of dictelement
    end
  else if (count < 0) then begin
    count = count + 1
    transmit leftmost item
    end
  else if (leftmost item a character) then transmit leftmost item
  end
```

Notes:
(1) The buffer has $p + 1$ cells, a cell can contain one item.
(2) *dictelement* refers to the current dictionary element.
(3) *count* is a counter for the range $-d$ to d.
(4) *lock* is a single bit, initially 0.
(5) *nextlock* is the value of *lock* for the succeeding element in the pipe.

FIG. 4. Algorithm 1. Static dictionary decoding algorithm.

Phase 2 is when characters are transmitted from one element to another along the decoding pipe. An element transmits at most one item during *phase* 2, and only if *nextlock* = 0. Given that *nextlock* = 0, there are three cases: First, if *count* > 0, then we are partway through transmitting a target from *dictelement*, so we decrement *count* and transmit the appropriate character of *dictelement*. In the second case, *count* < 0, so we must be in the process of transmitting the items comprising a pointer (but not a pointer to a suffix of *dictelement*), and we increment *count* and transmit the leftmost item of the buffer. The only two possibilities left are that the leftmost cell contains the start of a pointer (but not all the items comprising it are in the buffer yet) or the leftmost cell contains a character. *Phase* 2 transmits the leftmost item only if the second possibility holds; it is always possible to distinguish these two possibilities since an item that is the start of a pointer is always distinguishable from a character (as discussed at the start of this section).

A difficulty with Algorithm 1 concerns the locking to prevent buffer overflow. The locking works properly since, when a buffer becomes p-full, at most one more character is received before the element preceding it in the decoding pipe detects that *lock* = 1. However, in theory, *lock* = 1 could propagate arbitrary distances down the pipe (propagating at most a distance i after i cycles of *phase* 1 and *phase* 2), causing a problem if it should ever reach the entrance to the pipe. Such a global propagation is clearly unavoidable if no assumptions are made concerning the data

rate into the decoding circuit versus the amount of decompression that takes place. In practice, however, average compression is bounded above by a constant factor. For example, for English text, Shannon [8] argues that the information content is approximately 1 bit per character, which for 8-bit character codes, would bound average compression obtainable for English text by a factor of 8. For the static dictionary model, we henceforth assume that there is some constant that bounds the average amount of compression for any given string. Given this assumption, it must be that for any $0 < h < 1$, there is a constant l such that, on the average, less than a fraction h of a given string can be compressed with a pointer with an expanded target longer than l. Hence, for sufficiently large l, there is no significant loss of average compression to let:

l = Maximum length of an expanded target.

Given l, the maximum compression ratio is l/ρ. We also define the two parameters:

r = Rate of communication channel; that is, the minimum time between successive characters of input to the decoding circuit.
s = Speed of decoding circuit; that is, time for one cycle of *pulse* 1 and *pulse* 2.

Given the parameters l/ρ, r, and s, we make the basic assumption that the data rate of the communication channel is sufficiently slower than the speed of the chip so that the maximum compression ratio is less than the ratio of r to s:

$$\frac{l}{\rho} < \frac{r}{s}.$$

The following theorem shows that this assumption guarantees that the entrance to the pipe can never be held up. The parameter $\rho r/s$ can be given to a dictionary construction algorithm as a maximum value for l.

Definition 1. A compressed form S is *blocking* if there exists a dictionary (and corresponding circuit)[5] such that, when S is input to the decoding pipe, at some point, the first element of the pipe sets *lock* = 1 at a time when an item of S is arriving to the pipe.

THEOREM 1. *Given that $l/\rho < r/s$, no compressed form can be blocking.*

PROOF. Before proceeding, we observe that, for the purposes of this proof, Algorithm 1 can be modified by adding the condition

(buffer has $\geq p$ items)

to the first **if** of *phase* 1 and to the last **if** of *phase* 2. This does not change the correctness of the algorithm[6] but does ensure that every item effectively moves through the pipe one buffer cell at a time.[7] Also, it is clear that if S is blocking using Algorithm 1, then it is also blocking with Algorithm 1 modified as described above.

[5] That is, for some n, there is some dictionary of n elements that can be assigned to a pipe structure (of the type of Figure 2) of length n, subject to the constraints b, p, and d,

[6] There is a minor problem of how to flush out the pipe when the input ends or is suspended. This is irrelevant here, since we assume that the input is blocking.

[7] Actually, even with this modification, an item can skip over the $p + 1$th cell of the buffer, but we will be concerned only with the leftmost p cells in a buffer.

Now, let us assume the contrary to this theorem; that is, assume that S is a shortest blocking compressed form. It must be that S starts with a pointer, since a leading character could never hold up the pipe (and we could delete it to get a shorter blocking compressed form), and so

$$S = qT$$

where q is a pointer and T a compressed form.

For the remainder of this proof we measure time in units of s.[8] Hence we have

$$r > \frac{l}{\rho}.$$

This means that an item of S enters the decoding pipe at most once every l/ρ time units. We write q as

$$q = q_1, \ldots, q_k$$

where the q_i's are items and $\rho \leq k \leq p$. Now imagine a "tag" attached to q_k; we call it *tag* 1. The purpose of *tag* 1 is to mark the rightmost item generated by q. Although *tag* 1 is initially attached to q_k, when q is replaced by its target, *tag* 1 moves to the rightmost symbol of the target; and if the rightmost item of the target is part of a pointer, then *tag* 1 is moved again when this pointer is expanded, and so on.

A key observation is that there are at most $l - |q|$ time units that *tag* 1 does not forward one buffer cell. To see this, note that *tag* 1 moves forward every time unit except when a pointer is being replaced, and each replacement of a pointer x by a target y adds an effective delay of $|y| - |x|$. Hence, since the expanded target of q has length at most l, the delay is at most $l - |q|$.

We now consider the first ρ items of T, which we refer to as $i_1, \ldots, i_\rho$. Since $l < r\rho, i_\rho$ enters the pipe at least l time units behind q_k. We attach *tag* 2 to q_k as it enters the pipe. *Tag* 2 moves like *tag* 1, but marks the leftmost item generated by q_k; therefore, whenever a pointer is replaced, *tag* 2 goes to the left of the target.

The second key observation is that *tag* 2 can never catch up to *tag* 1. This is because *tag* 1 and *tag* 2 both move forward one buffer cell each time unit with two exceptions. The first is when *tag* 1 is held up due to pointer replacement, which accounts for a total delay of at most $l - |q|$. The second occurs only if i_ρ meets $i_1, \ldots, i_{\rho-1}$ to form a pointer, in which case *tag* 2 jumps forward a distance of $|\rho| \leq |q|$. Since q_k enters the pipe l time units ahead of i_ρ, the observation follows.

Thus, q can have no effect on how T travels through the pipe, and so T is a shorter blocking compressed form than S, a contradiction.

4. *Static Dictionary Model, Encoding*

Given a dictionary, a particular string may have many different compressed forms. Wagner [12] gives a simple dynamic programming algorithm to find a minimal length compressed form, when the dictionary is given. This algorithm does not assume variable length pointers, the ability to point to a suffix, or pointers within dictionary elements, but it is easily generalized to accommodate these features. For D, a given dictionary of strings, and $s = s_1, \ldots, s_m$ a string to be compressed, let:

$$q_{i,j} = \begin{cases} \textbf{if } (i = j) \textbf{ then } s_i \\ \textbf{else if (}\text{exists a pointer with expanded target } s_i, \ldots, s_j\textbf{) then } \text{the shortest one} \\ \textbf{else } \infty \end{cases}$$

[8] We assume each "tick" of the clock coincides exactly with the start of *pulse* 1.

FIG. 5. Algorithm 2. Wagner's Algorithm, slightly modified.

```
C(m + 1) = ⟨the empty string⟩
for i = m to 1 by −1 do
  C(i) = MIN{q_{i,j}C(j + 1): i ≤ j ≤ m}
```

FIG. 6. Algorithm 3. Static dictionary greedy encoding algorithm.

```
i = 1
while i ≤ m do
  begin
  let j be such that j − i − |q_{i,j}| is maximum
  replace S_i, . . . , S_j by q_{i,j}
  i = j + 1
  end
```

The representation and lengths of pointers are unimportant, but must be specified before $q_{i,j}$ can be defined. For a set of strings S, MIN(S) denotes a shortest one.

For each $1 \leq i \leq m$, Algorithm 2 (Figure 5) computes $C(i)$, a shortest compressed form of $s_i, \ldots, s_m$ (using dictionary D). The problem with Algorithm 2 is that, not only must the entire source string be available, but it can also involve global data flow. This section gives a simple "greedy" algorithm (Algorithm 3), and then compares its performance to that of Algorithm 2. From this greedy algorithm, a parallel algorithm suitable for our array structure is derived.

As discussed earlier, we have allowed dictionary elements to contain pointers to address certain practical situations. For the moment, we assume that dictionary elements do not contain pointers; we address this subject later in this section.

Algorithm 3 (Figure 6) is the greedy encoding algorithm. It simply proceeds through the string a character at a time, and at every opportunity, replaces a string by a pointer (choosing the best possible replacement). For a dictionary D and a string s, let $M(D, s)$ denote a compressed form obtained by Algorithm 2 (M for "minimum") and $G(D, s)$ denote the compressed form obtained by Algorithm 3 (G for "greedy").

