Parallel Parsing Algorithms for Static Dictionary Compression

Hideo Nagumo, Member, IEEE, Mi Lu, Senior Member, IEEE, and Karan L. Watson, Fellow, IEEE

Abstract—The data compression based on dictionary techniques works by replacing phrases in the input string with indexes into some dictionary. The dictionary can be static or dynamic. In static dictionary compression, the dictionary contains a predetermined fixed set of entries. In dynamic dictionary compression, the dictionary changes its entries during compression. We present parallel algorithms for two parsing strategies for static dictionary compression. One is the optimal parsing strategy with dictionaries that have the prefix property, for which our algorithm requires $O(L + \log n)$ time and $O(n)$ processors, where $n$ is the number of symbols in the input string, and $L$ is the maximum length of the dictionary entries, while previous results run in $O(L + \log^2 n)$ time using $O(n^2)$ processors or in $O(L + \log^* n)$ time using $O(n)$ processors. The other is the longest fragment first (LFF) parsing strategy, for which our algorithm requires $O(L + \log n)$ time and $O(n \log L)$ processors, while a previous result obtained an $O(L \log n)$ time performance on $O(n/\log n)$ processors. For both strategies, we derive our parallel algorithms by modifying the on-line algorithms using a pointer doubling technique.

Index Terms—Data compression, dictionary techniques, parallel algorithms, PRAM, optimal parsing, longest fragment first parsing.

1 INTRODUCTION

We present parallel algorithms for two parsing strategies for static dictionary compression: the optimal parsing strategy with dictionaries that have the prefix property and the longest fragment first (LFF) parsing strategy. The increasing speed of modern data processing systems, such as disk controllers and data transmission, requires the data compression to be accomplished at a very high speed. Parallelism can help with increasing speed and compression effectiveness. Parallel compression is therefore adopted in many applications for which sequential compression does not meet the performance criteria. The data compression based on dictionary techniques is popular for desk-top data compression utilities and fast hardware data compression systems. It works by replacing groups of consecutive symbols (phrases) in the input string with the corresponding dictionary indexes [1]. The dictionary can be static or dynamic. In static dictionary compression, the dictionary contains a predetermined fixed set of entries. In dynamic dictionary compression, the dictionary changes its entries during compression, reflecting newly encoded phrases in the input string.

The central part of dictionary techniques is the process of parsing the input string into phrases that are to be replaced with the corresponding dictionary indexes. There are different strategies for parsing. Many data compression programs based on dictionary techniques used today, e.g., Unix compress, employ a greedy parsing strategy. In this strategy, the input string is processed from left to right and, at each step, encoded is the longest phrase starting at the first symbol of the uncoded portion of the string that matches a dictionary entry.

The greedy parsing strategy is simple and efficient in sequential environment, but it has been shown that nongreedy parsing strategies can improve the performance of dictionary techniques [2], [3]. There are two nongreedy parsing strategies in the literature. They are the optimal parsing and LFF parsing strategies. An optimal parsing of the input string is a shortest possible sequence of dictionary indexes such that the concatenation of the corresponding dictionary entries forms the input string. If fixed length dictionary indexes are used, an optimal parsing produces the minimum number of dictionary indexes. In general, a sequential optimal parsing encoder must be able to look at arbitrarily large prefixes of the input string. However, if the dictionary has a property called the prefix property, optimal parsing can be computed on-line, or in a one-pass left-to-right scan with limited lookahead. A dictionary has the prefix property if all the prefixes of a phrase in the dictionary are also in the dictionary [4], [5]. We call a dictionary with this property a prefix dictionary.

An LFF parsing algorithm parses the input string by repeatedly locating the longest phrase in the uncoded portion of the input string that matches a dictionary entry. This phrase is replaced with the corresponding dictionary index [6]. In general, the compression performance of LFF parsing lies between greedy and optimal parsing [6].

Dictionary techniques are quite amenable to parallel processing since different positions of symbols in the input string can be compared to symbols in the dictionary independently [1]. Because of this property, several systolic array architectures of dictionary techniques have been proposed and some of them have been implemented in...
A number of papers have discussed efficient parallel algorithms for the dictionary techniques on parallel random access machines (PRAM) [4], [10], [11]. In this paper, we describe parallel parsing algorithms on PRAM. In this parallel processing model, processors work synchronously under the control of the central control unit, and they share the common random-access memory and communicate through this memory [12]. There are several variations of PRAM, and they are different in the assumptions on simultaneous reading and writing of the shared memory. Our algorithms in this paper assume the concurrent read exclusive write (CREW) PRAM model in which any number of processors may simultaneously read from the same memory location but no two processors may write simultaneously to the same memory location [12]. A number of parallel algorithms have been designed to solve string problems on PRAM [13].

Parallel algorithms for the optimal parsing strategy with prefix dictionaries have been studied by DeAgostino and Storer [4]. They presented two parallel algorithms for this problem with the complexities of \( O(L + \log n) \) time and \( O(n^2) \) processors, and \( O(L + \log^2 n) \) time and \( O(n) \) processors, respectively, where \( n \) is the number of symbols in the input string, and \( L \) is the maximum length of the dictionary entries. We present a parallel algorithm for this problem that requires \( O(L + \log n) \) time and \( O(n) \) processors. A parallel algorithm for the LFF parsing strategy was described by Stauffler and Hirschberg with the complexity of \( O(L \log n) \) time and \( O(n/\log n) \) processors [10]. We present a parallel algorithm for this problem that requires \( O(L + \log n) \) time and \( O(n \log L) \) processors.

