Parallel Algorithms for The Static Dictionary Compression

Hideo Nagumo, Mi Lu, and Karan Watson
Department of Electrical Engineering, Texas A&M University
College Station, TX 77843-3128

Abstract We study parallel algorithms for two static dictionary compression strategies. One is the optimal dictionary compression with dictionaries that have the prefix property, for which our algorithm requires \(O(L + \log n)\) time and \(O(n)\) processors where \(L\) is the maximum allowable length of the dictionary entries, while previous results run in \(O(L + \log n)\) time using \(O(n^2)\) processors, or in \(O(L + \log^2 n)\) time using \(O(n)\) processors. The other is the longest fragment first (LFF) dictionary compression, for which our algorithm requires \(O(L + \log n)\) time and \(O(nL)\) processors, while previous result has \(O(L \log n)\) time performance on \(O(n / \log n)\) processors. We also show that the sequential LFF dictionary compression can be computed on-line with the lookahead of length \(O(L^2)\).

1 Introduction

The dictionary type data compression works by replacing phrases in the input string with the indexes into some dictionary \([I]\). The dictionary can be static or dynamic. In static dictionary data compression, the dictionary contains a predetermined fixed set of entries. In dynamic dictionary compression, the dictionary changes its entries during compression.

In the dictionary type data compression, the input string is split into phrases that are also in the dictionary and are to be replaced with dictionary indexes. This process is called parsing and there are several different strategies for this. The most popular one in sequential environment is greedy parsing, where the input string is processed from left to right and at each step the longest match starting at the first symbol of the uncoded portion of the string is encoded by replacing it with a dictionary index. Greedy parsing is simple but in general is not optimal. Optimal parsing strategy is the one that produces the compressed form of minimum length. If fixed length dictionary indexes are used, the optimal parsing is the one that produces 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, sequential optimal parsing can be computed on-line with limited length of lookahead. A dictionary has the prefix property if all the prefixes of a phrase in the dictionary are also in the dictionary \([2, 3]\). The parsing strategy whose compression efficiency is somewhat in between greedy and optimal parsing is the longest fragment first (LFF) parsing \([4]\). This approach parses the input string by repeatedly locating the longest phrase in the uncoded portion of the input that is also in the dictionary, and replacing it by the corresponding dictionary index. It has been believed that the sequential LFF parsing encoder must be able to look at arbitrarily large prefixes of the input string \([4, 5]\).

In this paper, we consider parallel 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. A number of parallel algorithms for string problems have been designed on PRAM [6]. In addition, there are several papers that describe parallel algorithms for dictionary type data compression on PRAM [2, 5, 7]. There are several variations of PRAM and the difference comes from the assumptions on simultaneous reading and writing of the shared memory. Our algorithms in this paper use 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 [8].

In this paper, we study two static dictionary compression strategies, namely optimal parsing with dictionaries that have the prefix property, and the LFF parsing. Agostino and Storer have presented two parallel algorithms for the former 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 $L$ is the maximum allowable length of the dictionary entries [2]. We present a parallel algorithm that requires $O(L + \log n)$ time and $O(n)$ processors. A parallel algorithm for LFF parsing has been presented by Hirschberg and Stauffer with the complexity of $O(L \log n)$ time and $O(n/\log n)$ processors. We present an algorithm for this problem that requires $O(L + \log n)$ time and $O(nL)$ processors. We also show that the sequential LFF parsing can be computed on-line with the lookahead of length $O(L^2)$.

Throughout this paper, we assume that: 1. the input string is processed in blocks with each having $n$ symbols, 2. a dictionary $D$ includes all members of the input symbol set [5], 3. the length of the entries of $D$ is less than or equal to some maximum length $L$, and $L$ is of $O(\log n)$. The last assumption is commonly used and justified in [2, 5].

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 an input position $k$ is denoted by $Q(k,l)$. An input position $k$ is said to be on the left (right) side of position $i$ if $k < i$ ($k > i$). A phrase that has been selected to be replaced with an index into $D$ is called a selected phrase, and the position of the first symbol of a selected phrase is called a break point. When we say "next(i) is the next break point of position $i$", this means if position $i$ is a break point then next(i) is the next break point.

2 Optimal Parsing Algorithm

Hartman and Rodeh have shown 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. Their on-line optimal parsing algorithm can be implemented in $O(n)$ sequential time with a modified suffix tree data structure [3]. The parallelization of the on-line optimal parsing has been described by Agostino and Storer [2]. Their $O(L + \log n)$ time and $O(n^2)$ processor algorithm is shown in Fig. 1.

