

Detecting False Matches in String-Matching Algorithms¹

S. Muthukrishnan²

Abstract. Consider a text string of length n , a pattern string of length m , and a match vector of length n which declares each location in the text to be either a *mismatch* (the pattern does not occur beginning at that location in the text) or a *potential match* (the pattern may occur beginning at that location in the text). Some of the potential matches could be *false*, i.e., the pattern may not occur beginning at some location in the text declared to be a potential match. We investigate the complexity of two problems in this context, namely, *checking* if there is any false match, and *identifying* all the false matches in the match vector.

We present an algorithm on the CRCW PRAM that checks if there exists a false match in $O(1)$ time using $O(n)$ processors. This algorithm does not require preprocessing the pattern. Therefore, checking for false matches is *provably* simpler than string matching since string matching takes $\Omega(\log \log m)$ time on the CRCW PRAM. We use this simple algorithm to convert the Karp–Rabin Monte Carlo type string-matching algorithm into a Las Vegas type algorithm without asymptotic loss in complexity. We also present an efficient algorithm for identifying all the false matches and, as a consequence, show that string-matching algorithms take $\Omega(\log \log m)$ time even given the flexibility to output a few false matches.

Key Words. Parallel algorithms, Randomized (Las Vegas) string matching, Checking string matching algorithms.

1. Introduction. Given a pattern string of length m and a text string of length n , the problem of *string matching* is to output a match vector which indicates, for every location in the text, whether or not the pattern occurs beginning at that location in the text. Several efficient deterministic sequential and parallel algorithms are known for string matching. (See [A] for a survey, [KMP], [BM], and [W] for sequential algorithms, and [BG1], [V2], [G3], and [CC⁺] for parallel algorithms.) These algorithms commit no errors in outputting all the locations where the pattern matches the text. Karp and Rabin [KR] presented a probabilistic string-matching algorithm which is simple and efficient; however, it commits errors. In the match vector output by their algorithm, the matches indicated might be *false*, i.e., the pattern might not occur beginning at a location in the text indicated as a match. However, it is guaranteed that the pattern does not occur beginning at those locations in the text which are indicated as mismatches. In this paper we investigate the complexity of detecting the false matches, if any, in such a match vector for the given text and pattern.

The motivation for detecting false matches arises partly from the task of converting the Monte Carlo type string-matching algorithm of Karp and Rabin [KR] into a Las

¹ This research was supported in part by NSF/DARPA under Grant CCR-89-06949 and by NSF under Grant CCR-91-03953. A preliminary version of this paper appears in the *Proceedings of Combinatorial Pattern Matching '93*, Padova, Italy.

² Department of Computer Science, University of Warwick, Coventry, CV4 7AL, England. muthu@dcs.warwick.ac.uk.

Vegas type algorithm. Given a Monte Carlo type algorithm for a problem \mathcal{P} (string matching, in our case), an algorithm that detects errors (false matches, in our case) in the output of the Monte Carlo type algorithm and that which is provably simpler than any algorithm for \mathcal{P} , can be utilized to derive a Las Vegas type algorithm for \mathcal{P} . Also, checking for errors in string-matching algorithms has inherent interest in view of the recently formalized notion of program checking [BK]. In addition, as we show, it helps address a notion of approximate string matching. In the standard notion of approximate string matching, all those positions in the text are sought, where the pattern occurs in the text with at most k character-by-character mismatches, for a parameter k . An alternate notion of approximate string matching is to allow a few false matches in the output. This leads to a question partly addressed in this paper—can string matching be performed more efficiently if a few false matches are tolerated?

First, we consider the problem of *checking* in parallel if there exists a false match in the match vector. Checking can be done in linear work by simply running any optimal parallel string-matching algorithm [CC⁺], [BG1] on the given text and pattern and comparing the output with the given match vector. However, in the spirit of converting a Monte Carlo algorithm to a Las Vegas one, we would like a procedure for checking which is provably simpler than string matching. Utilizing the crucial observation that a series of “structured” matches can be checked quickly without explicitly considering each one of the matches, we present a parallel algorithm for checking which runs in $O(1)$ time on the CRCW PRAM using $O(n)$ processors. This algorithm is simple and it does not require preprocessing the pattern. Note that string matching takes at least $\Omega(\log \log m)$ time on the CRCW PRAM with $O(n)$ processors [BG2]. Hence, checking for the occurrence of a false match in string-matching algorithms is *provably* simpler than string matching.