The key to our parallel algorithms is that we derive them by modifying the on-line algorithms using a pointer doubling technique [12]. This derivation of parallel algorithms was used by Stauffler and Hirschberg for the greedy parsing algorithm [10] and can be applied to other on-line algorithms.

Our parallel algorithms in this paper and the parallel algorithms for static dictionary compression found in [4] address the problem of parsing. A parallel algorithm for producing the compressed output from the parsing that runs in \( O(\log n) \) time using \( O(n/\log n) \) processors can be found in [10].

The rest of this paper is organized as follows. Section 2 describes assumptions, notations, etc. Section 3 presents our parallel optimal parsing algorithm with prefix dictionaries. Section 4 presents our parallel LFF parsing algorithms. Section 5 concludes the paper.

2 Preliminaries

In this section, we summarize the assumptions and notations used in this paper.

2.1 Assumptions

Throughout this paper, we assume that the dictionary indexes have a fixed length. With this assumption, the number of dictionary indexes in a parsing determines the length of the compressed form. Given two parsings, the one with the smaller number of indexes will have the smaller compressed form.

We also assume that the dictionary \( D \) is represented as a suffix tree. A suffix tree for a dictionary \( D \) is a trie, or digital search tree data structure composed of all the phrases in \( D \). With this assumption, the time for searching for a phrase in \( D \) is proportional to the length of the phrase, and searching for all the prefixes of a phrase in \( D \) requires the same time complexity as searching for the phrase.

The set of dictionary entries must be complete in the sense that any fragment of the input string is representable by concatenation of dictionary entries [2]. To meet this requirement, we assume that \( D \) includes all the members of the input symbol set [10].

Also, we assume that the lengths of the entries of \( D \) are less than or equal to some maximum length \( L \), and we sometimes assume that \( L \) is equal to \( O(\log n) \) [4], [10]. This is a reasonable assumption in practice because, when compressing text with a dictionary of 64K entries, the average match length will be about five characters, and limiting the lengths of dictionary entries to 16 does not have significant effect on the compression performance [4].

2.2 Notations

An input string has \( n \) symbols which are numbered from 1 to \( n \). An input position \( k \) is said to be on the left (right) of position \( i \) if \( k < i \) (\( k > i \)). An input symbol at position \( i \) is denoted as \( x_i \).

If a phrase in the input string is also in \( D \), we call it a matched phrase or, simply, a match. A matched phrase with length \( l \) starting at input position \( k \) is denoted by \( M(k, l) \). A match that has been selected to be replaced with the corresponding dictionary index is called a selected match, and the position of the first symbol of a selected match is called a break point. A match \( M(k, l) \) is said to be the leftmost match of length \( l \) with respect to (w.r.t.) position \( i \) if there is no match \( M(j, l) \) of length \( l \) for which \( i \leq j < k \). Notice that if there is a match of length \( l \) at position \( i \), it is the leftmost match of length \( l \) w.r.t. position \( i \).

We use \( \text{next}(i) \) to denote the next break point after position \( i \) assuming position \( i \) is a break point. Therefore, if position \( i \) is a true break point, then the phrase \( x_{1} \cdots x_{\text{next}(i)-1} \) will be selected and the position \( \text{next}(i) \) will be a true break point.

3 Optimal Parsing Algorithm with Prefix Dictionaries

In this section, we present our parallel optimal parsing algorithm with prefix dictionaries. We start by reviewing an on-line optimal parsing algorithm with prefix dictionaries.

3.1 On-Line Optimal Parsing Algorithm with Prefix Dictionaries

Hartman and Rodeh presented an on-line optimal parsing algorithm with prefix dictionaries in [5]. DeAgostino and Storer formally proved in [4] that if a dictionary has the prefix property, optimal parsing can be computed on-line without having to look at arbitrarily large prefixes of the input string. The on-line optimal parsing algorithm with prefix dictionaries presented in [5], [4] is shown in Fig. 1.
\( i := 1; \)
\( m(1) := \text{longest match length at position } 1; \)
\( c(1) := 1 + m(1); \)
while \( c(i) \leq n \) do
  begin
    for \( k := i + 1 \) to \( c(i) \) do
      begin
        \( m(k) := \text{longest match length at position } k; \)
        \( c(k) := k + m(k); \)
      end
    determine \( k' \) that satisfies \( c(k') = \max\{c(k) \mid i + 1 \leq k \leq c(i)\}; \)
    encode phrase \( x_i \cdots x_{k'-1}; \)
    \( i := k'; \)
  end
  encode phrase \( x_i \cdots x_n; \)

Fig. 1. On-line optimal parsing algorithm with prefix dictionaries.

The notations are adjusted to this paper. This algorithm runs in \( O(n) \) sequential time with a modified suffix tree data structure [5].

In the on-line algorithm shown in Fig. 1, variable \( i \) is used to keep track of the current break point, \( m(i) \) receives the length of the longest match at position \( i \), and \( c(i) \) receives the index of the position after the longest match at position \( i \) \( (c(i) = i + m(i)) \). At first, we initialize \( i \) and find values for \( m(1) \), and \( c(1) \). In the next while loop, \( i \) advances through the input string from left to right. If \( c(i) \) is found to be greater than \( n \), that means position \( i \) is the last break point in the input string, and the operation terminates after the last phrase \( x_i \cdots x_n \) is encoded.