THEOREM 2. *Let D be a dictionary and s a string. Under the assumption that no dictionary element contains a pointer, then*

$$\frac{|M(D, s)|}{|G(D, s)|} \geq \frac{p}{p + \rho - 1}. \tag{1}$$

Furthermore, even if D is restricted and has only 2 elements and s is written over a 2-symbol alphabet, this bound may be obtained arbitrarily closely. That is, for any real $h > 0$, there are infinitely many strings such that

$$\frac{|M(D, s)|}{|G(D, s)|} < \frac{p}{p + \rho - 1} + h. \tag{2}$$

PROOF OF PART (2). We exhibit infinitely many strings that achieve the bound. Let

$$u = a(b^p), \qquad v = (b^{p+p-1})a, \qquad D = \{u, v\}$$

where any pointer to u has length p and any pointer to v has length ρ. Now, for any $i > 0$, let

$$s(i) = a(v^i).$$

For example, for $\rho = p = 2$, $s(3) = abbbabbbabbba$. If q_1 denotes a pointer to u, q_2 a pointer to v, then $M(D, s(i)) = a(q_2^i)$ has length $1 + i\rho$, whereas $G(D, s(i)) = (q_1 b^{\rho-1})^i a$ has length $1 + i(p + \rho - 1)$, from which the bound follows.

PROOF OF PART (1). The proof of this bound can be viewed as a generalization of a portion of a proof appearing in Storer and Szymanski [11]; although the proof here is more technical, the structure is the same.

Without loss of generality, we can assume that, in any minimal length compressed form, any substring that is represented by a pointer to an earlier occurrence is as long as possible; that is, if $s_m, \ldots, s_n$ is represented by a pointer, then $s_m, \ldots, s_{n+1}$ is not a suffix of a dictionary element. Otherwise, we could obtain an equivalent compressed form of the same or shorter length by changing the pointer to represent $s_m, \ldots, s_{n+1}$ and then either deleting a character (if the pointer was originally followed by a character) or changing the following pointer[9] (if the pointer was originally followed by another pointer).

To reduce notation, let $t = M(D, s)$ and $u = G(D, s)$. Form the finest partition of t and u into segments $t = t_1, \ldots, t_m$ and $u = u_1, \ldots, u_m$ such that, for $1 \leq j \leq m$, t_j and u_j represent the same substring of s. In order to establish the bound claimed, it is sufficient to show that

$$\frac{\rho}{p + \rho - 1} \leq \frac{|t_j|}{|u_j|} \leq 1 \qquad \text{for} \quad j > 1.$$

By definition of the greedy algorithm, it is impossible for some t_j to begin with a pointer while u_j begins with a character. Hence, it must be that t_j begins with a character and u_j begins with a pointer; otherwise, both t_j and u_j must consist of a single character or pointer (the pointers must be of the same length, due to the greedy algorithm), and so $|t_j| = |u_j|$. Thus, we can write

$$t_j = x_1 q_1 x_2 q_2 \cdots x_n q_n x_{n+1},$$
$$u_j = r_1 y_1 r_2 y_2 \cdots r_m y_m,$$

where each of the q_i's and r_i's is a pointer, x_1 is a string of one or more characters, and each of the remaining x_i's and y_i's is a string of zero or more characters.

Any substring of s that is represented by characters (as opposed to pointers) in either t_j or u_j must be represented by a pointer in the other, since t_j and u_j have been defined in terms of a finest possible partition of t and u. Figure 7 suggests the structure of that portion of s represented by t_j and u_j.

Notice that for each i, $1 \leq i \leq n$, q_i represents at least the last character represented by r_i, all of y_1, and at least the first character represented by r_{i+1}. Also, for $2 \leq i \leq m$, r_i represents at least the last character represented by q_{i-1}, all of x_i, and at least the first character represented by q_i. To verify the above facts, depicted in Figure 4, it is sufficient to observe that, except at the end, if q_i starts within r_i, then q_i must go beyond the end of r_i since if q_i ended earlier, then q_i wouldn't be as long as possible (as we assumed at the start of the proof), and if q_i ended at the same place, we wouldn't have the finest possible partition. Similarly, if r_i starts within a

[9] Remember, it will always be possible to change the following pointer because any suffix of a dictionary element may be a pointer target.

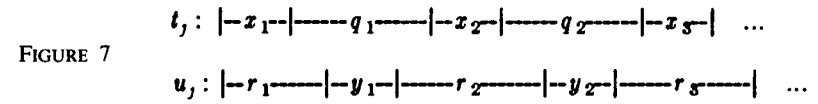


FIGURE 7

q_i, then r_i must go beyond the end of q_i since to end earlier would imply a violation of the greedy rule and to end in the same place would violate the finest partition. Thus, it must be that either $m = n$ or $m = n + 1$. In addition, for $1 \leq i < m$, $|y_i| \leq |q_i| - 1$, or else the greedy algorithm would have used a pointer instead of y_i (pointing to a suffix of the target q_i).

Now, one of the following three cases must hold:

Case 1. $n = m$. Since $|x_1| \geq 1$, $|t_j| \geq 1 + \sum_{i=1}^{n} |q_i|$. Hence, since for $1 \leq i < n$, $|y_i| \leq |q_i| - 1$, $|y_n| \leq |q_n|$, and for $1 \leq i \leq n$, $|r_i| \leq p$, we have

$$\frac{|t_j|}{|u_j|} \geq \frac{1 + \sum_{i=1}^{n} |q_i|}{np + \sum_{i=1}^{n} |y_i|}$$

$$\geq \frac{1 + \sum_{i=1}^{n} |q_i|}{1 + n(p-1) + \sum_{i=1}^{n} |q_i|}$$

$$\geq \frac{1 + n\rho}{1 + n(p + \rho - 1)}$$

$$\geq \frac{\rho}{p + \rho - 1}.$$

The above inequalities rest on the fact that, if $x \leq y$, then for any $a > 0$, $(a + x)/(a + y) \geq x/y$.

Case 2. $m = n + 1$ and x_{n+1} is the empty string. Thus, both t_j and u_j end in a pointer. It must be that y_{m-1} is also empty, or else the greedy algorithm would have replaced the string represented by $y_{m-1}r_m$ by a single pointer (to a suffix of the target of q_n). In addition, it must be that $|r_m| \leq |q_m|$, since the target of r_m is a suffix of the target of q_m. Hence

$$\frac{|t_j|}{|u_j|} \geq \frac{1 + \sum_{i=1}^{n} |q_i|}{np + |r_m| + \sum_{i=1}^{n-1} y_i}$$

$$\geq \frac{1 + \sum_{i=1}^{n} |q_i|}{1 + n(p-1) + \sum_{i=1}^{n} |q_i|}$$

and we may now continue as in Case 1.

Case 3. $m = n + 1$ and that x_{n+1} is not the empty string. By our definition of t and u in terms of a finest possible partition, it must be the case that y_{n+1} is the empty string. Also, the string represented by q_n extends at least $(|r_{n+1}| + 1) - |x_{n+1}|$ characters past y_n. Hence, it must be that $(|y_n| + |r_{n+1}| + 1 - |x_{n+1}|) \leq |q_n|$; otherwise, the presence of q_n implies that the greedy algorithm must make y_n the empty string and place a pointer directly after r_n (to a suffix of the target of q_n). This simplifies to $|y_n| \leq (|q_n| - |r_{n+1}| +$

$|x_{n+1}| - 1$). Thus we have

$$\begin{aligned}\frac{|t_j|}{|u_j|} &\geq \frac{1 + |x_{n+1}| + \sum_{i=1}^{n} |q_i|}{np + |r_{n+1}| + |y_n| + \sum_{i=1}^{n-1} y_i} \\ &\geq \frac{1 + |x_{n+1}| + \sum_{i=1}^{n} |q_i|}{n(p-1) + |x_{n+1}| + \sum_{i=1}^{n} |q_i|} \\ &\geq \frac{1 + |x_{n+1}| + n\rho}{|x_{n+1}| + n(p + \rho - 1)} \\ &\geq \frac{\rho}{p + \rho - 1}.\end{aligned}$$

In all of the above cases we have shown $|t_j|/|u_j| \geq \rho/(p + \rho - 1)$.

The above theorem says that the greedy algorithm and an optimal algorithm differ by at most a factor of $\rho/(p + \rho - 1)$, which implies that for $p = 1$, the greedy algorithm is optimal. Unfortunately, the second half of the theorem says that this bound can be obtained, even for large values of p (and small values of ρ). However, in practice, p is likely to be small. Furthermore, we conjecture that in practice, the worst case bound would not be obtained. In any case, from a theoretical standpoint, we view the above theorem to mean that the greedy algorithm is a reasonable approach (i.e., for small values of p, it differs from the optimal by at most a small constant factor).

Example 2. For this example, let us assume that the source consists only of the characters a through z and punctuation, and that pointers are represented by two-digit numbers. This makes compressed forms "readable," but here, we can have only 100 different pointers that are (exactly) two characters in length, whereas Example 1 showed that with 8-bit character codes, one- and two-character pointers could be represented that point to any of 2^{15} different suffixes of 4,096 dictionary entries of maximum length 8 characters. Suppose the dictionary consists of entries of maximum length 8 characters as follows:

```
00-07: "program"
08-15: "extreme"
16-21: "large"
22-26: "some"
27-30: "are"
31-38: "ALGOL00"     ("ALGOL program")
39-44: "08ly16"      ("extremely large")
45-48: "2739"        ("are extremely large")
```

Then the sentence, "Some ALGOL programs are extremely large.", would be transformed as follows:

```
Some ALGOL programs are extremely large.
Some ALGOL00s are extremely large.
S24 ALGOL00s are extremely large.
S24 ALGOL00s are08ly large.
S24 ALGOL00s are08ly16.
S2431s are08ly16.
S2431s2708ly16.
S2431s2739.
S2431s45.