Their algorithm computes the break points by constructing an $(n + 2) \times n$ matrix $M$. Each column $i$ corresponds to an input position $i$, and after completion of the matrix, column $i$ contains a sequence of break points that follow position $i$ (each
in parallel for \(1 \leq i \leq n\) do
begin
1. \(m(i) := \) longest match length at position \(i\);
2. \(M(1, i) = M(3, i) := i + m(i); M(2, i) := i; f(i) := 3;\)
3. while \(M(f(i), i) \neq n + 1\) do

   let \(l(i)\) be such that \(M(1, l(i)) = \max(M(1, k) : M(f(i) - 1, i) + 1 \leq k \leq M(f(i), i))\)

   in parallel for \(2 \leq j \leq n + 2\) do

      if \(j \leq f(l(i))\) then \(M(f(i) + j - 2, i) := M(j, l(i)); f(i) := f(i) + f(l(i)) - 2;\)
end

Figure 1: Agostino and Storer's optimal parsing algorithm.

column \(i\) assumes that position \(i\) is a break point). Since the input position 1 is a definitive break point, the first column will provide the true break points. At first, their algorithm finds the longest match at each input position \(i\) and puts it to variable \(m(i)\) in step 1. This can be done in \(O(L)\) time with \(n\) processors using suffix tree data structure in \(D\). In step 2, the first three rows of the matrix and the variable \(f(i)\) are initialized. The first and third rows of column \(i\) are initialized to the position just after the longest match at position \(i\) (i.e., \(i + m(i)\)). The second row of column \(i\) is initialized to \(i\). The rest are assumed to be 0. The variable \(f(i)\) for column \(i\) keeps track of the number of the last non-null row, and it is initialized to 3. Step 2 is done in constant time. Step 3 is the main loop to construct the matrix. At the beginning of a particular iteration, a variable \(l(i)\) is computed for each column \(i\) using \(M(f(i), i)\) and \(M(f(i) - 1, i)\). \(M(f(i), i)\) and \(M(f(i) - 1, i)\) are the first and second non-null values from the bottom in column \(i\) in the iteration. These two values must have occurred as third and second values in a column, say column \(k\) for which \(i \leq k\), after the initialization step. Because of step 2, \(M(f(i) - 1, i) = k\) and \(M(f(i), i) = k + m(k)\). Thus \(M(1, l(i))\) is a maximum among \(k + 1 + m(k + 1), k + 2 + m(k + 2), \ldots, k + m(k) + m(k + m(k))\). According to the on-line optimal algorithm shown in [2], if \(k\) is a break point, \(k, k + 1, \ldots, l(i) - 1\) is a definitive phrase of the parsing and \(l(i)\) is the next break point. Thus \(l(i)\) is the next break point after \(M(f(i) - 1, i)\), the second non-null value from the bottom of column \(i\). Once \(l(i)\) is found, non-null part of column \(l(i)\), excluding the first row, is copied after the second non-null value in the column \(i\). In each iteration, the number of completed rows is doubled. Thus step 3 requires \(O(\log n)\) iterations. To complete step 3 in \(O(\log n)\) time, the maximum finding and the copying have to be done in constant time, and this requires \(O(n^2)\) processors.

We use this algorithm as a starting point of ours. Note that in step 3 of this algorithm, if \(k\) is known, the computation of \(l(i)\) can be independent of \(i\). This means that the next break point \(next(k)\) of a position \(k\) can be computed before the loop starts at step 3. Thus if we place \(next(i)\) instead of \(i + m(i)\) in the third row of the matrix in step 2, we do not need to compute \(l(i)\) in each iteration of the loop. In fact,
once we calculated next(i) for a position i, we do not even need to construct a matrix. We can reduce the parsing problem to a path finding problem on a directed graph, and this can be solved by initializing position 1 to be a definitive break point and using the standard pointer doubling technique [8]. Since the computation of next(i) is done only once, this can be done in a sequential manner. This computation involves a maximum finding of at most L elements because m(i) is limited to L, and requires O(L) time.