In each iteration of the while loop, we encode one phrase at the current break point \( i \), and advance \( i \) to the next break point. At the beginning of each iteration of the loop, we know the values of \( i, m(i), \) and \( c(i) \). In the internal for loop, we go through \( m(i) \) positions, from position \( i + 1 \) to the position just after the longest match at position \( i \) (position \( i + m(i) \) or \( c(i) \)). We use variable \( k \) to indicate one of these positions. At each of these positions, we find the longest match length \( (m(k)) \) and the position just after the longest match \( (c(k) = k + m(k)) \). After the for loop, we compare the \( m(i) \) values \( c(i+1), c(i+2), \ldots, c(i) \) and find the maximum among them. In effect, we determine the position \( k' \) at which the longest match has the right-most ending point among the \( m(i) \) longest matches. This position \( k' \) will be the next break point. Thus, we encode the phrase starting from the current break point \( i \) and ending at the position just before the next break point \( k'-1 \) \( (x_i \cdots x_{k'-1}) \), and prepare for the next iteration by assigning \( k' \) to \( i \). During the current iteration of the while loop, we find the longest match length at the break point and the position just after the longest match for the next iteration.

### 3.2 Parallel Optimal Parsing Algorithm with Prefix Dictionaries

Now, we will derive our parallel optimal parsing algorithm with prefix dictionaries from the on-line algorithm in Fig. 1. Since we will use the pointer doubling technique, we have to determine \( next(i) \) for each input position \( i \) in the parallel algorithm [12]. We assign one processor to each input position and make each of them mimic the processor working on one iteration of the while loop in the on-line algorithm in Fig. 1.

We divide our parallel optimal parsing algorithm with prefix dictionaries into three steps. In Step 1, we find the longest match length and the position just after the longest match at each input position. We trace down the suffix tree in \( D \) to find the longest match at each input position \( i \) in parallel and put the length of the longest match in \( m(i) \). This process runs in \( O(L) \) time with \( n \) processors. After that, we find the position just after the longest match at each input position \( i \) \( (i + m(i)) \) in parallel and put the value in \( c(i) \). We assign one extra processor to a dummy input position (position \( n + 1 \) ) and, at this position, we assign \( \infty \) to \( c(n + 1) \) because we will use it in the next step. This process runs in constant time with \( (n + 1) \) processors.

In Step 2, we determine \( next(i) \) for each input position \( i \). At each input position \( i \), we compare \( m(i) \) values \( c(i+1), c(i+2), \ldots, c(i) \) sequentially and find the maximum among them. If \( c(k') \) is the maximum, then \( k' \) is the \( next(i) \). Since \( m(i) \leq L \), we compare at most \( L \) values at each input position. Thus, Step 2 runs in \( O(L) \) time with \( n \) processors.

1. Finding the maximum among, at most, \( L \) values could be completed in \( O(\log L) \) time with \( L / \log L \) processors [12]. Thus, Step 2 could be computed in \( O(\log L) \) time with \( nL/\log L \) processors. However, we retain the complexity of \( O(L) \) time and \( O(n) \) processors because Step 1 requires \( O(L) \) time.
Dictionary entries: a, b, ba, bab, baa, baaa

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| m(i) | 3  | 1  | 4  | 1  | 1  | 1  | 3  | 1  | 3  | 1  | 1  |   |
| c(i) | 4  | 3  | 7  | 5  | 6  | 7  | 10 | 9  | 12 | 11 | 12 |   |
| next(i) | 3  | 3  | 7  | 5  | 6  | 7  | 9  | 9  | 12 | 11 | 12 |   |

break points ● ● ● ● ● ●

Fig. 2. Example of the parallel optimal parsing algorithm with prefix dictionaries.

In Step 3, we determine all the true break points in the input string using a pointer doubling technique. This step requires \(O(\log n)\) time and \(O(n)\) processors.

Fig. 2 shows an example of our parallel optimal parsing algorithm with prefix dictionaries. We have 11 symbols in the input string, and position 12 is the dummy input position. A circle or two circles connected with a line show the longest match at each input position. The two rows with the marks \(m(i)\) and \(c(i)\) show the results of Step 1. For example, at position 7, the longest match is “bab” with length 3 and the position just after this longest match is position 10. Thus, \(m(7) = 3\) and \(c(7) = 10\). The row marked with \(\text{next}(i)\) in the figure shows the result of Step 2. For example, at position 7, we compare three values \(c(8), c(9),\) and \(c(10)\) sequentially because \(m(7) = 3\). The maximum among them is \(c(9) = 12\) at position 9. Therefore, we determine that \(\text{next}(7) = 9\). The filled circles in the bottom row show the true break points determined in Step 3. Position 1 is always a true break point. Positions 3, 7, and 9 are also determined to be the true break points because \(\text{next}(1) = 3,\) \(\text{next}(3) = 7,\) and \(\text{next}(7) = 9\).

Overall, our algorithm requires \(O(L + \log n)\) time and \(O(n)\) processors. Under the assumption that \(L\) is equal to \(O(\log n)\), the runtime becomes \(O(\log n)\).

## 4 LFF Parsing Algorithm

In this section, we present our parallel LFF parsing algorithms. We start with the description of the sequential LFF parsing algorithm and its on-line version. Then, we give an \(O(L + \log n)\) time and \(O(nL)\) processor parallel LFF parsing algorithm. After that, we improve the processor requirement to \(O(n\log L)\).