```

```
during phase 1 do
  begin
  if (buffer full) & (a prefix of buffer matches a suffix of dictelement) then
    begin
    Let s be the longest such suffix and q a pointer to s and if |q| < |s|, replace suffix s of
      the buffer by q
    end
  shift items in buffer to the right (so that all empty cells are at the left)
  end
during phase 2 do
  if (buffer full) then transmit rightmost item
```

Notes:
(1) The buffer has p cells; a cell can contain one item.
(2) *dictelement* refers to the current dictionary element.

FIG. 8. Algorithm 4. Static dictionary encoding algorithm.

Even with the greedy algorithm, there remains a problem for a systolic array implementation: searching all dictionary elements for the longest match requires global communication. The systolic encoding algorithm to be presented assumes the dictionary is organized in order of shortest to longest expanded targets, and encoding proceeds from left to right, taking a match whenever one is found. If a suffix of a dictionary element is never a prefix of another, then this is, effectively, the greedy algorithm. However, when this is not the case, we do not know how close this comes to the true greedy algorithm. Clearly, the answer depends somewhat on the dictionary selection algorithm; and so we have shifted some of the work of encoding on to the one-time task of dictionary selection.

Algorithm 4 (Figure 8) is the encoding algorithm. During *phase* 1, if the largest match between a suffix of the buffer and a suffix of *dictelement* is bigger than a pointer to the dictionary suffix, then it is replaced by the pointer. After this, the characters in the buffer are shifted to the right as far as possible (so that all empty space is to the left in the buffer). During *phase* 2, the rightmost item is transmitted only if the buffer is full; this ensures that the largest match possible is found. A few details have been left out; in particular, how to "flush out" the array when the input stream ends or slows down.

Like the decoding algorithm (Algorithm 1), a nice feature of Algorithm 4 is that the ability to handle pointers within dictionary elements comes for "free"; Algorithm 4 works as is, provided the dictionary is ordered properly. Dictionary elements without pointers can be ordered as before: larger to smaller going from left to right. The elements with pointers, however, must go the other way; that is, pointers must point to the left in the pipe. Whether pointers within dictionary elements give significant benefits in practice depends on both the source and the size of the dictionary elements, as well as the dictionary selection algorithm. We leave these issues as areas for future research.

5. *Sliding Dictionary Model, Basic Definitions*

In the sliding dictionary model, if S is a string, then a compressed form of S is a string of items T such that pointers in T indicate substrings of S, and S can be obtained by expanding the pointers of T from left to right. It will always be assumed that the pointers in T are such that the string they represent in S is strictly to the

right of their target; that is, pointers can be thought of as pointing to the left.[10] Given this, decoding from left to right is always well defined, since the target of a pointer is always a substring of the portion of the string already decoded. For encoding, Ziv and Lempel [13] present a simple "greedy" algorithm[11] that proceeds from left to right in the source string; at any given point, it is determined whether the longest string that starts at this point and that also occurs earlier in the source text is longer than the size of a pointer. If so, it is replaced by a pointer. This is the "pure" version of their greedy algorithm. Since the source string could be arbitrarily long, they consider for the above algorithm a modification that at each point looks for the longest string that starts at this point and is the substring of the last n characters, for some n. We refer to the last n characters as the *sliding dictionary* and use the following notation:

p = Pointer size, in units of characters ($p \geq 1$).
n = Length of the sliding dictionary, in units of characters.

As in the previous sections, the representation of a pointer is not important, we only assume that it is possible to distinguish pointers from characters. In order to simplify our presentation, we do not consider variable length pointers as in the previous sections; all pointers have exactly length p. We use the notation (x, y) to denote a pointer whose target has length y and where the first character of the target is the xth character of the source string.

Example 3. Suppose $p = 1$, $n = 3$, and the source string was

ababbababa

Then the greedy algorithm would produce the compressed form:

$ab(1, 2)(2, 3)(6, 2)a$

Note that the *aba* at the end of the source could not be replaced by the pointer (1, 3) because $n = 3$. Also note the meaning of a pointer; for example, the pointer (1, 2) refers to the first and second characters of the source string, not the compressed form (although in this case, they are the same).

Although Ziv and Lempel [13] show that the greedy algorithm is asymptotically optimal for ergodic sources as n and the length of the source string become arbitrarily large, Storer and Szymanski [11] show that the ratio of an optimal length compressed form to one produced by the greedy algorithm can be as bad as $(p + 1)/2p$. As with the greedy algorithm presented in Section 4, we conjecture that this worst case performance would not be achieved in practice, and the work of Seery and Ziv [6, 7] seems to support this conjecture, and that practical values of n can achieve good performance.

6. *Sliding Dictionary Model, Encoding*

This section develops several systolic implementations of the greedy encoding algorithm for the sliding dictionary model. To make some of our examples readable, we now assume (unlike the preceding sections) that data to be encoded enters from the right and leaves to the left. We store the last n characters processed (i.e., the sliding dictionary) in a systolic array (one character per processing element). The

[10] More general models than this, which we do not address here, are studied in Storer and Szymanski [11].

[11] Rodeh, Pratt, and Even [5] present a linear time implementation.

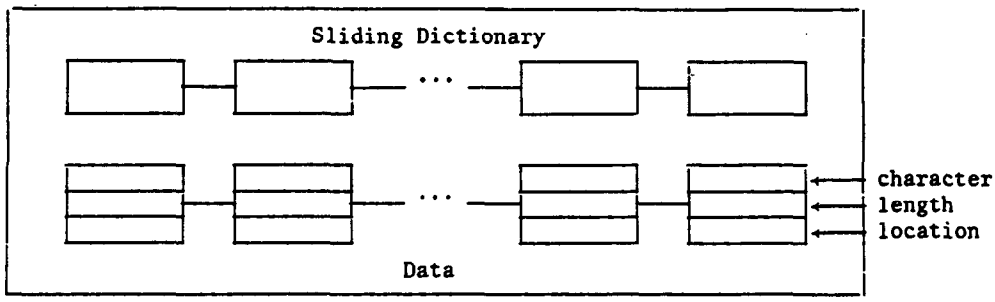


FIGURE 9

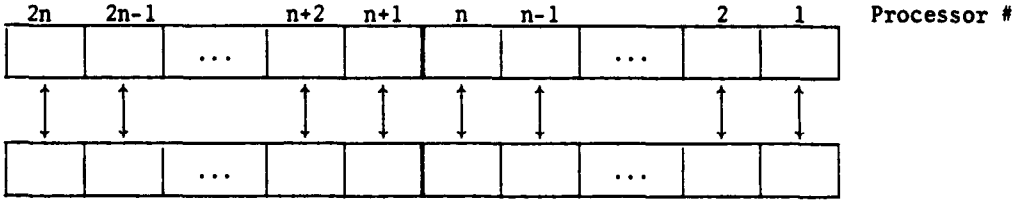


FIGURE 10

dictionary is updated by sliding old characters left in order to bring in new ones. We can think of this dictionary as a window of width n that is sliding through the input stream. As new characters are read in, we compare them with each of the elements of the dictionary. In other words, we compare them with the n characters preceding them. To each character, we attach a pair of values [*location, length*]. These values indicate the location and length of the longest substring of the dictionary that matches a prefix of the portion of the input that starts with each character (when the character gets shifted from element to element, its [*location, length*] values get shifted too). The [*location, length*] information is updated any time a longer match is found. After a character has been compared to the n elements, its [*location, length*] information is examined to decide whether data compression should take place. Figure 9 is a global view of the implementation.

Since the dictionary needs to be constantly updated so that each character can be compared to exactly the n characters preceding it, our implementation uses an array of $2n$ processors; that is, instead of just storing the last n characters, we store the last $2n$ characters, as depicted in Figure 10. The top row will serve as the dictionary consisting of processors numbered 1 through $2n$, from right to left. The bottom row will consist of the data elements being processed. Each element will be processed by the processor directly above it. Two different kinds of timing pulses will be used, called *pulse* 1 and *pulse* 2. After every n *pulse* 1's, a *pulse* 2 will occur. Each processor has a counter that tells the processor how many *pulse* 1's have occurred since the last *pulse* 2. The comparisons are done in blocks of n characters at a time. While these characters are processed, new characters are read in, and characters that have already been processed are transmitted (or pointers to them, if compression took place).

