Theoretical Computer Science 51 (1987) 341–348
North-Holland

NOTE

PARALLEL STRING MATCHING WITH k MISMATCHES*

Zvi GALIL

Department of Computer Science, Columbia University, New York, NY 10027, U.S.A., and Department of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel

Raffaele GIANCARLO

Department of Computer Science, Columbia University, New York, NY 10027, U.S.A., and Department of Computer Science, University of Salerno, 84100 Salerno, Italy

Communicated by G. Ausiello
Received September 1986

Abstract. Two improved algorithms for string matching with k mismatches are presented. One algorithm is based on fast integer multiplication algorithms whereas the other follows more closely classic string-matching techniques.

Introduction

Given a text string $t_0t_1 \ldots t_{n-1}$, a pattern string $p_0p_1 \ldots p_{m-1}$ and an integer k, $k \leq m \leq n$, we are interested in finding all occurrences of the pattern in the text with at most k mismatches, i.e., with at most k locations in which the pattern and the text have different symbols. We refer to this problem as string matching with k mismatches.

Recently, an efficient sequential algorithm for this problem was devised by [6, 7] and an improvement of it was presented in [2]. Moreover, Landau and Vishkin have also considered both sequential and parallel algorithms for the more general problem of string matching with k differences [8]; that is, a total of at most k *mismatches* between symbols, *insertions* and/or *deletions* of symbols are allowed in order to obtain an occurrence of the pattern in any position of the text. Obviously, the parallel algorithm for string matching with k differences, referred to as LV algorithm, can also handle the special case of string matching with k mismatches, but no substantial improvement in its time bound of $O(k+\log m)$ would result. This time bound is optimal if $k = O(\log m)$. But for larger values of k the advantage of using parallelism is lost.

Here we present two efficient parallel algorithms for string matching with k mismatches. One algorithm, referred to as A1, is a parallel implementation of

* Work supported in part by NSF Grants MCS-830-3139 and DCR-85-11713.

Schönhage and Strassen integer-multiplication algorithm adapted to compute the Hamming distances between a binary pattern and its potential occurrences in a binary text. Such an algorithm uses $O(nq \log m \log\log n)$ processors and it takes $O(\log n)$ time, where $q = \min(\sigma, m)$ and σ denotes the size of the alphabet. We remark that the use of fast integer-multiplication algorithms to solve efficiently a wide class of string matching problems is not new [1] (see also [4]), and we discuss A1 for the sake of completeness.

The other algorithm, referred to as A2, assigns $2k$ processors to each position of the text and then locates (up to) $k+1$ mismatches between the pattern and such a substring of the text. It consists of two major steps:

(a) preprocessing of the pattern and the text and

(b) finding all occurrences of the pattern in the text with at most k mismatches.

Step (b) can be implemented in time $O(\log(m/k) \log k/(\log\log m + \log\log k))$ by using $O(nk)$ processors. The preprocessing step can be implemented by means of two different algorithms. One algorithm, due to [8], takes $O(\log m)$ time by using $O(m^2+n)$ processors. The other algorithm, due to [11], takes $O(\log n)$ time by using $O(n+m)$ processors. Thus, A2 performs in time $O(\log(m/k) \log k/(\log\log m + \log\log k) + \log m)$ with $O(m^2+nk)$ processors or in time $O(\log(m/k) \log k/(\log\log m + \log\log k) + \log n)$ with $O(m+nk)$ processors depending on which implementation of the preprocessing is adopted.

An informal discussion of the main features of algorithms LV, A1 and A2 is in order. As it was pointed out earlier, LV is efficient only when $k = O(\log m)$. Algorithm A1 guarantees a good time performance irrespective of the order of magnitude of k. However, it has two major drawbacks: the number of processors depends linearly on q, and thus on the alphabet size, and the constant hidden in the big-O notation is quite large. Moreover, its worst-case time bound is achieved by any instance of the problem. This is also true for LV. As far as A2 is concerned, its worst-case time bound is $O(\log m)$ whenever $k = O(\log^c m)$ or $k \geq m/\log^c m$, c constant, and is never worse than $O(\log^2 m/\log\log m)$. Moreover, its time performance depends on the input strings; thus, A2 may behave better than its worst-case time bound. The major drawback of A2 is that if $k \approx m$, it uses essentially the same number of processors as the naive algorithm achieving the same time performance.