### 4.1 Sequential LFF Parsing Algorithm

The first LFF parsing algorithm was originally developed for database compression by Schuegraf and Heaps [6], and the first parallel LFF parsing algorithm was presented by Stauffer and Hirschberg [10]. We quote Schuegraf and Heaps’ description of their LFF parsing algorithm from [6]:

All fragments contained in the record must first be located and stored. After all fragments of the record have been stored the LFF parsing algorithm recursively removes the longest fragment and deletes all the subfragments which overlap the one removed.

This algorithm can be divided into two steps. In the first step, the input string is scanned and all matches are stored. In the second step, the longest match in the uncoded portion of the input string is recursively selected, and the matches that overlap the selected one are eliminated. We call this process in the second step selection and elimination process.

We need a more detailed algorithm to start with because, from the above description, it is not clear how the matches are stored and what happens if more than one longest matches overlap each other. To describe the details of LFF parsing algorithm, we follow the notations and strategies used in Stauffer and Hirschberg’s paper [10]. In their paper, a match list is first computed at each input position to store the fragments. That is, an element \(m(k, l)\) of a two-dimensional array \(m(i, j)\) for which \(1 \leq i \leq n, 1 \leq j \leq L\) is set to 1 if there is a match of length \(l\) at position \(k\), otherwise \((k, l)\) remains 0. In addition, variable \(a(i)\) is employed at each input position \(i\) and used to denote the longest match permitted at position \(i\). This variable can be used to eliminate matches that are longer than a certain length at position \(i\). For example, the matches that are longer than \(l\) are eliminated at position \(i\) by setting \(a(i) = l - 1\). Also, in [10], when \(l\) is the length of the longest matches, a maximum collection \(C\) of nonoverlapping matches of length \(l\) is found. This way of selecting the longest matches is reasonable because we do not want to replace one longest fragment with more than one shorter fragments. The maximum collection \(C\) is found by selecting matches of length \(l\) from left to right.

We describe a detailed sequential LFF parsing algorithm in Fig. 3. In Step 1, we compute match lists \(m(i, j)\) as in [10]. We can compute a match list \(m(k, j)\) for \(1 \leq j \leq L\) at position \(k\) in \(O(L)\) time by tracing down the suffix tree. Therefore, Step 1 requires \(O(nL)\) sequential time.

In Step 2, we iteratively compute the selection and elimination process, starting from the longest match.
length $L$ and going down to the shortest match length 1 in the for loop. Variable $l$ keeps track of the current longest match length. In the while loop, we go through the input string from left to right and select matches of length $l$. In each iteration of the while loop, we check if there is a match of length $l$ at position $i$ ($M(i, l)$). If there is, we select it and mark the input positions $i, \ldots, i + l - 1$, over which the selected match extends as selected. We can mark these positions by using variable $a(i)$ for each input position $i$ and assigning 0 to $a(i)$ if position $i$ is in the range of a selected match ($a(i)$ for $1 \leq i \leq n$ must be initialized to $L$ at the beginning of Step 2). This will eliminate all matches at positions that are in the range of the selected match. Also, we eliminate the matches that overlap the selected one in the $l - 1$ preceding positions $i - l + 1, \ldots, i - 1$. We can carry out this operation by assigning $i - k$ to $a(k)$ for $i - l + 1 \leq k \leq i - 1$ as in [10]. This will essentially eliminate all the matches longer than $i - k$ from the match list at position $k$. After the elimination of the matches, or when we cannot find a match of length $l$ at position $i$, we advance variable $i$ to the right until we reach the next unselected position. In an iteration of the while loop in Step 2, when $M(i, l)$ is selected, we have to modify $(2l - 1) - a(k)$s, but we select at most $n/l$ matches. Therefore, each iteration of the while loop runs in $O(n)$ time, and Step 2 requires $O(nL)$ sequential time. Overall, the sequential LFF parsing algorithm requires $O(nL)$ time.

4.2 On-Line LFF Parsing Algorithm

The LFF parsing algorithm shown in Fig. 3 is not an on-line algorithm because we have to go through the input string in each iteration of the for loop in Step 2. We will modify the LFF parsing algorithm to an on-line version in preparation for designing parallel LFF parsing algorithms with the pointer doubling technique. At first, we will show three lemmas that are necessary for the modification.

To simplify the description of the lemmas, we define some terms. If a match $M(k, l)$ is eliminated after another match $M(k_a, l_a)$ is selected, we say $M(k_a, l_a)$ eliminates $M(k, l)$. In Step 2 in Fig. 3, $M(k_a, l_a)$ may eliminate $M(k, l)$ if $l_a > l_b$ and they overlap $(k_a - l_b + 1 \leq k_b \leq k_a + l_a - 1)$, or if $l_a = l_b$ and $l_a < l_b$, and they overlap $(k_b \leq k_a + l_a - 1)$. A match is selected if it is not eliminated until it becomes the left-most one among the longest matches that have been neither selected nor eliminated. We call the matches that have been neither selected nor eliminated the surviving matches. Eventually, a match $M(k_a, l_a)$ will be eliminated or selected. However, if the result would be different when assuming that another match $M(k_b, l_b)$ did not exist, we say $M(k_a, l_a)$ affects $M(k_b, l_b)$. If a match at an input position is eliminated, it is as though that the match did not take place at this particular position. Thus, a match that will be eliminated cannot affect any other match. Also, the influence of a match starts when it is selected. Thus, a match cannot affect any match that is longer than itself because the longer match has been selected earlier.