We begin with an informal explanation of how the algorithm works. Suppose that characters $x_1, \ldots, x_n$ have already been processed and, while these characters were being processed, $x_{n+1}, \ldots, x_{2n}$ were read in. Figure 11a shows what this situation would look like in our array. Since we have just finished processing a

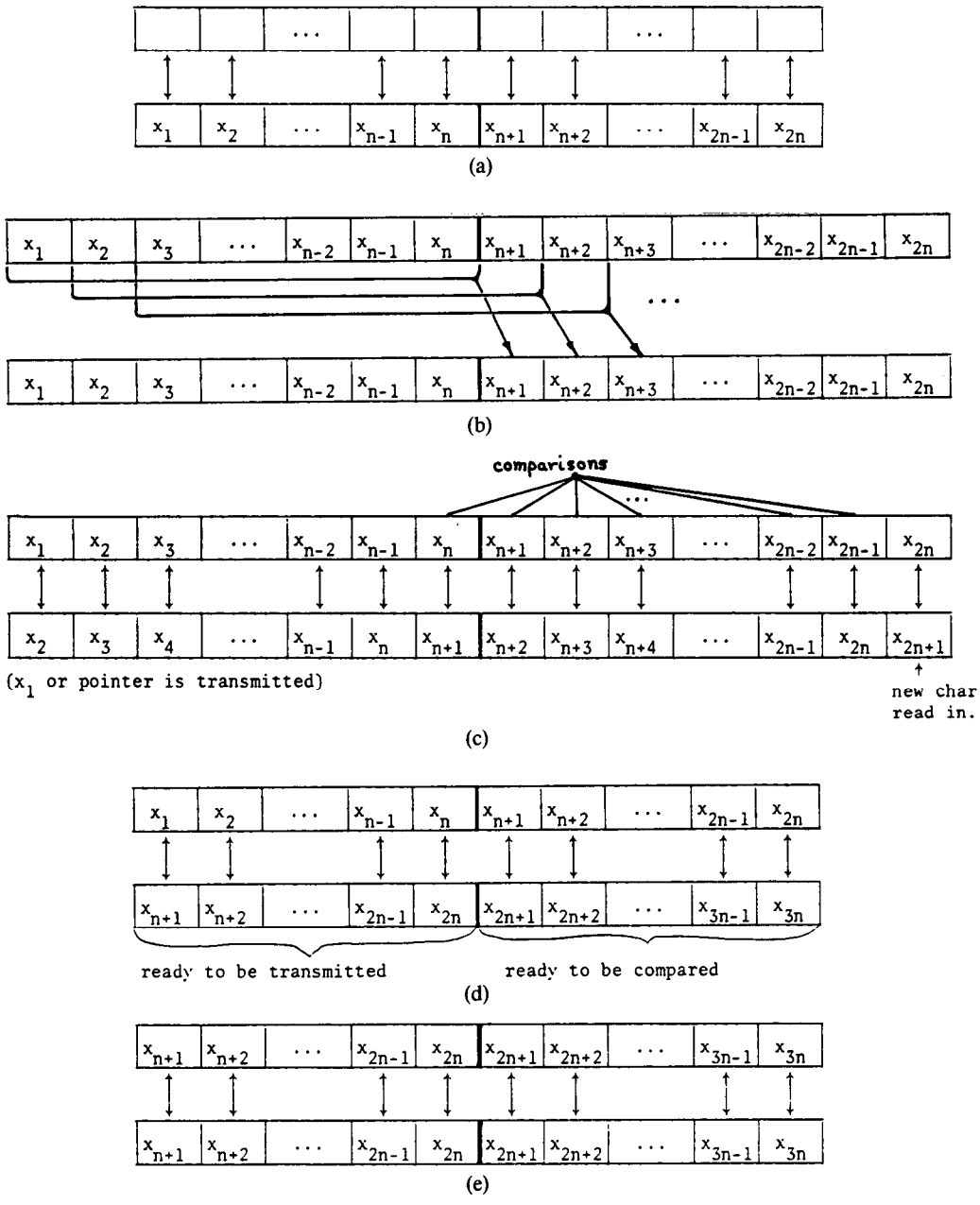


FIGURE 11

block of n characters, a *pulse* 2 would occur. On a *pulse* 2, every processor will copy the character below it into some (local) memory location, call it *char*. Note that we want to compare x_{n+i} with x_i through x_{n+i-1}, which correspond to the n characters preceding x_{n+i}; Figure 11b illustrates this. Figure 11c shows what occurs after a *pulse* 1 takes place. As this figure depicts, all characters are shifted, the leftmost character (or a pointer) is transmitted, the block $x_{n+1}, \ldots, x_{2n}$ is involved in the comparisons, and a new character is read. After each of the n pulses, this process is repeated, giving us the situation shown in Figure 11d. After the *n pulse*

1's, a *pulse* 2 will occur. This pulse will tell the processors to copy the character below into *char*. Figure 11e shows what the array will look like after this pulse. The whole process is then repeated.

Note that we have not discussed how to obtain the [*location, length*] information that is attached to each character. This information must be updated after each comparison. We shall present two schemes for doing this; but for the moment, to simplify the presentation, we are presenting the systolic algorithm without any provisions to update this information.

It is important to note that not all processors do the same thing all the time. For example, Processor 2 will make a comparison right after the first *pulse* 1, but after the next *pulse* 1's, it will only be reading characters. For this reason, every processor needs to keep a count of the number of *pulse* 1's. Each processor also needs to know where in the array it is located, and so a processor number is assigned to each one. For example, suppose we are starting out with the situation shown in Figure 11a, and suppose that processor i is in the right half of the array (i.e., $i \leq n$). Then notice that processor i will process x_{2n} at the $(i-1)$th *pulse* 1. Since x_{2n} is the last character of the block that is being processed, we know that processor i need not make any more comparisons. Similarly, suppose that processor j is in the left half of the array (i.e., $j > n$). Notice that x_{n+1} will not reach processor j until the jth (*mod n*) *pulse* 1. Suppose $j = n + 2$, then x_{n+1} will reach this processor at the second *pulse* 1. Therefore, processor j does not need to make any comparisons until the jth (*mod n*) *pulse* 1.

The leftmost processor (processor $2n$) has the additional special function of deciding what gets transmitted. At *pulse* 1, it looks at the [*location, length*] node of the character it is processing. If *length* $\leq p$, then it transmits the character. If *length* $> p$, then it transmits the first character of the pointer [*location, length*]. In addition, it sets the (local) memory location *buffercount* to *length* $- 1$, (*buffercount* is initially 0), and places the remaining $p - 1$ characters of the pointer in a (local) buffer. On the next successive $p - 1$ pipe shifts, *buffercount* is decremented, and the character coming from the right in the pipe is ignored and, instead, the next character of the buffer is transmitted. On the following successive pipe shifts (until *buffercount* becomes 0), *buffercount* is decremented, the character coming from the right is ignored, and nothing is transmitted. Once *buffercount* equals zero, this processor will begin transmitting again.

Algorithm 5 (Figure 12) is the encoding algorithm, less the facility to update the [*location, length*] information. We now present two schemes updating this information; that is, an implementation for the procedure **update** of Algorithm 5.

Scheme 1. We build a binary tree of processors where the leaves are the processors of the sliding dictionary; for simplicity, we assume that n is a power of 2. The algorithm works as follows. After each comparison, each processor that matched (i.e., for which *char* matched the character below) checks to see if it is at the start or at the end of a matching string. If the processor at its left did not match, then this processor sends a left bracket ("[") to its parent processor in the binary tree. Similarly, if the processor at its right did not match, then it sends a right bracket ("]") to its parent processor. These left and right brackets are sent up in the tree until some processor is able to pair them up, calculate the length of the matching string, and then return this value down the tree to the processor that sent the left bracket (i.e., to the processor currently processing the first character of the matching string). This processor will then update the [*location, length*] information if the length of this matching string is greater than the current *length* value. If the

```
during pulse 2 do
  begin
  if (processor# = 2n) then
    begin
    if (buffercount = 0) then
      begin
      if (length ≤ p) then transmit character
      else
        begin
        transmit first character of the pointer [location, length]
          (and put the remaining characters in local buffer)
        buffercount = length − 1
        end
      end
    else
      begin
      if (buffer not empty) then transmit next character of buffer
      buffercount = buffercount − 1
      end (*)
    end
  read character from processor to right (or from input stream if processor# =1)
  pulse 1 count = pulse 1 count +1
  if (pulse 1 count < processor# ≤ pulse 1 count + n) then update [location, length]
  end
during pulse 2 do
  begin
  Copy character below into char
  pulse 1 count = 0
  end
```

FIG. 12. Algorithm 5. Sliding dictionary encoding algorithm, less **update** procedure.

value is not greater, then it must be that there exists some other longer string starting with this character that matched. If this is not the case, we update by writing the length of this matching string and by writing this processor number as the location. Figure 13 illustrates an example.

The input to any of the processors in the binary tree (excluding the leaves) can be as shown in Table I.

If we were to read the brackets of the sliding dictionary from left to right, we would find a sequence of the form [ ] [ ] [ ] · · · [ ]. Hence, a processor in the binary tree will not receive as input two consecutive ['s or]'s, and so the only possible inputs are the ones shown above. Thus, every processor can receive at most four brackets as input, and can send at most two brackets as output.

Algorithm 6 (Figure 14) is the procedure **update** to be used by Algorithm 5 to implement Scheme 1 and Algorithm 7 (Figure 15) is the algorithm used by the processors of the tree. In Algorithm 6, *curchar* denotes the current character below this processor in the pipe (so the test *curchar* = *char* checks if this processor has a match). In Algorithm 7, it is assumed that the processor number is passed along with the bracket.