Our algorithm for optimal parsing with dictionaries that have the prefix property is shown in Fig. 2. In step 1, we find the longest match length m(i) and computes

\begin{verbatim}
 for 1 ≤ i ≤ n + 1 do
   begin
   1. if i < n + 1 then m(i) := longest match length at position i else do nothing;
   2. if i < n + 1 then c(i) := i + m(i) else c(i) := ∞;
   end

   if i = 1 then mark(i) := 1 else mark(i) := 0;
   P(i) := next(i);
   while mark(n + 1) = 0 do
     begin
       if mark(i) = 1 then mark(P(i)) := 1;
       P(i) := P(P(i));
     end
   end
\end{verbatim}

Figure 2: The O(L + log n) time O(n) processor optimal parsing algorithm.

i + m(i) at each input position i. The variable c(i) receives the value i + m(i). In step 2, we find the next break point next(i) of position i similar to the second line of step 3 in the algorithm in Fig. 1. The values c(i + 1), c(i + 2), ..., c(i + m(i)) are compared sequentially to find the maximum among them, and the position that gives the maximum is the next(i). Since m(i) ≤ L, at most L values are compared. In step 3, we find all the break points in the input string using the standard pointer doubling technique. At first, position 1 is marked as a break point because that is always true, and next(i) is copied to P(i) so that P(i) can be used as a pointer. In each iteration of the loop, the position pointed to by position i is marked as a break point if position i has been marked, and the pointer is doubled. The iteration ends when position n + 1 is marked as a break point.

Step 1 requires O(L) time using suffix tree data structure and n + 1 processors. Step 2 also requires O(L) time and n + 1 processors. Step 3 requires O(log n) time and n + 1 processors. Overall, the algorithm in Fig. 2 requires O(L + log n) time and O(n) processors. Using the assumption that L is in the O(log n), the time complexity becomes O(log n).
1. Compute match lists at each input position
2. Compute maximum match length, $L^*$
3. For each match length $l := L^*$ down to 1 do
   a. Find a maximum collection $C$ of non-overlapping matches of length $l$
   b. Update match lengths at positions overlapping $C$

Figure 3: Stauffer and Hirschberg's LFF parsing algorithm.

3 LFF Parsing Algorithm

The PRAM algorithm for LFF dictionary compression has been studied by Stauffer and Hirschberg [5]. Their algorithm, in a high level description, is shown in Fig. 3. They compute match lists at each input position in step 1. Since the length of a dictionary entry is limited to $L$ there are at most $L$ matches at an input position. They use an array $m(i, j)$ ($1 \leq i \leq n, 1 \leq j \leq L$) for the list. If there is a match of length $j$ at position $i$, $m(i, j)$ is set to 1, otherwise $m(i, j)$ is 0. They trace down a suffix tree to construct a match list at each input position. Thus it requires $O(L)$ time and $n$ processors to construct the match list (alternatively $O(L \log n)$ time and $O(n/\log n)$ processors). Then they find the maximum match length $L^*$ in the input string in step 2. In step 3, starting from the match length $l = L^*$ down to 1, they iteratively compute the LFF parsing. That is, in step 3a they select the phrases of length $l$ so that there are maximum number of selected phrases of the length, and in step 3b they essentially eliminate the phrases shorter than $l$ that overlap the selected phrases from the match list. Each iteration requires $O(\log n)$ time, and since there are $O(L)$ iterations, the whole iteration steps require $O(L \log n)$ time.

Our goal in this section is to make the time bound of the LFF parsing $O(L+\log n)$. Our approach for LFF parsing is similar to the one we used for the optimal parsing, in that we find $\text{next}(i)$ for each input position $i$ and use the standard pointer doubling technique to mark all the break points in the input string. This is possible because we can prove that if an input position $i$ is a break point the next break point can be found by inspecting only the left-most match with respect to (w.r.t.) position $i$ for each match length.

We start from Stauffer and Hirschberg's algorithm in Fig. 3. Note that this algorithm does not specify whether it is for a parallel machine or for a sequential machine. From now on, let us assume that it is a sequential algorithm and that the maximum collection of non-overlapping matches of length $l$ in the step 3a is computed from left to right in the following way:

Select left-most match of length $l$, and eliminate any match of length $l$ that overlaps the selected one. Repeat this process until there are no more matches of length $l$.

1The meaning of the left-most match w.r.t. position $i$ is inclusive, i.e., if there is a match at position $i$, it is the left-most match.
In the algorithm in Fig 3, elimination of phrases with length less than \( l \) that overlap the selected phrases is done after all the selection of phrases with length \( l \) is completed. This, however, can be done immediately after each match of length \( l \) is selected. Thus we can modify the step 3 as follows, without altering the result of the algorithm.