The model of computation that we assume is the exclusive-read/exclusive-write (EREW) parallel random-access machine (PRAM) [3]. An EREW PRAM is composed of t synchronous processors all having access to a common memory. However, processors are not allowed to read simultaneously or to write from the same memory location.

1. Algorithm A1

Let $text = t_0 t_1 \ldots t_{n-1}$ be a binary text, and let $pat = p_0 p_1 \ldots p_{m-1}$ be a binary pattern. The occurrences of *pat* in *text* with at most k mismatches can be found by

computing the Hamming distance H between *pat* and $t_i \ldots t_{i+m-1}$, $0 \leq i \leq n-m$. If $H(pat, t_i \ldots t_{i+m-1}) \leq k$, then i is an occurrence of the pattern in the text.

The Hamming distance between two strings a and b of length m is given by $H(a, b) = \sum_{j=0}^{m-1} a_j \oplus b_j$.

Let $rev(b) = b_{m-1} \ldots b_0$. Since $a_j \oplus b_j = (a_j \overline{b_j}) + (\overline{a_j} b_j)$, $H(a, b)$ can be rewritten as

$$H(a, b) = \sum_{j=0}^{m-1} (a_j \overline{rev(b)_{m-j}}) + \sum_{j=0}^{m-1} (\overline{a_j} rev(b)_{m-j}).$$

$H(a, b)$ can be computed by first inserting $\log m$ 0's between each bit of a and each bit of b, thus obtaining two strings a' and b' of length $m(\log m + 1)$ each. Then, the products $c = a' \overline{rev(b')}$ and $d = \overline{a'} rev(b')$ are computed. Finally, $H(a, b)$ is given by the sum of the two binary numbers

$$c_{(m-1)(\log m+1)+\log m} \cdots c_{(m-1)(\log m+1)} \quad \text{and} \quad d_{(m-1)(\log m+1)+\log m} \cdots d_{(m-1)(\log m+1)}$$

extracted from c and d respectively. The role of the blocks of 0's is to separate the result from the other carries.

The above method can be easily extended to compute concurrently $H(pat, t_i \ldots t_{i+m-1})$ for all i, $0 \leq i \leq n-m$. Indeed, both the text and the pattern are transformed into strings *text'* of length $n(\log m + 1)$ and *pat'* of length $m(\log m + 1)$. Then, the products

$$c = (text')(\overline{rev(pat')}) \quad \text{and} \quad d = (\overline{text'})(rev(pat'))$$

are computed. Now, $H(pat, t_i \ldots t_{i+m-1}) = c_i + d_i$, where

$$c_i = c_{(m-1+i+1)(\log m+1)-1} \cdots c_{(m-1+i)(\log m+1)}$$

and

$$d_i = d_{(m-1+i+1)(\log m+1)-1} \cdots d_{(m-1+i)(\log m+1)}.$$

It has been shown in [3] that a parallel integer multiplication of two s-bit numbers can be performed in time $O(\log s)$ with $O(s \log s)$ processors. Thus, parallel string matching with k mismatches can be performed in time $O(\log(n \log m))$ with $O(n \log m \log \log n \log m)$ processors provided that the input alphabet is binary. If the size σ of the input alphabet is greater than two, then each character can be represented by $q = \min(\sigma, m)$ bits, i.e., the ith character is represented by a bit vector with the ith bit set to 1 and the remaining ones set to 0. Thus, parallel string matching with k mismatches can be solved in $O(\log nq)$ time by $O(nq \log m \log \log n)$ processors.

2. Algorithm A2

Algorithm A2 is composed of two major steps:

(1) preprocessing of the pattern and the text;

(2) detection of all occurrences of the pattern in the text with at most k mismatches.

The first step is devoted to the characterization of all substrings of the text in terms of substrings of the pattern. This goal can be accomplished by means of two different algorithms. One algorithm [8] constructs some tables, whereas the other one [11] uses the suffix tree of the pattern and the text [9, 12]. In what follows we briefly describe the two algorithms omitting the details of their implementation. Then we present the string matching algorithm.