As we observe from Algorithm 7, only the first character of the matching string knows the length of this string. This could be a problem when overlapping pointers can occur. For example, suppose that the input currently being processed is *bcdefgh* and suppose that *bcd* matched at location i of the dictionary, and suppose that *defgh* matched at location j of the dictionary. In this case, b has node $[i, 3]$ and d has node $[j, 5]$, but the other characters do not have any information about these

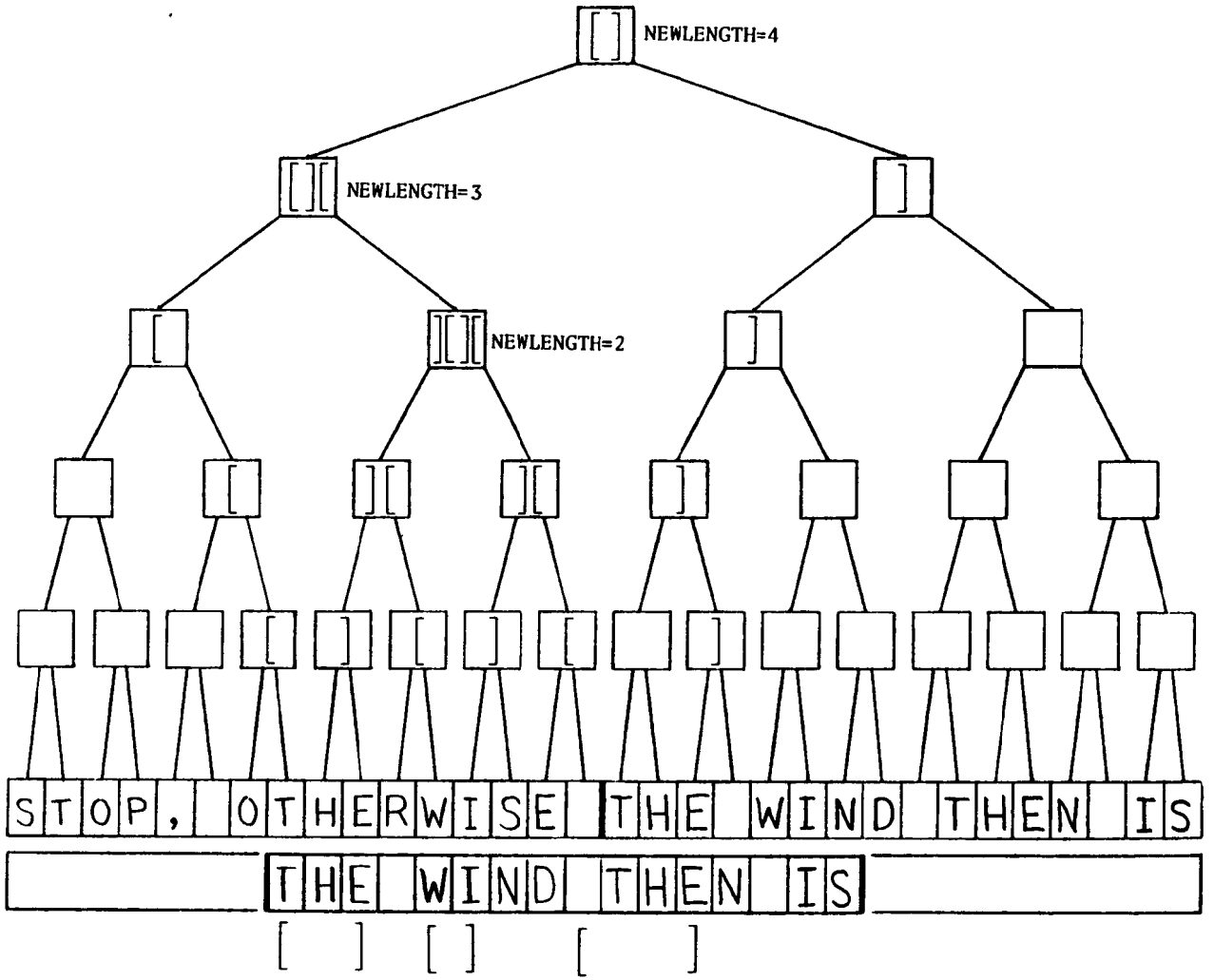

NEWLENGTH=4
NEWLENGTH=3
NEWLENGTH=2
STOP, OTHERWISE THE WIND THEN IS
THE WIND THEN IS

FIGURE 13

TABLE I

Number	Input	Action
(1)	Nothing	None
(2)	[	Send [to parent
(3)	]	Send] to parent
(4)	][	Send] [to parent
(5)	[]	Calculate length and send it to the processor that sent it
(6)	[]]	Handle first two brackets as in (5) and send last bracket to parent
(7)	][]	Send first to parent and handle last two brackets as in (5)
(8)	][][	Send first and last to parent and handle middle two as in (5)

```
if (curchar = char) then
  begin
  if (processor to the left did not match) & (processor to the right matched) then
    begin
    Send to parent processor (processor#, [ )
    Wait for parent to respond with newlength
    if (newlength > length) then
      begin
      length = newlength
      location = processor#
      end
    end
  else if (processor to right did not match) & (processor to the left matched)
    then send to parent (processor#, ] )
  end
```

FIG. 14. Algorithm 6. **update** procedure, scheme 1: Algorithm for sliding dictionary elements.

matches. So, if *bcd* were compressed, we would like character *e* to know that *efgh* matched at location $j-1$ of the dictionary. In other words, if *e* does not have a match longer than 4 in its node, we would like to update its node by making it $[j-1, 4]$. Processor $2n$ (the leftmost processor) can take care of this. Referring to Algorithm 5, recall that *buffercount* contains the number of characters that remain to be "ignored" because they are part of the pointer that was last transmitted. Suppose a character x is part of the pointer last transmitted, so when it reaches processor $2n$, *buffercount* $\neq 0$. If the value *length* of its node is greater than *buffercount*, then there are some characters from the matching string represented by the node that are not part of the pointer last transmitted. In this case, its node information is transmitted to the character on the right. Hence, to modify the algorithm all we have to do is to execute Algorithm 8 (Figure 16) whenever statement (*) is executed in Algorithm 5.

Algorithms 6 and 7 work by attaching a binary tree to the encoding pipe. This structure violates the basic notion of a systolic array in two ways. The first is probably not a serious issue in practice; it is that there is a $O(\log(n))$ time delay between pipe shifts (for brackets to propagate up and down the tree). The second problem concerns layout. In Storer [10], it is shown how to lay out such a structure on a $O(\sqrt{n})$ by $O(\sqrt{n})$ grid so that pipe edges have length bounded above by a constant (independent of n) and contain no bends; Figure 17 shows such a layout for a binary tree connected to a pipe of 64 nodes. However, the tree edges in this layout have length as large as $O(\sqrt{n})$. In addition, Paterson et al. [4] show $O(\sqrt{n}/\log(n))$ to be a lower bound on the maximum tree edge length; hence, little

```
begin
case of brackets received begin
  no brackets: do nothing
  [: begin
    Send [ to parent
    wait = true
    end
  ]: begin
    Send ] to parent
  ][:begin
    Send ][ to parent
    wait = true
    end
  [ ]: begin
    newlength = (processor# of [ ) − (processor# of ] ) + 1
    Send newlength to child that sent the [
    end
  [ ][: begin
    Handle [ ] as is done for case [ ]
    Send [ to parent
    wait = true
    end
  ][ ]: begin
    Send ] to parent
    Handle [ ] as is done for case [ ]
    end
  ][ ][: begin
    Send first ] and last [ to parent
    Handle [ ] as is done for case [ ]
    wait = true
  end
if (wait = true) then
  begin
  Wait for parent to transmit newlength
  Transmit newlength to child that sent the [
  end
end
```

FIG. 15. Algorithm 7. **update** procedure, scheme 1: Algorithm for tree elements.

```
begin
if (length > buffercount) & (length − 1 > (length attached to character to right))
then
  begin
  (length attached to character to right) = length − 1
  (location attached to character to right) = location − 1
  end
end
```

FIG. 16. Algorithm 8. Scheme 1: Modification to Algorithm 5 to handle overlapping pointers.

reduction is possible for the maximum length of tree edges in the construction of Storer [10]. Furthermore, long edges are a potential problem for large VLSI layouts. Although there has been much debate as to the right model for a VLSI signal to propagate along a wire (see Bilardi et al. [1]), many authors argue that a constant delay model is not realistic and some argue that as much as $O(n^2)$ should be charged to an edge of length n. Thus, although Algorithm 6 has only a $\log(n)$ logical delay, this must be balanced against the long tree edges. Scheme 2, to be presented next, eliminates long edges, but introduces a longer logical delay.

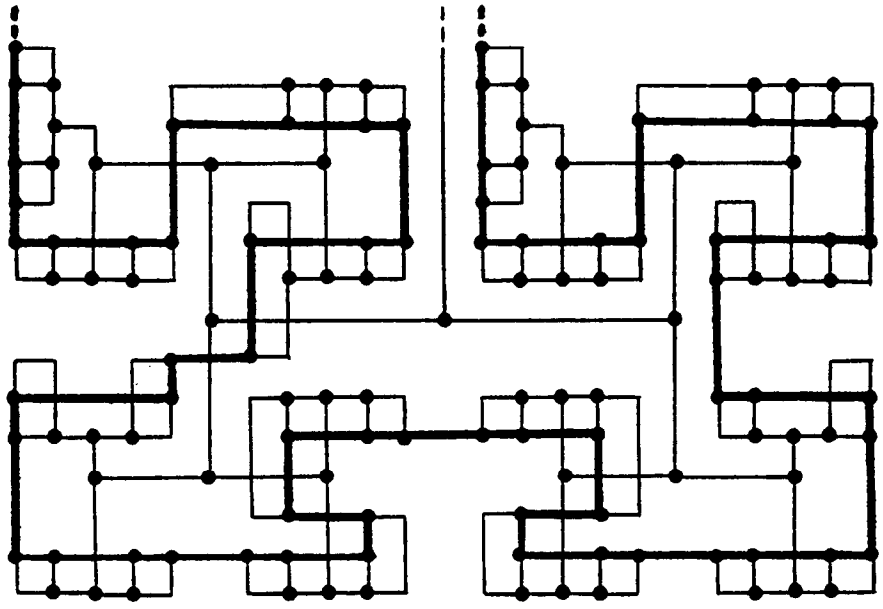


FIGURE 17

Scheme 2. This scheme requires two additional special pulses; *pulse* 3 and *pulse* 4. After every pipe shift, there are $2\sqrt{n}$ *pulse* 3's followed by $\sqrt{n}$ *pulse* 4's. We first describe how the *pulse* 3's are used.