Let us consider the LFF parsing algorithm in Fig. 3. Suppose that, on a particular iteration of the while loop, a match $M(k, l)$ has been selected and the matches of length less than or equal to $l$ that overlap $M(k, l)$ have been eliminated. Then, the matches on the left of $M(k, l)$ and those on the right are isolated, and a further parsing of the left side is not affected by the further parsing of the right. This is stated in the following lemma.

**Lemma 1.** If a match $M(k, l)$ has been selected in a process of the LFF parsing, no match $M(k_1, l_1)$ for which $l_1 < l$ and $k_1 + l_1 - 1 < k$ will be affected by any match $M(k_r, l_r)$ for which $l_r \leq l$ and $k_r > k$.

Suppose that there are $q$ matches $M(i, l_1), M(i, l_2), \ldots, M(i, l_q)$ at position $i$ and that position $i$ is a break point, then one of them will be selected. Using Lemma 1, we prove that only the left-most match w.r.t. position $i$ for each match length can affect these matches at position $i$. This means that, if position $i$ is a break point, we need to check only the left-most match w.r.t. position $i$ for each match length to compute the next break point. This is formally stated in the next lemma.

---

**Fig. 3.** Sequential LFF parsing algorithm.
Lemma 2. Suppose that input position \( i \) is a break point. A match \( M(k, l) \) will affect the matches at position \( i \) only if \( M(k, l) \) is the left-most match of length \( l \) w.r.t. position \( i \).

The next lemma states that we do not need to search for the left-most matches for arbitrarily long distances, but it is enough if we look for a match of length \( l \) up to \( \frac{1}{2}(l - 1)(l - 2) \) positions to the right.

Lemma 3. Suppose that input position \( i \) is a break point and that \( M'(k, l) \) is the left-most match of length \( l \) w.r.t. position \( i \). If \( k_i - i > \frac{1}{2}(l - 1)(l - 2) \), then \( M'(k, l) \) will not affect any match at position \( i \).

A more general description of Lemma 3 can be given as a relation between two matches as follows: A match \( M(k_a, l_a) \) will not affect another match \( M(k_b, l_b) \) for which \( l_b > l_a \) if \( k_b - k_a > \frac{1}{2}(l_b - l_a)(l_a + l_b - 3) \).

Since the longest match length is \( L \), Lemma 3 means that no match which is more than a distance of \( \frac{1}{2}(L - 1)(L - 2) \) away from position \( i \) will affect the matches at position \( i \).

Now, we will modify the LFF parsing algorithm in Fig. 3 so that the match lists are generated only for the first \( \frac{1}{2}(L - 1)(L - 2) + 1 \) positions from position \( i \). We also find the positions of the left-most matches w.r.t. position \( i \) in this step. We use the distance \( d_l \) from position \( i \) to specify the position of the left-most match of length \( l \). If \( d_l \) is greater than \( \frac{1}{2}(l - 1)(l - 2) \), we simply assign \( \infty \) to it to indicate that this left-most match will not affect the matches at position \( i \).

We then modify Step 2 in Fig. 3 so that only the left-most matches w.r.t. position \( i \) participate in the selection and elimination process as shown in Step 2 in Fig. 4. We go through the left-most matches one by one from the longest one toward the shortest one until a selected match is found at position \( i \). For each left-most match, we check if it has been eliminated by the longer selected matches. If it has not, then it is selected and eliminates the shorter left-most matches that overlap it. If the left-most match of length \( l \) is eliminated, \( \infty \) is assigned to \( d_l \).

The while loop in Fig. 4 starts from the longest match length \( L \) and, in each iteration, the length is decremented. On a particular iteration, we check the distance \( d_l \) of the left-most match of length \( l \). If \( d_l = 0 \), the match of length \( l \) is at position \( i \) and has not been eliminated. Thus, the match of length \( l \) at position \( i \) is encoded, and the while loop stops here. If \( 0 < d_l < \infty \), then the match of length \( l \) at position \( i + d_l \) is selected. Therefore, in the next for loop, the left-most matches of length less than \( l \) that overlap or that are on the right of the selected match of length \( l \) are eliminated. We eliminate the left-most matches that are on the right of the selected match as well because, by Lemma 1, they will not affect the matches at position \( i \). The while loop will eventually terminate because \( d_l = 0 \) by assumption.

4.3 Parallel LFF Parsing Algorithm

Now, we will parallelize the on-line LFF parsing algorithm shown in Fig. 4. Again, we determine \( \text{next}(i) \) for each input position \( i \) and use the pointer doubling technique to mark
all the true break points in the input string. Our aim is to make the runtime of the parallel LFF parsing algorithm $O(L + \log n)$. We first present a parallel algorithm that requires $O(nL)$ processors.

The parallel LFF parsing algorithm is shown in Fig. 5. When we parallelize Step 1 of the on-line algorithm in Fig. 4, we separate it into two steps, Step 1 and Step 2 in Fig. 5. Step 3 in Fig. 5 is the parallelization of Step 2 in Fig. 4.

In Step 1, we compute the match lists $m(i, j)$. We assign one processor to each input position to carry out this operation. At each input position $i$, we find all matches that start from position $i$ by tracing down the suffix tree $D$. This step runs in $O(L)$ time with $O(n)$ processors.