3. for each match length \( l := L^* \) down to 1 do
   begin
     for \( i := 1 \) to \( n \) do
       begin
         if there is a match \( Q(i, l) \) then select \( Q(i, l) \);
         eliminate any match with length less than or equal to \( l \) that overlaps \( Q(i, l) \);
       end
   end

If a matched phrase \( Q(k_2, l_2) \) is eliminated after another matched phrase \( Q(k_1, l_1) \) is selected, we say \( Q(k_1, l_1) \) eliminates \( Q(k_2, l_2) \). From above modified step 3, it is clear that \( Q(k_1, l_1) \) may eliminate \( Q(k_2, l_2) \) if and only if \( l_1 > l_2 \) and they overlap \((k_1 - l_2 + 1 \leq k_2 \leq k_1 + l_1 - 1, \) or \( l_1 = l_2 \) and \( k_1 < k_2 \) and they overlap \((k_2 \leq k_1 + l_1 - 1) \). A matched phrase is selected if it is not eliminated until it becomes the left-most among the longest matches that have been neither selected nor eliminated. Eventually, a matched phrase \( Q(k_2, l_2) \) will be eliminated or selected. However, if the result would be different when assuming that another matched phrase \( Q(k_1, l_1) \) did not exist, we say \( Q(k_1, l_1) \) affects \( Q(k_2, l_2) \). If a matched phrase at an input position is eliminated, it would be the same as that the match did not take place at this particular position. Thus a matched phrase that will be eliminated will not affect any other matched phrase. Also, the influence of a matched phrase starts when it is selected. Thus a matched phrase will not affect any match that is longer than itself because the longer match is selected earlier.

Let us consider the LFF algorithm with the modified step 3 shown above, and suppose that on a particular iteration of the loop, a matched phrase \( Q(k, l) \) has been selected and the matches of length less than or equal to \( l \) that overlap \( Q(k, l) \) have been eliminated. Then, the matches on the left side of \( Q(k, l) \) and those on the right side are isolated, and further parsing of the left side is not affected by that on the right side because there is no match on one side that overlap any match on the other side. Thus we obtain the following lemma.

**Lemma 1:** If a matched phrase \( Q(k, l) \) has been selected in a process of the LFF parsing, any matched phrase \( Q(k_1, l_1) \) for which \( k_1 + l_1 - 1 < k \) will not be affected by any match \( Q(k_2, l_2) \) for which \( k_2 > k \).

Suppose that there are \( q \) matches at position \( i \), \( Q(i, l_1), Q(i, l_2), \ldots, Q(i, l_q) \) and that position \( i \) is a break point, then one of them will be selected. Using Lemma 1, we will 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 inspect 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.

**Lemma 2:** Suppose that input position \( i \) is a break point. A matched phrase
$Q(k, l)$ will affect the matched phrases at position $i$ only if $Q(k, l)$ is the left-most match of length $l$ w.r.t. position $i$.

**Proof:** Suppose $Q(k, l)$ is not the left-most match of length $l$ w.r.t. position $i$ and that the left-most match of length $l$ is $Q(j, l)$ ($i \leq j < k$).

Case 1: when $Q(k, l)$ overlaps $Q(j, l)$ ($j + 1 \leq k \leq j + l - 1$)

1.1 if $Q(j, l)$ is not eliminated by a match with longer length, then $Q(k, l)$ will be eliminated by $Q(j, l)$.

1.2 if $Q(j, l)$ is eliminated by $Q(g, h)$ for which $h > l$, then $Q(k, l)$ is also eliminated by $Q(g, h)$.

1.3 if $Q(j, l)$ is eliminated by $Q(g, h)$ for which $h > l, g < j$, even if $Q(k, l)$ is not eliminated by $Q(g, h)$, it cannot affect any of the matches at position $i$ because of Lemma 1.

Case 2: when $Q(k, l)$ does not overlap $Q(j, l)$ ($j + l = k$)

2.1 if $Q(j, l)$ is not eliminated by matches with longer length, then $Q(j, l)$ is selected and $Q(k, l)$ will not affect any of the matches at position $i$ because of Lemma 1.