The purpose of *pulse* 3's is to signal each processor to pass information to the processor on the left about the consecutive matches starting with itself. For instance, suppose that processors i through $i - 3$ matched. Then on the first *pulse* 3, processor i knows that processor $i - 1$ matched, processor $i - 1$ knows that processor $i - 2$ matched, and processor $i - 3$ knows that processor $i - 4$ did not match. Since each processor now knows about the processor that follows it, it can now transmit this information on the next *pulse* 3. So, after k pulses, each processor can know the number of consecutive matches of length $\leq k$, starting with itself. We require each processor to have a memory location, call it *sum* 3, where it will count the number of consecutive matches. First, if the processor matched, it will set *sum* 3 = 1. At the first *pulse* 3, it will transmit a 1 to the left indicating that it matched. It will also receive information from the right, telling if processor to the right matched or not. If it receives a 1, it will increment *sum* 3. At the next *pulse* 3, it will again transmit a 1, because it last received a 1. This is repeated until it receives a 0 (non-match), in which case it updates node [*location, length*] to [Processor#, *sum* 3] if *sum* 3 > *length* (*sum* 3 > *length* if this current match is longer than any other encountered). In addition, *sum* 3 is set to 0. From then on, it will transmit a 0 to the left and *sum* 3 will remain 0 regardless of what information may be received.

We now describe how the *pulse* 4's are used. Assuming that the pipe is laid out in a snake fashion as depicted in Figure 3, the *pulse* 4's are used to transmit the information gathered by the *pulse* 3's vertically, as depicted by the dashed lines in Figure 18. The treatment of the *pulse* 4's is similar to that of the *pulse* 3's. As with *pulse* 3, we assume that the processors in row 1 will receive 0 at *pulse* 4 indicating

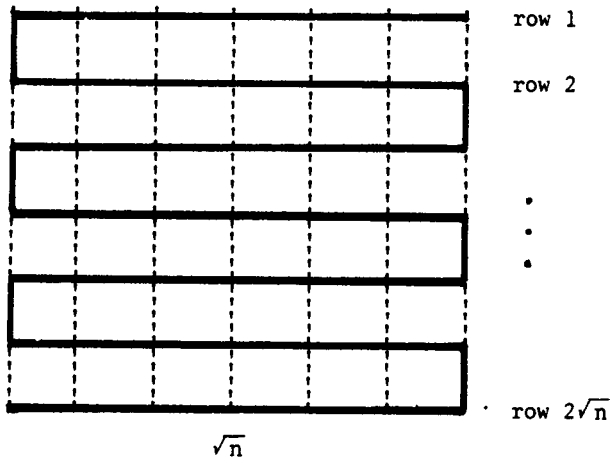


FIGURE 18

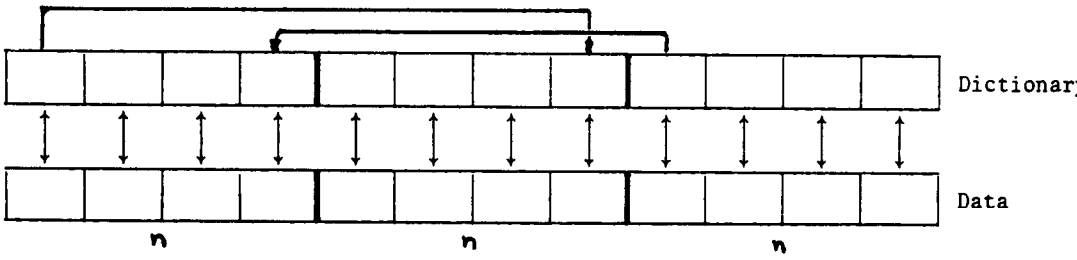
0 more characters matched, and the processors in row $2\sqrt{n}$ will not transmit anything. Each processor has an additional memory location, call it *sum* 4, used to count the number of consecutive matches. After all the *pulse* 3's, each processor knows how far back it matched (up to $2\sqrt{n}$), and the value of *sum* 3 will be copied into *sum* 4. If *sum* 4 = $2\sqrt{n}$ at the end of the *pulse* 3's, then it is possible that the string that matched is longer. Suppose that processor i has value *sum* 4 = $2\sqrt{n}$. Notice that processor $i - 2\sqrt{n}$ is exactly two rows above it, and processor $i - 4\sqrt{n}$ is two rows above processor $i - 2\sqrt{n}$, and so on. Therefore, the *pulse* 4's that processor i is interested in are the even-numbered ones. At odd-numbered *pulse* 4's, processor i will simply take the information it receives, store it, and transmit it at the next pulse. On even numbered pulses, it takes the value it receives and adds it to *sum* 4. If the value it received was $2\sqrt{n}$, then it will wait for the next even pulse because it is possible that the matching string is longer than the value in *sum* 4. However, if the value it received was less than $2\sqrt{n}$, it will update the node as in the previous algorithm, setting it to [Processor#, *sum* 4] if *sum* 4 > *length*. In addition, *sum* 4 is set to 0.

Thus, unlike Scheme 1, Scheme 2 has an extremely simple layout (the grid connection shown in Figure 18) with no long edges. However, the logical delay is $O(\sqrt{n})$ instead of $O(\log(n))$. Clearly, Scheme 2 can be modified to have a logical delay of k, for any $k \geq 1$, if one is willing to limit the maximum length of a target to k, a possible practical compromise. In this case, k would (presumably) be less than $\sqrt{n}$, and so the *pulse* 4's (and the vertical connections) would not be needed.

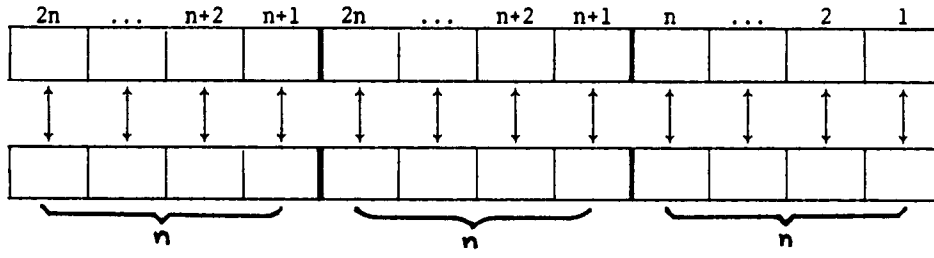
7. *Sliding Dictionary Model, Decoding*

As the compressed data is received by the decoding chip, it gets decoded into the original text and transmitted to the receiving processor; the function of this chip is to expand all the pointers of the encoded data. All pointers point to earlier strings that are at most n characters away (since they were in the sliding dictionary when these strings were being compressed). Assuming we store the last n characters (after decompression), then the string represented by a pointer (*loc*, *lgth*) is obtained by concatenating the characters from processor *loc* to processor *loc* − *lgth* + 1.

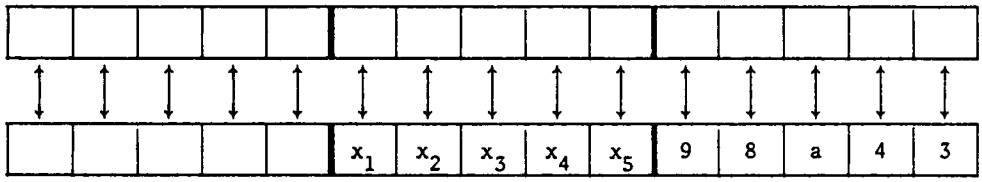
Like the encoding algorithm, the decoding algorithm employs a sliding dictionary. It uses an array of length $3n$ with two extra connections as shown in



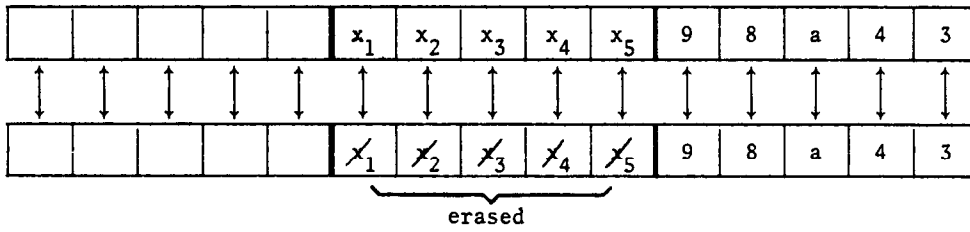
(a)



(b)



(c)



(d)

FIGURE 19

Figure 19a. The following theorem notes that these two extra connections do not cause a problem for the circuit layout.

THEOREM 3. *The array of Figure* 19*a can be laid out on an* $O(\sqrt{n})$ *by* $O(\sqrt{n})$ *grid so that all edges, including the two extra connections, have length* $O(1)$.[12]

[12] Although it is not directly relevant to the results of this paper, it should be noted that the construction to be presented can be generalized to any fixed number of extra connections, placed at arbitrary points in the pipe.