The preprocessing step in [8] computes two arrays: MAX-LENGTH$[0, \ldots, m-1; 0, \ldots, m-1]$ and BEST-FIT$[0, \ldots, n-1]$. They are defined as follows. MAX-LENGTH$[i, j] = l$ if $p_i \ldots p_{i+l-1} = p_j \ldots p_{j+l-1}$ and $p_{i+l} \neq p_{j+l}$, i.e., suffixes $p_i \ldots p_m$ and $p_j \ldots p_m$ of the pattern have a maximal common prefix of length l. BEST-FIT$[i] = (j, l)$ if $t_i \ldots t_{i+l-1} = p_j \ldots p_{j+l-1}$ and j is the position in the pattern for which l is maximal. That is, $p_j \ldots p_{j+l-1}$ is the longest substring of the pattern starting at position i of the text.

Table MAX-LENGTH can be easily computed in $O(\log m)$ time by $m^2/\log m$ processors, whereas BEST-FIT can be computed in $O(\log m)$ time by $m^2 + n/\log m$ processors. The interested reader can find the details of such a construction in [8].

A basic operation in the pattern-matching algorithm is the detection of the leftmost mismatch between a suffix of the text and one of the pattern. This operation can be performed in constant time by making use of MAX-LENGTH and BEST-FIT. Indeed, let FIND-MISMATCH(i, j) be a function that gives the text position of the leftmost mismatch between suffixes $t_i \ldots t_{n-1}$ and $p_j \ldots p_{m-1}$. Now, FIND-MISMATCH$(i, j) = i + \min(q, l)$ if BEST-FIT$[i] = (s, q)$ and MAX-LENGTH$[s, j] = l$. This result easily follows by observing that BEST-FIT$[i] = (s, q)$ implies that $t_i \ldots t_{i+q-1} = p_s \ldots p_{s+q-1}$ with $t_{i+q} \neq p_{s+q}$ and MAX-LENGTH$[s, j] = l$ implies that $p_s \ldots p_{s+l-1} = p_j \ldots p_{j+l-1}$ and $p_{s+l} \neq p_{j+l}$. Thus, $t_{i+\min(q,l)} \neq p_{j+\min(q,l)}$ and such a mismatch must be the first one between the given suffixes.

The function FIND-MISMATCH(i, j) can also be computed as follows. Assume that T is the suffix tree of the string $t_0 \ldots t_{n-1} \$ p_0 \ldots p_{m-1}$ [9, 12]. Then, FIND-MISMATCH$(i, j) = i + q$, where q is the depth of the *Lowest Common Ancestor* (LCA) between leaves i and $n + j + 1$ of T. Recently, Schieber [11] showed how to construct suffix tree T in $O(\log n)$ time by using $O(n)$ processors. He also presented parallel algorithms that set up data structures as in [5] allowing each LCA query on T to be answered in $O(1)$ time. These data structures can be obtained in $O(\log n)$ time with $O(n)$ processors. Thus, FIND-MISMATCH(i, j) can be computed in constant time provided that the preprocessing step consists of the algorithms in [11].

In order to test whether an occurrence of the pattern starts at position i of the text (Procedure Occurrence) (see Appendix) $2k$ processors are used to detect (up to $k+1$) mismatches between $t_i \ldots t_{i+m-1}$ and $p_0 \ldots p_{m-1}$. Processors may be active or idle depending on whether or not they are assigned to a substring of $t_i \ldots t_{i+m-1}$. Each active processor is assigned to a substring $t_{i+j} \ldots t_{i+s}$, $0 \leq j \leq s \leq m-1$ of the text and its task is to find up to the first $\log k$ (w.l.o.g., $k = 2^l$) mismatches between

such a substring and $p_j \ldots p_s$ (Procedure Mismatch) (see Appendix). We remark that the strings assigned to the active processors need not be contiguous. As soon as an active processor completes its task, it can be in one of two states: busy or free.