In Step 2, we compute distance $d(i, j)$ of the left-most match of length $j$ w.r.t. position $i$ from position $i$ for $1 \leq i \leq n$ and $1 \leq j \leq L$. To compute $d(i, j)$, we first find the position $p(i, j)$ of the left-most match of length $j$ w.r.t. position $i$. We use the pointer doubling technique to carry out this operation. Then, we compute $d(i, j)$ by subtracting $i$ from $p(i, j)$.

At the beginning of Step 2, we initialize $p(i, j)$ for $1 \leq i \leq n+1$ and $1 \leq j \leq L$. For any input position $i$ with $i < n+1$, if there is a match of length $j$ at position $i$, we initialize $p(i, j)$ to $i$ because the left-most match of length $j$ w.r.t. position $i$ is at position $i$. Otherwise, we initialize $p(i, j)$ to 0 to indicate that there is no match at position $i$. We also use $p(i, j)$ for $1 \leq j \leq L$ at the extra input position $n+1$, and we initialize them to $\infty$ to indicate that there is no match after the last input position. After that, we initialize the pointers. We initialize pointer $P(i)$ to $i+1$ for each input position $i$ if $i < n+1$. We use one more pointer $P(n+1)$ and use it as the last pointer that points to itself.

In the next repeat loop, for each match length $j$, we propagate the position index of each match to the left using the pointer doubling technique. For this loop, we assign $L$ processors to each input position $i$. Each processor takes care of one element of the array $p(i, j)$.

In each iteration, $p(i, j)$ receives $p(P(i), j)$ if $p(i, j)$ does not have a finite value, but $p(i, j)$ will not be changed if it already has a finite value. Therefore, if a match $M(k, l)$ for which $k < k$ is the first match of length $l$ on the left of a match $M(k, l)$, the position index $(k)$ of $M(k, l)$ will not propagate to the left of $M(k, l)$. Since we do not have to find a distance $d(i, j)$ if it is greater than $\frac{1}{2}(L-1)(L-2)$, we repeat the body of the repeat loop only $\left[\log_2(L-1)(L-2)+1\right]$ times. With $O(L)$ processors, we can initialize $p(i, j)$ in constant time, compute $p(i, j)$ in $O(\log L)$ time, and compute $d(i, j)$ from $p(i, j)$ in constant time. Also, we can initialize the pointers in constant time using $O(n)$ processors. Thus, Step 2 runs in $O(\log L)$ time with $O(nL)$ processors.

In Step 3, we perform the selection and elimination process and determine $next(i)$ for each input position $i$. This is a direct parallelization of Step 2 in Fig. 4, and processors assigned to each input position mimic the processor working on Step 2 in Fig. 4. However, we use the for loop in Fig. 5 (the second for loop) and repeat the body of the loop $L$ times, instead of the while loop in Fig. 4 because, if we used the while loop and repeated the body of the loop until a match was selected at every input position, it would have taken more than constant time to determine whether a match is selected at every input position with the CREW assumption. Since we cannot stop the loop when a match is selected at position $i$ (when $d(i, l) = 0$) using the for loop, we employ variable $length(i)$ at each input position $i$ to record the length $l$ when $d(i, l) = 0$. We initialize $length(i)$ for $1 \leq i \leq n$ to $\infty$ at the beginning of Step 3. Once a left-most match of length $l$ w.r.t. position $i$ is found to be selected ($d(i, l) < \infty$), we carry out the elimination process in parallel in the second for loop. We assign one processor to each element of the array $d(i, j)$ so that the elimination process can be carried out in constant time. Thus, Step 3 runs in $O(L)$ time with $O(nL)$ processors.

In Step 4, we use the pointer doubling technique to determine all the true break points in the input string. This step requires $O(\log n)$ time and $O(n)$ processors.

Over all, this LFF parsing algorithm requires $O(L + \log n)$ time and $O(nL)$ processors. With the assumption that $L$ is equal to $O(\log n)$, it requires $O(\log n)$ time and $O(n \log n)$ processors.

### 4.4 Improvements on the Parallel LFF Parsing Algorithm

The number of processors required for the parallel LFF parsing algorithm in Fig. 5 is dominated by Step 2 and Step 3. Therefore, we try to improve the number of processors required in these two steps.

At first, we look at Step 2 in Fig. 5. This step requires $O(\log L)$ time and $O(nL)$ processors. We can increase this runtime to $O(L)$ by decreasing the number of processors because Step 1 requires $O(L)$ time. In the repeat loop in Step 2, the pointer doubling is performed for all $L$ match lengths in parallel. We can partition the $L$ match lengths into $\left[\log L\right]$ groups so that each group contains at most $\left[\log L\right]$ different match lengths, and perform the pointer doubling for each group in parallel.

The modified Step 2 is shown in Fig. 6. In this figure, the repeat loop is now nested in a for loop (the third for loop), and the initialization of pointers ($P(i)$) is also carried out in this for loop. We iterate the body of this for loop $\left[\log L\right]$ times, and, in each iteration, we compute $p(i, j)$ for $\left[\log L\right]$ different match lengths using the pointer doubling technique. With this modification, each iteration of this for loop requires $O(\log L)$ time, but we iterate the body of the for loop $O(L/\log L)$ times. Thus, this for loop requires $O(L)$ time and $O(n \log L)$ processors.

In Fig. 6, we also modified the part to initialize the array $p(i, j)$ (the first for loop) and the part to compute the array $d(i, j)$ (the second for loop from the bottom) so that they require $O(L)$ time and $O(n)$ processors. Thus, the modified Step 2 requires $O(L)$ time and $O(n \log L)$ processors.