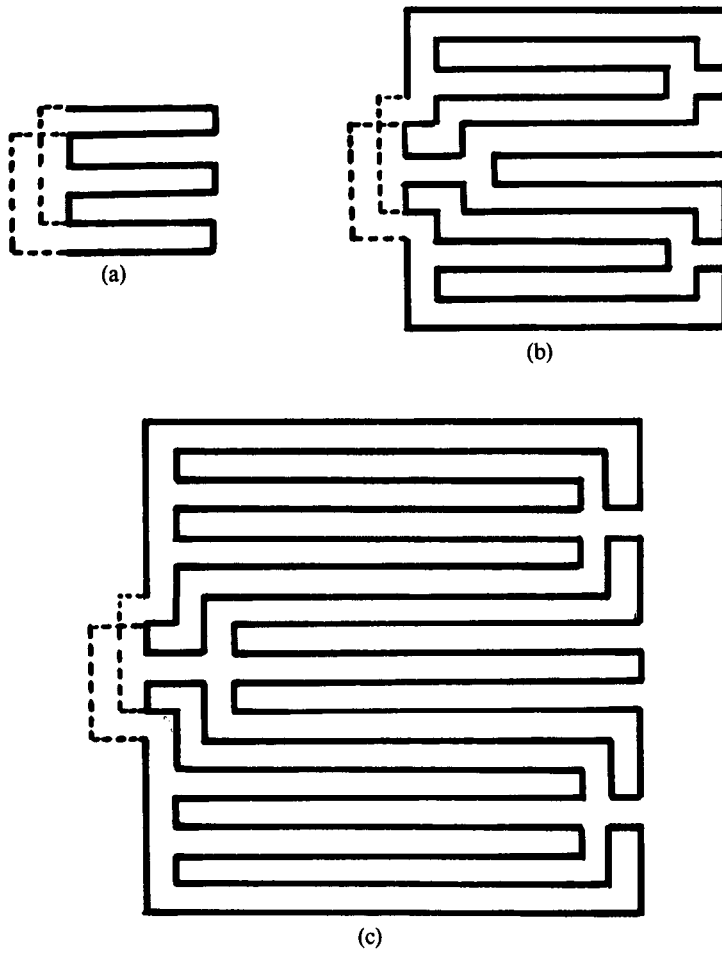


FIGURE 20

PROOF. If $n = (3k)^2$ for some even integer k, then the pipe can be placed on a grid of size exactly $3k$ by $3k$; Figures 20a, b, and c show the layout for $k = 2$, 4, and 6, respectively. This construction can be viewed as a generalization of the snake layout of Figure 3. It is essentially three snake layouts of size k by $3k$. Each of these three layouts has exactly $k(3k - 4)$ nodes in columns 3 through $3k - 2$. The top and bottom snakes each have $(3k - 2)/2$ nodes in each of the first two columns, whereas the middle snake only has 2 nodes in each of the first two columns. On the other hand, the middle snake has $2k - 2$ nodes in each of the last two columns whereas the top and bottom snakes each have only $(k + 2)/2$ nodes in each of the last two columns. Hence, each of the three snakes has exactly $3(k^2)$ nodes. In addition, the two extra connections have constant lengths (6 and 8 units, respectively). If n is not of the form $(3k)^2$ for some even k, then first, the construction can be carried out for the least integer m larger than n which is of this form, and then nodes can be deleted from each of the three snakes (by trimming back from the rightmost columns). Since it must be that $m \leq (\sqrt{n} + 5)^2 = n + 10\sqrt{n} + 25$, for sufficiently large n, $m < 2n$ (in fact, for any $\epsilon > 0$, $m < (1 + \epsilon)n$ for sufficiently large n).

Two different kinds of timing pulses will be used, called *pulseA* and *pulseB*; after every $2n$ *pulseA*'s, there is one *pulseB*. The processors will contain a counter for the *pulseA*'s. Each processor is assigned a processor number such that those processors on the leftmost third will be assigned the same numbers as those on the middle third; that is, the processor that would normally be processor $2n + i$ will be assigned the number $n + i$. Figure 19b depicts this numbering.

Decoding is done in blocks of n characters. While this is being done, more input is being read in. A pointer (*loc, lgth*) arriving to the pipe is transformed into the sequence of integers *loc*, $loc - 1$, $loc - 2$, . . . , $loc - lgth + 1$. For example, pointer (5, 4) would become 5, 4, 3, 2. Each one of these integers represents the location of the systolic array in which that character of the original string may be found. These characters need to be marked in some way to distinguish them from integers that are part of the text; to simplify the presentation, these details are left out of the algorithm to be presented. After pointers are expanded into such a sequence of integers, these integers are entered into the pipe on every even *pulseA*.[13] As was discussed in Section 3, unless some assumptions are made regarding the relative speeds of the chip and the data rate of the communication channel, there is no guarantee that another pointer will not arrive before the integers for the current pointer have been entered into the pipe (and there is no bound on the maximum size of a buffer to handle this). As discussed in Section 3, this is not likely to be a problem in practice, and we do not address this issue further here.

Let us illustrate the algorithm via an example. Suppose that $n = 5$, and suppose that $x_1, \ldots, x_5$ were the last 5 characters processed. In addition, suppose that the next thing that was read was (9, 2)*a*(4,2). Figure 19c shows what this situation would look like in our array. Since we have just finished processing a block of n characters, a *pulseB* would occur. On a *pulseB* each processor copies the character (or digit) below it into its memory location *char*. Figure 19d shows this. Note, in Figure 19b, the block $x_1, \ldots, x_5$ was transmitted while it was being decoded. So, it can now be erased from the data elements. After each *pulseA*, the block 9, 8, *a*, 4, 3 will get shifted one processor to the left. Each processor will look at the element below it. If it is a character, it does not do anything. But if it is a digit, it checks to see if the processor# is equal to this digit. If so, it replaces this digit by the contents of its memory location *char*. There will be $2n$ *pulseA*'s. At every even-numbered pulse, a new character will be read in. The new characters that are read in will only get shifted at even-numbered pulses so that at the end of all the *pulseA*'s, they will all be underneath the rightmost n processors. The function of the leftmost n processors is the same as that of the middle block. Their functions are exhanged after every block of n characters is processed. So, during the next set of *pulseA*'s, the pointers of the next block will be expanded by using the characters from the leftmost n processors.

Figure 21 illustrates the algorithm by continuing the example of Figure 19d. As can be seen from Figure 21, it is possible to have pointers to pointers; after the second *pulseA*, processor 4 had a 4 below it, and replaced it with an 8 (the character at location 4 was compressed with a pointer that pointed to location 8). Notice that, after the tenth pulse, the block just decoded is underneath the leftmost processors.