Using this algorithm that checks for false matches, we convert the Monte Carlo type string-matching algorithm of Karp and Rabin [KR] to a Las Vegas type algorithm. Karp and Rabin gave a parallel algorithm for string matching, denoted \mathcal{A} , that performs linear work, and that outputs false matches with a small probability. To remove the occurrences of the false matches in the output, they suggested naively verifying every match indicated by their algorithm. Thus the resultant algorithm \mathcal{B} that does not commit errors in the form of false matches, performs $O(n + m + (\#M + \#F)m)$ work³ where $\#M$ denotes the number of the matches of the pattern in the text and $\#F$ denotes the number of the false matches in the output of the algorithm \mathcal{A} . In the worst case, $\#M = O(n)$ since the pattern can occur beginning at each of the $n - m + 1$ possible text locations. Therefore, algorithm \mathcal{B} is no better than the naive string-matching algorithm.⁴ By combining our algorithm that checks for an occurrence of a false match with their linear work algorithm \mathcal{A} which might output false matches, we derive a string-matching algorithm that, on any input, performs $O(n + m)$ work with a high probability, never outputs false matches, and detects all the occurrences of the pattern in the text.

Next, we consider the problem of identifying *all* the false matches in the match vector. We present a parallel algorithm for this problem which works in $O(\#F_c)$ time

³ Our discussion here in terms of the work performed by the parallel algorithms in [KR] applies to the time taken by the sequential algorithms there as well.

⁴ Karp and Rabin [KR] state the string-matching problem *unconventionally* as that of finding *one* occurrence of the pattern in the text. Therefore in their case $\#M = 1$.

using $O(n)$ processors on the CRCW PRAM where $\#F_c$ denotes the maximum number of consecutive false matches in any substring of length $m/2$. We use this as a reduction to show that any string-matching algorithm which outputs $o(\log \log m)$ consecutive false matches using $O(n)$ processors while making no errors in finding the locations where the pattern does not match, has to take $\Omega(\log \log m)$ time on the CRCW PRAM.

The rest of the paper is organized as follows. In Section 2, the preliminaries are stated. Two main observations are presented in Section 3. In Section 4, the algorithm for checking is described. This is used in Section 5 to provide Las Vegas type string matching. In Section 6, the complexity of identifying all false matches is considered.

2. Problem Definition and Preliminaries. Let t be the text string of length n , let p be the pattern string of length m , and let M be a binary vector of length n called the *match vector* such that the following holds: if $M(i) = 0$, p does not match t beginning at the location i . If $M(i) = 1$, i is a *potential match* location. A potential match i at which p does not match t is called a *false match*. A potential match i at which p matches t is called a *true match*. Note that $M(i) = 0$ for $n - m + 2 \leq i \leq n$. We consider two problems:

CHECKING. Given t , p , and M , output 1 if there exists a false match in M and 0 otherwise. This problem is denoted $\text{check}(t, p, M)$.

IDENTIFYING. Given t , p , and M , output $N[1 \cdots n - m + 1]$ such that $N[i] = 1$ if i is a false match, and $N(i) = 0$ otherwise. This problem is denoted $\text{identify}(t, p, M)$.

Consider the auxiliary problem of **verifying**: Given a location i in the text, output 1 if the pattern matches the text beginning at i and 0 otherwise. This problem is denoted $\text{verify}(t, p, i)$. Our algorithms for checking and identifying use algorithms for verifying as “black boxes”. The problems of checking, identifying, and verifying can be contrasted with the classical problem of string matching.

STRING MATCHING. Given t and p , determine M , a binary vector of length n such that $M(i) = 1$ if p occurs in t beginning at i and $M(i) = 0$ otherwise.

In what follows, assume that m is even. Our algorithms can be easily modified to the case when m is odd.