2.2 if $Q(j, l)$ is eliminated by a match with longer length, even if $Q(k, l)$ is not eliminated it will not affect any of the matches at position $i$ because of Lemma 1.

With this background, we can now state our parallel LFF parsing algorithm. It is shown in Fig. 4. In step 1, the match list $(m(i, j))$ is constructed as in step 1 of Stauffer and Hirschberg's algorithm. This step requires $O(L)$ time and $O(n)$ processors. In step 2, the distance of the left-most match of length $j$ from each input position $i$ ($c(i, j)$) is computed for $j = 1$ through $L$. We use the pointer doubling technique to carry out this operation. At first, the pointer $P(i)$ is initialized to $i + 1$ for $1 \leq i \leq n$ and $P(n + 1)$ to $n + 1$, and the temporary variable $temp(i, j)$ ($1 \leq i \leq n + 1$ and $1 \leq j \leq L$) is initialized by assigning $i$ if there is a match of length $j$ at position $i$, and 0 otherwise. The next repeat loop is designed in such a way that after completion, $temp(i, j)$ holds the position of the left-most match of length $j$ w.r.t. position $i$. Finally, the distance $c(i, j)$ is computed by subtracting $i$ from $temp(i, j)$. In this step, one processor is assigned to each element of the array $temp(i, j)$. Thus $O(nL)$ processors are required. The time bound is $O(\log n)$. In step 3, we find the next break point for each input position $i$. As mentioned before, inspection of only the left-most match w.r.t. position $i$ for each match length is enough to carry out this operation. The loop of step 3 starts from the longest match length $L$ and in each iteration the length is decremented. On a particular iteration, we check the left-most match of length $l(i)$ w.r.t. position $i$ by inspecting the distance $c(i, l(i))$. If $c(i, l(i)) = 0$, the match of length $l(i)$ is at position $i$, thus a phrase of length $l(i)$ is selected at position $i$ and the next break point is $i + l(i)$. Then the iteration for position $i$ stops here. If $0 < c(i, l(i)) < \infty$, the phrase of length $l(i)$ at position $i + c(i, l(i))$ is selected, and the left-most (w.r.t. position $i$) matches of length shorter than $l(i)$ that overlap the selected phrase are eliminated. We eliminate them by setting the corresponding distances to $\infty$. We also eliminate the matches that are on the right side of the selected phrase because they are not the matches at position $i$ nor will...
in parallel for $1 \leq i \leq n + 1$ do
  begin
    if $i < n + 1$ then do
      for $1 \leq j \leq L$ do
        if there is a match with length $j$ at position $i$ then $m(i, j) := 1$ else $m(i, j) := 0$;
      else $m(i, j) := 0$;
    2 if $i < n + 1$ then $P(i) := i + 1$ else $P(i) := i$;
  in parallel for $1 \leq j < L$ do
    begin
      if $m(i, j) = 1$ then $temp(i, j) := i$ else $temp(i, j) := \infty$;
      repeat $[\log n]$ times
        begin
          if $temp(i, j) = \infty$ then $temp(i, j) := temp(P(i), j)$;
          $P(i) := P(P(i))$;
        end
      $c(i, j) := temp(i, j) - i$;
    end
  3 if $i < n + 1$ then do
    begin
      $l(i) := L$;
      repeat until $c(i, l(i)) = 0$ do
        begin
          if $c(i, l(i)) < \infty$ then do
            in parallel for $1 \leq j < l(i)$ do
              if $c(i, j) + j - 1 \geq c(i, l(i))$ then $c(i, j) := \infty$;
              $l(i) := l(i) - 1$;
            end
          next(i) := $i + l(i)$;
        end
      else next(i) := $i$;
    4 same as step 3 of Fig 2
  end

Figure 4: The $O(L \log n)$ time $O(nL)$ processor algorithm for LFF parsing.
they affect the matches at position \( i \). By assigning one processor to each element of the array \( c(i,j) \), we can do the elimination of matches in constant time, thus step 3 requires \( O(L) \) time and \( O(nL) \) processors. Step 4 applies the same pointer doubling technique as the one we used in step 3 of Fig. 2, thus it requires \( O(\log n) \) time and \( O(nL) \) processors. Overall, our LFF parsing algorithm requires \( O(L + \log n) \) time and \( O(nL) \) processors. With the assumption \( L = O(\log n) \), it requires \( O(\log n) \) time and \( O(n\log n) \) processors.