Algorithm 9 (Figure 22) is the decoding algorithm. Referring to Algorithm 9, note that processor n will pass the character (or digit) below it to the middle block

[13] Characters enter the pipe only on even *pulse A*'s because every block of n characters that enters the pipe will have to pass by $2n$ processors of the pipe before the next block of n characters enters the pipe.

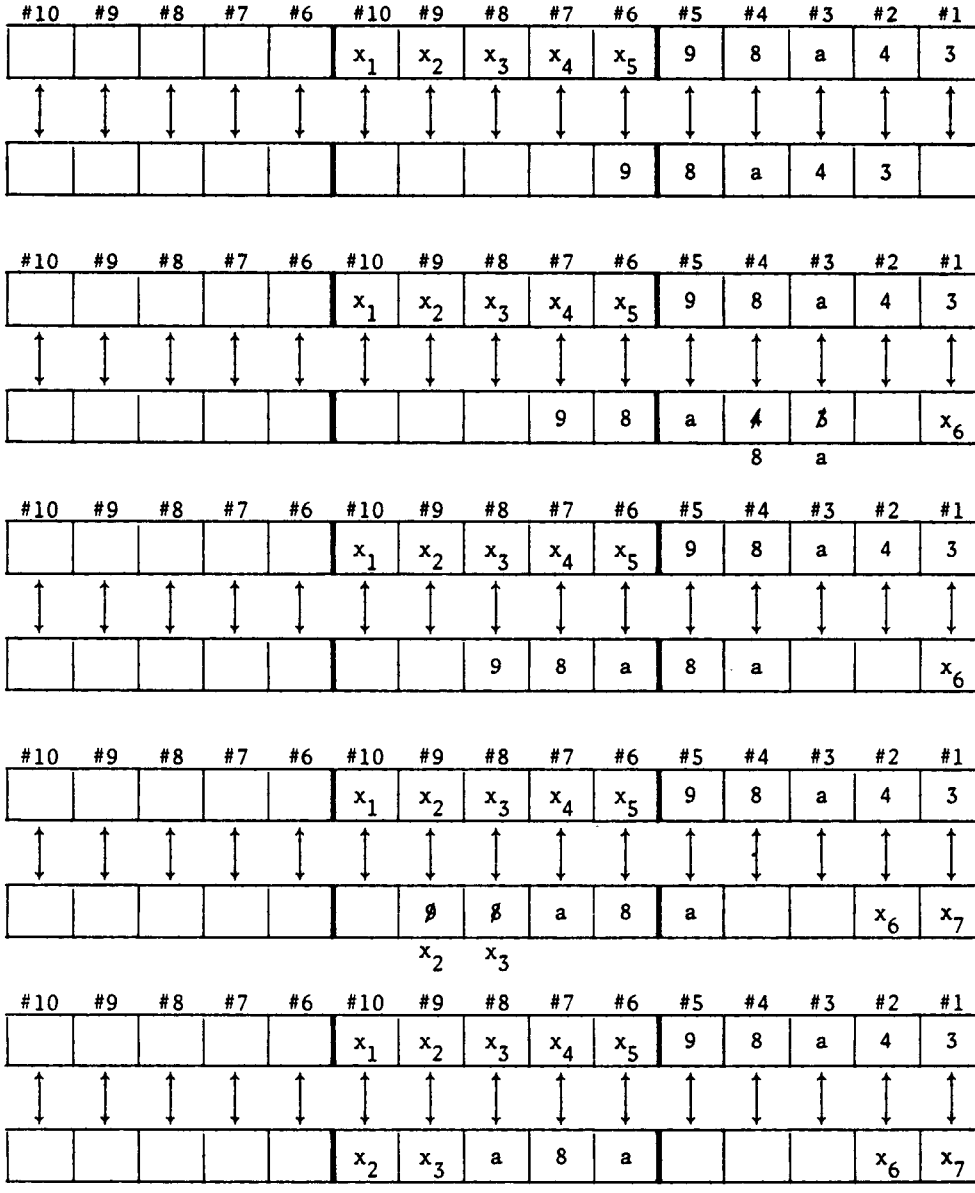

#10 #9 #8 #7 #6 #10 #9 #8 #7 #6 #5 #4 #3 #2 #1
x1 x2 x3 x4 x5 9 8 a 4 3
9 8 a 4 3
#10 #9 #8 #7 #6 #10 #9 #8 #7 #6 #5 #4 #3 #2 #1
x1 x2 x3 x4 x5 9 8 a 4 3
9 8 a 8 a x6
#10 #9 #8 #7 #6 #10 #9 #8 #7 #6 #5 #4 #3 #2 #1
x1 x2 x3 x4 x5 9 8 a 4 3
9 8 a 8 a x6
#10 #9 #8 #7 #6 #10 #9 #8 #7 #6 #5 #4 #3 #2 #1
x1 x2 x3 x4 x5 9 8 a 4 3
x2 x3 a 8 a x6 x7
#10 #9 #8 #7 #6 #10 #9 #8 #7 #6 #5 #4 #3 #2 #1
x1 x2 x3 x4 x5 9 8 a 4 3
x2 x3 a 8 a x6 x7

FIGURE 21a

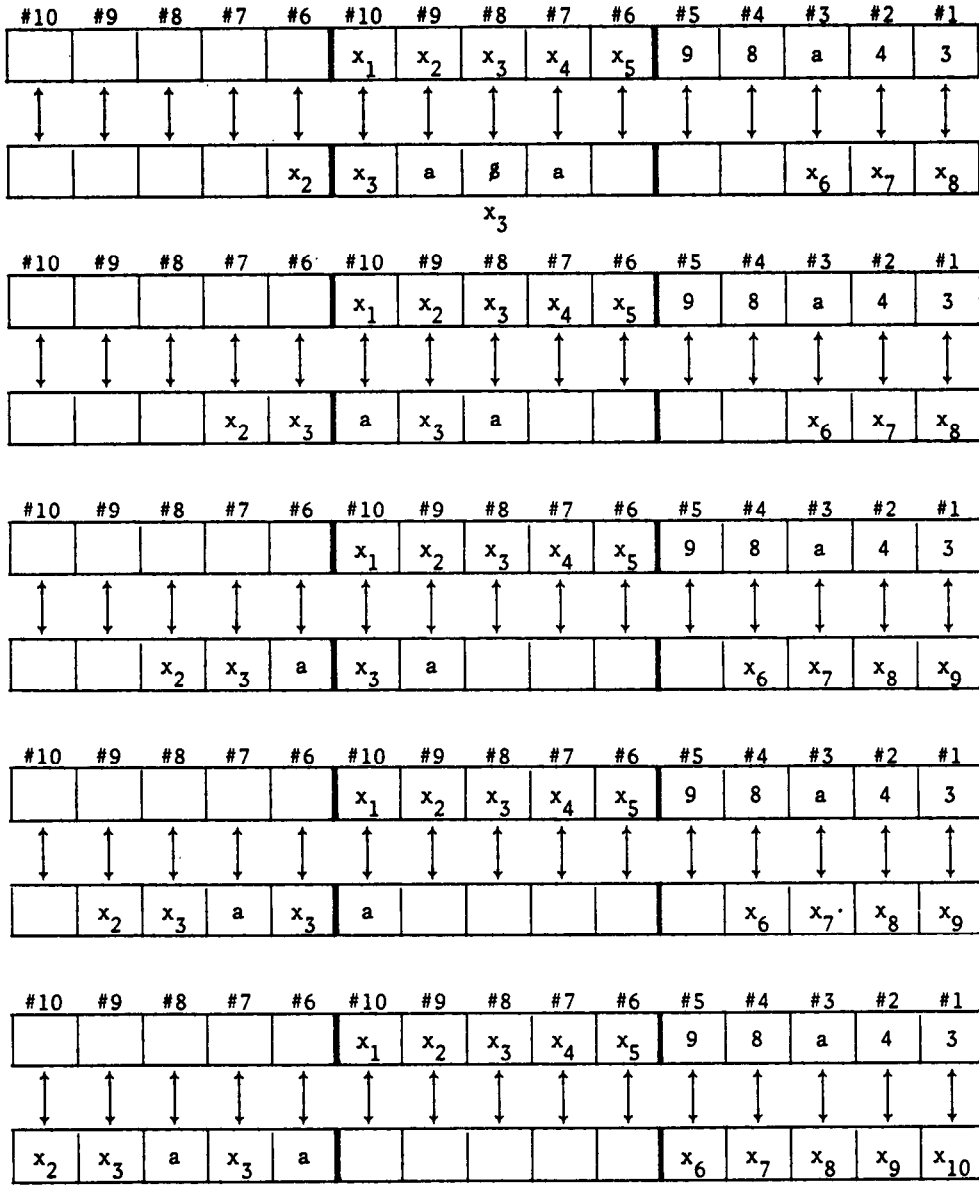

#10 #9 #8 #7 #6 #10 #9 #8 #7 #6 #5 #4 #3 #2 #1
x1 x2 x3 x4 x5 9 8 a 4 3
x2 x3 a g a x6 x7 x8
x3
#10 #9 #8 #7 #6 #10 #9 #8 #7 #6 #5 #4 #3 #2 #1
x1 x2 x3 x4 x5 9 8 a 4 3
x2 x3 a x3 a x6 x7 x8
#10 #9 #8 #7 #6 #10 #9 #8 #7 #6 #5 #4 #3 #2 #1
x1 x2 x3 x4 x5 9 8 a 4 3
x2 x3 a x3 a x6 x7 x8 x9
#10 #9 #8 #7 #6 #10 #9 #8 #7 #6 #5 #4 #3 #2 #1
x1 x2 x3 x4 x5 9 8 a 4 3
x2 x3 a x3 a x6 x7 x8 x9
#10 #9 #8 #7 #6 #10 #9 #8 #7 #6 #5 #4 #3 #2 #1
x1 x2 x3 x4 x5 9 8 a 4 3
x2 x3 a x3 a x6 x7 x8 x9 x10

FIGURE 21b

```
during pulseA do
  begin
  pulseAcount = pulseAcount + 1
  if (pulseAcount is even) or (processor# > pulseAcount) then shift left
  if (processor has a digit below it) & (processor# = digit)
    then replace digit by char
  if (processor# = 2n) then
    if (odd block and middle) or (even block and leftmost) then transmit
  end
during pulseB do
  begin
  Copy character below into char
  if (processor# > n) then erase character below
  pulseAcount = 0
  end
```

FIG. 22. Algorithm 9. Sliding dictionary decoding algorithm.

if an odd-numbered block is being processed, or to the leftmost block if an even-numbered block is being processed. The leftmost processor will pass to processor $n + 1$ in middle block. Note that, as mentioned earlier, Algorithm 9 assumes that pointers entering the pipe are expanded to a sequence of integers (e.g., a pointer $(i, 3)$ is expanded to the three integers i, $i + 1$, and $i + 2$). Also, omitted from Algorithm 9 are the details of marking integers produced in this way (so that they can be distinguished from characters).

8. *Conclusion*

The structures presented here are easily implemented in current n-MOS technology. Further research concerning the details of the layouts described here could provide an estimate of the size of a dictionary that could be placed on a single chip. As mentioned earlier, an important area for future research is the study of algorithms and VLSI structures for dynamic dictionary selection (for the static dictionary model). Such research is likely to involve approximation algorithms since the NP-completeness of optimal dictionary selection follows from results contained in Storer [9].

REFERENCES

1. BILARDI, G., PRACCHI, M., AND PREPARATA, F. P. A critique and appraisal of VLSI models of computation. In *Conference on VLSI Systems and Computations.* Carnegie-Mellon Univ., Pittsburgh, Pa., 1981, pp. 81–88.
2. KUCERA, H., AND FRANCIS, W. N. *Computational Analysis of Present-Day American English.* Brown University Press, Providence, R.I., 1967.
3. MEAD, C., AND CONWAY, L. *Introduction to VLSI Systems.* Addison-Wesley, Reading, Mass., 1982.
4. PATTERSON, M. S., RUZZO, W. L., AND SNYDER, L. Bounds on minimax edge length for complete binary trees. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing* (Milwaukee, Wisc., May 11–13). ACM, New York, pp. 293–299.
5. RODEH, M., PRATT, V. R., AND EVEN, S. Linear algorithm for data compression via string matching. *J. ACM 28*, 1 (Jan. 1981), 16–24.
6. SEERY, J. B., AND ZIV, J. A universal data compression algorithm: Description and preliminary results. Tech. Memorandum 77-1212-6. Bell Laboratories, Murray Hill, N.J., 1977.
7. SEERY, J. B., AND ZIV, J. Further results on universal data compression. Tech. Memorandum 78-1212-8. Bell Laboratories, Murray Hill, N.J., 1978.
8. SHANNON, C. E. Prediction of entropy of printed English text. In *Key Papers in the Development of Information Theory,* D. Slepian, Ed. IEEE, New York, 1973, pp. 42–46.

9. STORER, J. A. NP-completeness results concerning data compression. Tech. Rep. 234. Dept. Electrical Engineering and Computer Sci., Princeton Univ., Princeton, N.J., 1977.
10. STORER, J. A. Combining pipes and trees in VLSI. Tech. Rep. CS-82-107. Dept. Computer Sci., Brandeis Univ., Waltham, Mass., 1982.
11. STORER, J. A., AND SZYMANSKI, T. G. Data compression via textual substitution. *J. ACM 29,* 4 (Oct. 1982), 928–951.
12. WAGNER, R. A. Common phrases and minimum-space text storage. *Commun. ACM 16,* 3 (Mar. 1973), 148–152.
13. ZIV, J., AND LEMPEL, A. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory 23,* 3 (1977), 337–343.

RECEIVED MARCH 1983; REVISED OCTOBER 1984; ACCEPTED NOVEMBER 1984