The combinatorial structure of the strings is utilized in our algorithms. A period length of a string s is k if $s[i + k] = s[i]$ for $i = 1, \dots, m - k$. The minimal such k that is less than $m/2$ is henceforth called the *period length* [G1], [V1]. The *period* of the string is $s[1] \cdots s[k - 1]$. The period length and the period are also defined alternately (and equivalently) as follows [G1], [V1]: u is the period of s if $s = u^k \bar{u}$, $k > 1$, \bar{u} is a prefix of u and $|u|$, the period length, is minimal. The following basic lemmas are proved elsewhere [G1], [V1].

LEMMA 1 (GCD Lemma) [LS]. *If l_1 and l_2 are two period lengths of a string s such that $|s| \geq l_1 + l_2$, then $\gcd(l_1, l_2)$ must be a period length of s .*

LEMMA 2. *If p occurs in t beginning at i and at $i + d$, $1 \leq d < m/2$, and nowhere in between, then d is the period length of p .*

LEMMA 3. *Let the period length of p be d . If p occurs in t beginning at i and $i + j$ such that $j < m/2$, then j is a multiple of d .*

We divide the text into disjoint segments of length $m/2$ and each segment is called a *block*. Assume that $n = k(3m/2)$, $k \geq 1$. Our algorithms can be modified to handle the other values of n . The following lemma will often be used.

LEMMA 4 [FRW]. *Given a binary vector of length l , the first, second, and last 1 in this vector can be found in $O(1)$ time using $O(l)$ processors on the CRCW PRAM.*

3. Two Basic Observations. Now we make two observations that are useful in checking and identifying false matches quickly. Consider two potential matches in the text at i and $i + kd$ such that d is the period length of the pattern and $kd < m/2$.

LEMMA 5 (Quick-Match Lemma). *If the pattern matches at i and $i + kd$, it matches at $i + ld$ for $1 \leq l \leq k - 1$.*

PROOF. Let the pattern be $u^j\bar{u}$ as in the definition. Since it matches at i and $i + kd$, the text string at i contains $u^k u^j \bar{u}$ which is $u^l u^j u^{k-l} \bar{u}$ for any l , $1 \leq l \leq k - 1$. Thus the pattern matches at $i + ld$. \square

Let the pattern p occur beginning at i in the text and let d be the period length of p . Consider placing a copy of the pattern on the text beginning at $i + kd$, $kd < m/2$. Let $i + kd + f$ be the leftmost position in which this copy of the pattern does not match the text.

LEMMA 6 (Quick-Identify Lemma). *The pattern matches the text beginning at a position $i + ld$ in the text for $1 \leq l \leq k$, if and only if $i + ld + m - 1 < i + kd + f$.*

PROOF. From the definition of the period, it follows that when copies of the pattern are placed on the text beginning at $i + ld$, $1 \leq l \leq k$, all the positions which fall on each other contain the same symbol. Hence, none of the copies which fall on t_{i+kd+f} match the text. Any copy on $i + ld < i + kd + f - (m - 1)$ does not fall on t_{i+kd+f} and every other copy falls on t_{i+kd+f} (see Figure 1). \square

Given a sequence of “structured” potential matches (separated by the period length d), the Quick-Match Lemma implies that it is sufficient to verify the occurrence of the pattern at the last location in the sequence to ascertain if the entire sequence contains a

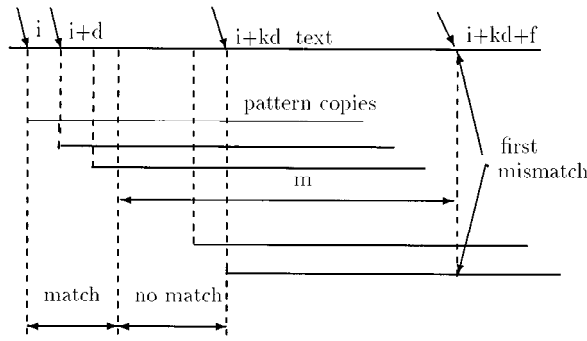


Fig. 1. Quick-Identify.

false match. The Quick-Identify Lemma implies something stronger: placing the pattern at the last location in the sequence, and looking at the leftmost position where it does not match the text, we can *precisely* determine all the false matches in the sequence.

4. Parallel Checking. Assume the common CRCW PRAM model of computation, i.e., all the processors which attempt to write into a memory location write the same value [J].