In our parallel LFF parsing algorithm, we used the distance \( c(i, 1) \) of the left-most (w.r.t. position \( i \)) match with length 1 from position \( i \) to compute the next break point. This distance could be as large as \( n - 1 \). However, it is not necessary to look for a match in an arbitrarily large substring on the right of position \( i \). In fact, it is enough if we look for a match of length 1 up to \( \frac{1}{2}(l - 1)(l - 2) \) positions to the right. This fact is stated in the following lemma.

**Lemma 3:** Suppose that input position \( i \) is a break point and that \( Q'(k_1, l) \) is the left-most match of length 1 w.r.t. position \( i \). If \( k_1 - i > \frac{1}{2}(l - 1)(l - 2) \) for \( l \geq 2 \), then \( Q'(k_1, l) \) will not affect the matches at position \( i \).

**Proof:** We consider only the left-most matches w.r.t. position \( i \) in this proof because by Lemma 2 these are the only matches that may affect the matches at position \( i \). Also, we consider only the matches that are shorter than or equal to \( l \) because by the time \( Q'(k_1, l) \) is selected all the matches with longer length must have been eliminated or selected. Let us denote these left-most matches of length 1, 2, \ldots, \( l \) by \( Q'(k_1, 1), Q'(k_2, 2), \ldots, Q'(k_l, l) \) respectively. In order for a match \( Q'(k_1, 1) \) to be able to affect a match at position \( i \), any match \( Q'(k_a, a) \) for which \( a < l \) and \( k_a < k \) has to overlap with at least one match \( Q'(k_b, b) \) for which \( a < b \leq l \), because if there is a match \( Q'(k_a, a) \) for which \( a < l \) and \( k_a < k \), but with which no match with longer length overlap, then \( Q'(k_a, a) \) will be selected and because of Lemma 1 \( Q'(k_1, l) \) will not affect any match at position \( i \). The longest distance a match \( Q'(k_1, l) \) can be far away from position \( i \) without violating above condition is obtained when \( Q'(k_1, 1) \) and \( Q'(k_2, 2) \) are at position \( i \), and for any \( Q'(k_c, c) \) for which \( 3 \leq c \leq l - 1 \) the left-most symbol overlaps with the right-most symbol of \( Q'(k_{c+1}, c - 1) \) and the right-most symbol overlaps with the left-most symbol of \( Q'(k_{c+1}, c + 1) \). The longest distance is given by \( 1 + 2 + \cdots + (l - 2) = \frac{1}{2}(l - 1)(l - 2) \).

Since the longest possible match length is \( L \), Lemma 3 means that no match that is more than distance of \( \frac{1}{2}(L - 1)(L - 2) \) away from position \( i \) will affect the matches at position \( i \). Thus in step 2 of the algorithm in Fig. 4, we need to repeat the loop only \( \lfloor \log \left( \frac{1}{2}(L - 1)(L - 2) \right) \rfloor \) times instead of \( \lfloor \log n \rfloor \) times.

We can use Lemma 3 to show that the sequential LFF dictionary compression can be computed on-line with the lookahead of length \( O(L^2) \). In the on-line LFF dictionary compression, the input string is processed from left to right and at each step a matched phrase starting at the first symbol of the encoded portion of the input string is selected, encoded, and removed. The selection of the matched phrase has to be done in such a way that the result is the same as when the LFF parsing algorithm in Fig. 3 is used. Suppose that the input string has been encoded up to position \( i - 1 \). Because of Lemma 2 selection of the matched phrase at position \( i \) can be done
by finding the left-most match for each match length, and because of Lemma 3 we can disregard the left-most match of length \( l \) if its starting position is farther than \( i + \frac{1}{2}(l-1)(l-2) \). Thus if we have a lookahead buffer of length \( \frac{1}{2}(L-1)(L-2)+L \) we can compute LFF dictionary compression on-line. There is an addition of \( L \) because we may have to find a match of length \( L \) at the right-most position that can affect the matches at position \( i \) (this right-most position is \( i + \frac{1}{2}(L-1)(L-2) \)), thus we have to be able to look at up to position \( i + \frac{1}{2}(L-1)(L-2)+L-1 \).

4 Future Work

Since Constantinescu and Storer have shown in [9] that generalization of dictionary type data compression to approximate pattern matching and two-dimensional data is important, parallel algorithms for generalized dictionary type data compression are of interest.

References