A processor, assigned to $t_{i+j} \ldots t_{i+s}$, is in the *busy* state if in detects the (log k)th mismatch in a position $q-1 < i+s$. That is, substring $t_q \ldots t_{i+s}$ has not been processed yet and it must be tested for possible mismatches. Otherwise, a processor is in the *free* state. A busy processor reports the endpoints (q, s) of the substring that remains to be tested.

As soon as active processors finish their task, the $2k$ processors are again assigned to substrings of $t_i \ldots t_{i+m-1}$ that have not been processed yet and then each processor performs its task on the given substring. We remark that, at this stage, some processors may turn out to be idle. When such an assignment takes place, we say that a new iteration is started. Initially, $t_i \ldots t_{i+m-1}$ is divided into k contiguous substrings of length at least $\lfloor m/k \rfloor$ and at most $\lceil m/k \rceil$. Then, k processor are assigned to each one of such strings. Subsequently, processors are assigned to strings of almost the same length as follows.

Assume that, at the end of iteration j, c processors report that substrings $x_1, x_2, \ldots, x_c$, with endpoints $(q_1, s_1), \ldots, (q_c, s_c)$, of $t_i \ldots t_{i+m-1}$ remain to be tested. Notice that $c < k/\log k$ since each of these processors found log k mismatches. Let $z = \sum_{i=1}^{c} z_i$, $z_i = s_i - q_i + 1$. We assign $p_i = \lceil kz_i/z \rceil$ processors to substrings x_i. The p_i processors can be assigned to substrings x_i, for all i, $1 \leq i \leq c \leq k$, by sorting the triples (q_i, s_i, p_i). Thus, at the beginning of iteration $j+1$, $k \leq 2k$ processors are active and each active processor is assigned to a substring of length at most $\lceil z_i/p_i \rceil \leq z/k$. It is worth to point out that whenever $z < 2k$, the string matching process for position i of the text is concluded as soon as active processors complete their task.

Procedures Occurrence(i) and Mismatch implement the algorithm outlined above for the detection of an occurrence of the pattern in the text at position i.

Complexity of Procedure Occurrence(i).

Each iteration of the **while** loop in Procedure Occurrence(i) takes O(log k) time. Indeed, a cali to Procedure Mismatch takes O(log k) time since it finds up to log k mismatches by using function Find-Mismatch. The operation **update** v is a parallel addition of all the mismatches found during the current iteration and thus it can be performed in O(log k) time by $2k$ processors. Finally, the assignment of processors to substrings takes O(log k) since it essentially reduces to sorting at most $k/\log k$ triples [10].

The number of iterations sufficient to test whether $t_i \ldots t_{i+m-1}$ is an occurrence of the pattern with at most k mismatches can be derived as follows.

Let k_i and l_i be the total number of mismatches found and the total length of the substrings that remain to be processed repectively, when iteration i is started. Initially, $l_0 = m$ and $k_0 = 0$. Let $\alpha_i k$ be the number of busy processors after iteration

i. Then, at the completion of such iteration, $k_{i+1} \geq k_i + \alpha_i k \log k$ since at least $\alpha_i k \log k$ mismatches have been found and $l_{i+1} \leq \alpha_i k l_i / k = \alpha_i l_i$. The algorithm halts when either $l_i \leq 2k$ or $k_i \geq k$. Thus, the maximum number of iterations is given by the *maximal* s such that

$$m \prod_{j=1}^{s} \alpha_j \geq k, \qquad \sum_{j=1}^{s} \alpha_j k \log k \leq k.$$

Now, the *maximal* s is achieved when all the α_j's are equal, that is $\alpha_j = 1/(s \log k)$. Thus, we obtain that

$$s \approx \frac{\log(m/k)}{\log\log(m/k) + \log\log k}.$$

Hence, Procedure Occurrence takes $O(\log(m/k) \log k/(\log\log(m/k) + \log\log k))$.

It follows from the analysis of Procedure Occurrence that the overall time complexity of the algorithm presented in this section is

$$O\left(\frac{\log(m/k) \log k}{\log\log m + \log\log k} + \textit{time preprocessing}\right),$$

where *time preprocessing* can be either $O(\log m)$ ([8] preprocessing algorithm) or $O(\log n)$ ([11] preprocessing algorithm). The number of processors needed is $O(m^2 + nk)$ or $O(nk)$ respectively.