The problem of verifying, that is, $\text{verify}(t, p, i)$, can be performed using m processors in $O(1)$ time on a common CRCW PRAM. Processor number j , $1 \leq j \leq m$, compares t_{i+j} with p_j , and logical AND of the result of these m comparisons is computed in $O(1)$ time. Naively, the problem of checking can be solved by verifying each potential match in parallel and thereby performing $O(nm)$ work in the worst case.

We perform the checking in linear work by avoiding explicitly verifying every potential match in the match vector. Recall that a block is a substring of length $m/2$. Henceforth assume that each block under consideration has at least two potential matches. Otherwise, we can verify each potential match in the block as described above.

Algorithm Par-Check(t, p, M)

/ Output is set to 1 if a false match is found and to 0 otherwise */*

Set *Output* to 0. Divide the text t (and, correspondingly, M) into $2n/m$ disjoint blocks. For each block do in parallel the following three steps:

1. Let $M[f]$ and $M[s]$ be the first and the second 1 in the block, respectively. Perform $\text{verify}(t, p, f)$ and $\text{verify}(t, p, s)$. If the pattern does not match the text at either f or s , set *Output* to 1 and exit.
2. Let $d = s - f$. If the distance of every potential match from f is not an integral multiple of d , set *Output* to 1 and exit.
3. Let $M[l]$ be the last 1 in the block. Perform $\text{verify}(t, p, l)$. If the pattern does not match the text beginning at l , set *Output* to 1.

LEMMA 7. *Algorithm Par-Check correctly detects the occurrence of a false match.*

PROOF. Within each block, if the first two potential matches f and s are true matches, then the period length of p is $d = s - f$ (Lemma 2). The true matches are multiples of d away from f (Lemma 3). Consider the series of potential matches in the block that are integral multiples of d away from f . The first occurrence in this series, namely, f , is a true match. By the Quick-Match Lemma, if the last occurrence l is a true match, every potential match in the series is a true match. Note that if there is a failure in any of the steps, a false match is found. \square

THEOREM 1. *There exists an algorithm for $check(t, p, M)$ that takes $O(1)$ time using $O(n)$ processors on the common CRCW PRAM.*

PROOF. Consider Algorithm Par-Check. Each of the three verifications per block, namely at $f, s,$ and $l,$ take $O(1)$ time using $O(m)$ processors. Other steps take $O(1)$ time using $O(m)$ processors (Lemma 5). Hence each block can be checked using $O(m)$ processors in $O(1)$ time. The theorem follows since there are $O(n/m)$ such blocks in all. \square

Note that to perform checking in constant time optimally, we have only used the information about the period length of the pattern gleaned from matching the pattern at various positions in the text. In particular, our algorithm does not involve preprocessing the pattern. Let $n = 3m/2$. Breslauer and Galil [BG2] prove that, in this case, any string-matching algorithm takes $\Omega(\log \log m)$ time with $O(n)$ processors on the CRCW PRAM. That Algorithm Par-Check takes a constant time with $O(n)$ processors implies that checking is *provably* simpler than string matching on the CRCW PRAM.

Consider the weaker model of the EREW PRAM [J]. The following holds.

THEOREM 2. *There exists an algorithm that checks for false matches in $O(\log m)$ time and $O(n)$ work on the EREW PRAM.*

PROOF. In Algorithm Par-Check, all the tasks that take $O(1)$ time on the CRCW PRAM using $O(m)$ processors can be replaced by $O(\log m)$ -time algorithms on the EREW PRAM with optimal work by computing along a balanced binary tree in a standard way [J]. \square

5. Las Vegas String Matching. Karp and Rabin [KR] presented a Monte Carlo algorithm (referred to as Algorithm MC henceforth) for string matching.

LEMMA 8 [KR]. *By randomly choosing a prime $\leq mn^k,$ Algorithm MC finds all the matches in $O(\log m)$ time and $O(n)$ work on the EREW PRAM with the probability that a false match occurs on an input instance being at most c/n^{k-1} for a small constant $c.$*

Our Las Vegas string-matching algorithm (referred to as Algorithm LV henceforth) works as follows: it repeatedly runs Algorithm MC checking the output using Algorithm Par-Check till the output contains no false matches.