Next, we look at Step 3 in Fig. 5, the selection and elimination process. The problem in this step is that the third for loop in this step that invokes $n$ operations in parallel has a nested for loop (the second for loop from the bottom) that invokes at most $L - 1$ operations in parallel. If
Step 3 is to be computed in $O(L)$ time, the internal for loop has to be computed in constant time. This is the reason why we have to employ more than one processors at each input position. Thus, we try to eliminate this nested for loop.

To determine if a left-most match of length $l$ w.r.t. position $i$ ($M^i(k_l, l)$) is eliminated in Step 3, we check the condition $d(i, l) < \infty$. However, we can also determine this by going through the left-most matches w.r.t. position $i$ that are longer than $l$, namely

$$M^i(k_{i+1}, l + 1), M^i(k_{i+2}, l + 2), \ldots, M^i(k_L, L),$$

and by checking if any of them would eliminate $M^i(k_l, l)$. The next lemma shows that, to determine if $M^i(k_l, l)$ is eliminated by any of the longer left-most matches, we need to check if $M^i(k_l, l)$ would be eliminated by only one left-most match: the shortest match among the selected left-most matches that are longer than $l$ (the shortest selected match).

**Lemma 4.** Suppose that we go through the left-most matches w.r.t. position $i$ ($M^i(k_l, l)$) in this order to determine whether each of them is eliminated or selected, in Step 3 of Fig. 5. Suppose also that, when we check a left-most match $M^i(k_l, l)$, the match $M^i(k_a, a)$ for which $a > l$ is the shortest selected match. If $M^i(k_l, l)$ is eliminated by a selected match $M^i(k_b, b)$ for which $b > a$, then $M^i(k_l, l)$ is also eliminated by $M^i(k_a, a)$.

We modify Step 3 using Lemma 4 so that each left-most match is visited only once, instead of checking all the left-most matches of length less than $l$ after the match of length $l$ is selected. The modified Step 3 is shown in Fig. 7. We use variable $s(i)$ to keep the position of the shortest selected match for each input position $i$. For each match length $l$, we determine whether the left-most match of length $l - 1$ is eliminated by the shortest selected match by checking if $d(i, l - 1) + (l - 1) - 1 \geq s(i)$. If it is not eliminated, it becomes the new shortest selected match and $d(i, l - 1)$ is assigned to $s(i)$.

With one processor in each input position, the modified Step 3 can run in $O(L)$ time. Thus, it requires $O(L)$ time and $O(n)$ processors.

With the modified Step 2 and Step 3, our parallel LFF parsing algorithm requires $O(L \log n)$ time and $O(n \log L)$ processors. With the assumption that $L$ is equal to $O(\log n)$, it requires $O(\log n)$ time and $O(n \log \log n)$ processors.
5 CONCLUSIONS

The complexities of parallel parsing algorithms for the greedy parsing, the optimal parsing with prefix dictionaries, and the LFF parsing strategies are summarized in Table 1. We have improved the runtime or number of processors required for the parallel optimal parsing algorithm with prefix dictionaries. We have also improved the runtime of parallel LFF parsing algorithm over Stauffer and Hirschberg's algorithm in [10], but paid some penalty in the number of processors required.

The key to our parallel parsing algorithms is the pointer doubling technique. If we can run a sequential parsing algorithm on-line, in the corresponding parallel parsing algorithms, we can compute next break point next(i) for each input position i, assuming position i is a break point, and use the pointer doubling technique to determine all the true break points.

APPENDIX

PROOFS OF LEMMAS

In this Appendix, we prove Lemma 1, Lemma 2, Lemma 3, and Lemma 4.

Proof of Lemma 1
Proof. Suppose a match \( M(k, l) \) has been selected and the matches of length less than or equal to \( l \) that overlap \( M(k, l) \) have been eliminated. Any surviving match \( M(k_l, l) \) on the left of position \( k \) satisfies the condition \( l_i < l \) and \( k_i + l_i - 1 < k \), and any surviving match \( M(k_r, l) \) at position \( k \) or on the right of position \( k \)

2. for \( j := 1 \) to \( L \) do
   for all \( i, 1 \leq i \leq n + 1 \), in parallel do
      if \( m(i, j) = 1 \) then \( p(i, j) := i \); else \( p(i, j) := \infty \);
   for \( k := 0 \) to \( \lceil L/\log L \rceil - 1 \) do
      begin
      for all \( i, 1 \leq i \leq n + 1 \), in parallel do
         if \( i < n + 1 \) then \( P(i) := i + 1 \); else \( P(i) := i \);
      repeat \( \lceil \log(\frac{1}{2}(L - 1)(L - 2) + 1) \rceil \) times
         for all \( i, 1 \leq i \leq n + 1 \), in parallel do
            begin
            for all \( j, k \lceil \log L \rceil \leq j \leq (k + 1)\lceil \log L \rceil - 1 \), in parallel do
               if \( p(i, j) = \infty \) then \( p(i, j) := p(P(i), j) \);
               \( P(i) := P(P(i)) \);
            end
            end
      for \( j := 1 \) to \( L \) do
         for all \( i, 1 \leq i \leq n + 1 \), in parallel do \( d(i, j) := p(i, j) - i \);

Fig. 6. Modified Step 2 of the parallel LFF parsing algorithm.