Complexity for a random string

Consider the following restricted version of the algorithm. We define a processor to be free iff it finds at most 1 mismatch in the string assigned to it. Obviously, at any stage, there cannot be more than $\frac{1}{2}k$ busy processors.

Letting $q \leq \frac{1}{2}$ be the probability of a mismatch we find that the probability of a processor being free after the first step is $q^{(m/k)-1}$. Thus, the average number of free processors after the first step is $kq^{(m/k)-1}$ which is less than $\frac{1}{2}k$ for $k \leq \frac{1}{2}m$. Hence, after the first step, the number of busy processors is larger than $\frac{1}{2}k$ on the average. This immediately establishes an $O(\log k)$ time bound for the algorithm.

Concluding remarks

The main advantage of A2 compared to LV is that its time complexity does not depend linearly on k. Moreover, the constants involved in the time bound do not depend on the alphabet size, so A2 compares favorably also with respect to A1.

Appendix

Procedure Occurrence(i)
begin
$v := 0$

```
  (* v is equal to the number of mismatches found so far *)
  partition t_i ... t_{i+m-1} into k contiguous strings of roughly the same length and
  assign them to k processors
  while v ≤ k and (number of active processors) > 0 do
  begin
    for each active processor q pardo Mismatch(q);
    update v
    if v ≤ k then assign substrings not processed to processors
             else stop
  end
  if v ≤ k then print "position i is an occurrence of the pattern in the text"
end.

Procedure Mismatch(q)
(* Assume that string t_{i+j} ... t_{i+s} has been assigned to processor q *)
begin
  vv := 0; free := false; tp := i+j; pp := j; mp := 0;
(* tp and pp denote current text and pattern positions respectively *)
(* mp denotes a mismatching position in the text *)
while vv < log k or free = false do
  begin
    mp := FIND-MISMATCH(tp, pp)
      begin-case
        mp < i+s: vv := vv+1;
        mp = i+s: vv := +1; free := true;
        mp > i+s: free := true;
      end-case
    tp := tp+mp+1; pp := +mp+1
  end
end.
```

References

[1] M.J. Fischer and M.S. Paterson, String-matching and other products, in: R.M. Karp, ed., *Complexity of Computation* (*SIAM-AMS Proceedings* 7) (American Mathematical Society, Providence, RI, 1974) 113-125.

[2] Z. Galil and R. Giancarlo, Improved string matching with *k* mismatches, *Sigact News* 17 (1986) 52-54.

[3] Z. Galil and W.J. Paul, An efficient general-purpose parallel computer, *J. ACM* 30 (1983) 360-387.

[4] J. Hershberger and E. Mayr, Fast sequential algorithms to find shuffle minimizing and shortest paths in a shuffle exchange network, Techn. Rept. STAN-CS-85-1050, Department of Computer Science, Stanford University, Stanford, CA, 1985.

[5] D. Harel and R.E. Tarjan, Fast algorithms for finding nearest common ancestors, *SIAM J. Comput.* 13 (1984) 338-355.

[6] G.M. Landau and U. Vishkin, Efficient string matching in the presence of errors, in: *Proc. 26th IEEE FOCS* (1985) 126-136.
[7] G.M. Landau and U. Vishkin, Efficient string matching with k mismatches, *Theoret. Comput. Sci.* 43 (1986) 239-249.
[8] G.M. Landau and U. Vishkin, Introducing efficient parallelism into approximate string matching and a new serial algorithm, in: *Proc. 18th ACM STOC* (1986) 220-230.
[9] E.M. McCreight, A space economical suffix tree construction algorithm, *J. ACM* 23 (1976) 262-272.
[10] J.H. Reif, An optimal parallel algorithm for sorting, *Proc. 26th IEEE FOCS* (1985) 496-503.
[11] B. Schieber, Personal communication, 1986.
[12] P. Wiener, Linear pattern matching algorithms, *Proc. 14th Symp. on Switching and Automata Theory* (1973) 1-11.