THEOREM 3. *Algorithm LV correctly finds all true matches. It takes $O(\log m)$ time and performs $O(n)$ work, with high probability, on the EREW PRAM.*

PROOF. Let $k = 2$ in Lemma 9. The probability of occurrence of a false match is at most c/n . The probability that Algorithm MC is run exactly $r + 1$ times is at most c/n^r . Therefore,

$$\Pr(\text{Algorithm MC is run } \geq 3 \text{ times}) \leq \frac{c}{n^2} + \frac{c}{n^3} + \cdots \leq \frac{c}{n(n-1)}.$$

As $n \rightarrow \infty$, $c/n(n-1) \rightarrow 0$. Hence with high probability, Algorithm MC is run $O(1)$ times. The theorem follows from Lemma 9. \square

A sequential simulation of Algorithm LV on the RAM model [AHU] gives a string-matching algorithm which works in linear time with a high probability and detects all the occurrences of the pattern in the text without outputting any false matches.

Karp and Rabin [KR] had suggested naively verifying all the potential matches so as to avoid outputting false matches. Let $\#M$ be the number of occurrences of the pattern in the text and let $\#F$ be the number of false matches in the output of Algorithm MC. Their sequential algorithm takes $O(n + m + (\#M + \#F)m)$ time to find all the occurrences of the pattern in the text. In the worst case, $\#M = O(n)$, and their algorithm is no better than the naive algorithm for string matching even though the probability of Algorithm MC outputting false matches is provably low. In contrast, the sequential simulation of our Algorithm LV works in linear time with a high probability irrespective of $\#M$. Furthermore, it retains the appealing simplicity of Algorithm MC.

6. Identifying in Parallel. In this section we consider the problem of identifying all the false matches. Here the Quick-Identify Lemma is crucial.

Algorithm Par-Identify(t, p, M)

For each block in parallel perform the following three steps:

1. Starting from the leftmost potential match location in the block, consider each successive potential match location in sequence. Verify each potential match in parallel using m processors in $O(1)$ time until the first two true matches f and s are found.
2. Let $d = s - f$. As in Algorithm Par-Check determine all potential matches in the block that are integral multiples of d away from f . Each of the other potential matches in this block is false.
3. Let l be the rightmost potential match that survives the previous step in this block. Find the smallest position $l + s$ where the pattern placed at l does not match the text. Every potential match k such that $k + m - 1 \geq l + s$ is false.

THEOREM 4. *Let $\#F_c$ be the maximum number of consecutive false matches in any*

block. There exists an algorithm that detects all the false matches in $O(\#F_c)$ time using $O(n)$ processors on the common CRCW PRAM.

PROOF. Consider Algorithm Par-Identify. Clearly, at most $2(\#F_c)$ potential matches are verified in Algorithm Par-Identify in sequence before the first two true matches in a block are found. The rest of the algorithm, including finding the first mismatch in step 3, takes $O(1)$ time using $O(m)$ processors (Lemma 5). Correctness easily follows from the Quick-Identify Lemma. \square

THEOREM 5. *Let $n \leq 3m/2$. Any string-matching algorithm \mathcal{A} which outputs at most $\#F_c$ consecutive false matches where $\#F_c = o(\log \log m)$, takes $\Omega(\log \log m)$ time using $O(n)$ processors on the CRCW PRAM.*

PROOF. Suppose algorithm \mathcal{A} takes $o(\log \log m)$ time using $O(m)$ processors on the CRCW PRAM. Given any input text and pattern, run algorithm \mathcal{A} first. Subsequently run Algorithm Par-Identify on the text, the pattern, and the output of algorithm \mathcal{A} and remove all false matches. Thereby, we achieve a string-matching algorithm which correctly outputs all true matches in $o(\log \log m)$ time using $O(m)$ processors on the CRCW PRAM. However, since it is known that string matching takes $\Omega(\log \log m)$ time using $O(m)$ processors on the CRCW PRAM [BG2], we derive a contradiction. \square

Thus the claim that the false matches in the output of a string-matching algorithm can be detected and removed efficiently implies string matching takes $\Omega(\log \log m)$ time even when given the flexibility to output a few false matches. Therefore algorithms faster than those for standard string matching in parallel cannot be achieved by settling for the approximate matching notion of permitting a few false matches in the output of string-matching algorithms.