3. for all \( i, 1 \leq i \leq n \), in parallel do
   begin
   \( \text{length}(i) := \infty \);
   \( s(i) := d(i, L) \);
   end
   for \( l := L \) down to 1 do
   for all \( i, 1 \leq i \leq n \), in parallel do
      if \( d(i, l) = 0 \) then
         if \( \text{length}(i) = \infty \) then \( \text{length}(i) := l \);
         else
            if \( d(i, l - 1) + (l - 1) - 1 > s(i) \) then \( d(i, l - 1) := \infty \);
            else \( s(i) := d(i, l - 1) \);
      for all \( i, 1 \leq i \leq n \), in parallel do \( \text{next}(i) := i + \text{length}(i) \);

Fig. 7. Modified Step 3 of the parallel LFF parsing algorithm.
Proof of Lemma 2

Proof. We prove this lemma by exhausting all the possible cases. Suppose that $M(k, l)$ is not the left-most match of length $l$ w.r.t. position $i$ and that the left-most match of length $l$ is $M(j, l)$ ($i < j < k$).

Case 1: When $M(k, l)$ overlaps $M(j, l)$ ($j < k < j + l - 1$)

1. If $M(j, l)$ is not eliminated by any longer match, then $M(k, l)$ will be eliminated by $M(j, l)$ and will not affect any match at position $i$.
2. If $M(j, l)$ is eliminated by $M(g, h)$ for which $h > l$ and $g > j$, then $M(k, l)$ will also be eliminated by $M(g, h)$ and will not affect any match at position $i$.
3. If $M(j, l)$ is eliminated by $M(g, h)$ for which $h > l$ and $g < j$, then, by Lemma 1, $M(k, l)$ will not affect any match at position $i$ because $l < h$ and $k > g$.

Case 2: When $M(k, l)$ does not overlap $M(j, l)$ ($j + l - 1 < k$)

1. If $M(j, l)$ is not eliminated by any longer match, then $M(j, l)$ is selected and, by Lemma 1, $M(k, l)$ will not affect any match at position $i$.
2. If $M(j, l)$ is eliminated by a longer match $M(g, h)$ for which $h > l$ and $g - 1 + j < g - h + l$, then, by Lemma 1, $M(k, l)$ will not affect any match at position $i$ because $l < h$ and $k > g$.

Proof of Lemma 4

Proof. Suppose that $l < a < b$ and that both $M^*(k_a, a)$ and $M^*(k_b, b)$ are selected. We need to prove that if $M^*(k_b, b)$ eliminates $M^*(k_i, l)$, then $M^*(k_a, a)$ would also eliminate $M^*(k_i, l)$. We have $k_a + a - 1 < k_b$ because both $M^*(k_a, a)$ and $M^*(k_b, b)$ are selected. We also have $k_i + l - 1 > k_b$ because $M^*(k_b, b)$ would eliminate $M^*(k_i, l)$. Combining the two, we get $k_i + l - 1 > k_a + a - 1$. Considering $k_i < k_a$, we get $k_i + l - 1 > k_a$. Thus, $M^*(k_a, a)$ would eliminate $M^*(k_i, l)$.

References


Hideo Nagumo received the MS and PhD degrees in electrical engineering from Texas A&M University, College Station, in 1991 and 1996, respectively. He joined the faculty of Niigata Seiyo Women’s Junior College in 1997 and is currently an associate professor there. His research interests include parallel algorithms, data compression, VLSI algorithms, and computer arithmetic. He is a member of the IEEE.

Mi Lu received the MS and PhD degrees in electrical engineering from Rice University, Houston, in 1984 and 1987, respectively. She joined the Department of Electrical Engineering at Texas A&M University in 1987, and she is currently a professor. Her research interests include parallel computing, distributed processing, parallel computer architectures and algorithms, computer arithmetic, computer network, computational geometry, and VLSI algorithms. She has published over 70 technical papers in these areas. She has served as an associate editor of the Journal of Computing and Information and the Information Sciences Journal. She was the stream chair of the Seventh International Conference on Computing and Information and will be the conference chair of the Fifth International Conference on Computer Science and Informatics; served on the panels of the National Science Foundation and the 1992 IEEE Workshop on Imprecise and Approximate Computation; served on the many conference program committees; is chair of 36 research advisory committees for PhD and masters students; is a registered professional engineer; and is a senior member of the IEEE and a regular member of the IEEE Computer Society.

Karan L. Watson received her BS, MS, and PhD degrees in electrical engineering at Texas Tech University in 1977, 1981, and 1982, respectively. She joined Texas A&M University in 1983 and is now the associate dean of engineering and a professor of electrical engineering. She is also the director of the Texas AMP and part of the management team for the Foundation Coalition. Her experience includes time with Hicks & Ragland Engineering, Lubbock, Texas, and AT&T Long Lines, Houston. She has been the advisor of 24 PhD students and 61 masters students. Among her national honors are the IEEE fellow grade; the American Association for the Advancement of Science Mentor Award (1998); the U.S. President’s Award for Excellence in Science and Technology Mentoring (1997); the American Society of Engineering Educators Minorities in Engineering Award (1997); the Harriett B. Regis Award, Hewlett-Packard Company (1996); and the IEEE Undergraduate Teaching Medal, Board of Directors of the IEEE (1996). Some of her national service includes the IEEE Education Society president (1999) and vice president (1996-1998); the Committee on Engineering Accreditation Activity Board (1995-2000); ABET Visitor (1993-2003); the Liaison Committee chair (1999), and the Awards Committee (1997-1999); and the Education Activity Board Curriculum Committee (1992),