7. Discussion. We have shown that, in the CRCW PRAM model, checking for false matches in the output of string-matching algorithms is *provably* simpler than string matching. We have also converted the Karp–Rabin Monte Carlo type string-matching algorithm to a Las Vegas type string-matching algorithm. The fingerprinting technique of Karp and Rabin has several other applications [KR] yielding Monte Carlo type algorithms. It will be of interest to convert these into Las Vegas type algorithms as well.

The following problem is the dual of checking for false matches. Consider text t , pattern p , and match vector M such that if $M(i) = 1$, the pattern is guaranteed to match the text beginning at location i . If $M(i) = 0$, i is a *potential mismatch* location. The pattern might match the text beginning at i , for some potential mismatch location i . Such locations are called *false mismatches*. The problem of checking for false mismatches in the match vector remains unexplored.

Acknowledgments. The author sincerely thanks Ravi Boppana, Richard Cole, Babu Narayanan, and Krishna Palem for very helpful discussions.

References

- [A] A. Aho. Algorithms for finding patterns in strings. In *Handbook of Theoretical Computer Science*, Vol. 1, J. Van Leeuwen, ed. Elsevier, Amsterdam, 1990, pp. 255–300.
- [AHU] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.
- [BG1] D. Breslauer and Z. Galil. An optimal $O(\log \log n)$ time parallel string matching algorithm. *SIAM J. Comput.*, 19 (1990), 1051–1058.
- [BG2] D. Breslauer and Z. Galil. A lower bound for parallel string matching. *Proc. 23rd Annual ACM Symposium on Theory of Computation*, 1991, pp. 439–443.
- [BK] M. Blum and S. Kannan. Designing programs that check their work. *Proc. 21st Annual ACM Symposium on Theory of Computation*, 1989, pp. 86–97.
- [BM] R. Boyer and S. Moore. A fast string matching algorithm. *Comm. ACM*, 20 (1977), 762–772.
- [CC⁺] R. Cole, M. Crochemore, Z. Galil, L. Gasieniec, R. Hariharan, S. Muthukrishnan, K. Park, and W. Rytter. Optimally fast parallel algorithms for preprocessing and pattern matching in one and two dimensions. *Proc. IEEE Symposium on Foundations of Computer Science*, 1993.
- [FRW] F. E. Fich, R. L. Ragde, and A. Wigderson. Relations between concurrent-write models of parallel computation. *SIAM J. Comput.*, 17 (1988), 606–627.
- [G1] Z. Galil. Optimal parallel algorithms for string matching. *Inform. and Control*, 67 (1985), 144–157.
- [G2] Z. Galil. Open problems in stringology. In *Combinatorial Algorithms on Words*, A. Apostolico and Z. Galil, eds. NATO ASI Series, Springer-Verlag, New York, 1985, pp. 1–8.
- [G3] Z. Galil. Hunting lions in the desert optimally or a constant time optimal parallel string matching algorithm. *Proc. 24th Annual ACM Symposium on Theory of Computation*, 1992, pp. 69–76.
- [J] J. JaJa. *Introduction to Parallel Algorithms*. Addison-Wesley, Reading, MA, 1991.
- [KMP] D. E. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6 (1973), 323–350.
- [KR] R. Karp and M. O. Rabin. Efficient randomized pattern matching algorithms. *IBM J. Res. Develop.*, 31 (1987), 249–260.
- [LS] R. Lyndon and M. Schutzenberger. The equation $a^M = b^N c^P$ in a free group. *Michigan Math. J.*, 9 (1962), 289–298.
- [V1] U. Vishkin. Optimal pattern matching in strings. *Inform. Control*, 67 (1985), 91–113.
- [V2] U. Vishkin. Deterministic sampling—A new technique for fast pattern matching. *Proc. 22nd Annual ACM Symposium on Theory of Computation*, 1990, pp. 170–180.
- [W] P. Weiner. Linear pattern matching algorithms. *Proc. 14th IEEE Annual Symposium on Switching and Automata Theory*, 1973, pp. 1–11.