[39] Li, M. and Yesha, Y. (1986), String-matching cannot be done by a two-head one-way deterministic finite automaton, *Information Processing Letters 22*, 231-235.

[40] Lothaire, M. (1983), Combinatorics on Words, *Encyclopedia of mathematics and its applications, Vol. 17*, Addison Wesley.

[41] Lyndon, R. C. and Schutzenberger, M. P. (1962), The equation $a^M = b^N c^P$ in a free group, *Michigan Math. J. 9*, 289-298.

[42] McCreight, E. M. (1976), A space-economical suffix tree construction algorithm, *Journal of ACM, 33:3*, 262-272.

[43] Rosser, J. B. and Schoenfeld, L. (1962), Approximate formulas for some functions of prime numbers, *Illinois Journal of Mathematics*, 6:64-94.

[44] Shiloach, Y. and Vishkin, U. (1981), Finding the maximum, merging and sorting in a parallel computation model, *J. Algorithms 2*, 88-102.

[45] Valiant, L. G. (1975), Parallelism in comparison models, *SIAM J. Comput. 4*, 348-355.

[46] Vishkin, U. (1985), Optimal parallel pattern matching in strings, *Information and Control 67*, 91-113.

[47] Vishkin, U. (1990), Deterministic sampling - A new technique for fast pattern matching, *SIAM J. Comput. 20:1*, 22-40.

[48] Weiner, P. (1973), Linear pattern matching algorithms, *Proc. 14th IEEE symp. on switching and automata theory*, 1-11.

[27] Galil, Z. and Giancarlo, R. (1988), Data structures and algorithms for approximate string matching, *Journal of Complexity 4*, 33-72.

[28] Galil, Z. and Seiferas, J. (1980), Saving space in fast string-matching, *SIAM J. Comput. 2*, 417-438.

[29] Galil, Z. and Seiferas, J. (1983), Time-space-optimal string matching, *J. Comput. Syst. Sci. 26*, 280-294.

[30] Geréb-Graus, M. and Li, M. (1990), Three one-way heads cannot do string matching, manuscript.

[31] Karp, R. M., Miller, R. E. and Rosenberg, A. L. (1972), Rapid identification of repeated patterns in strings, trees and arrays, *Proceedings of the 4th ACM Symposium on Theory of Computation*, 125-136.

[32] Karp, R. M. and Rabin, M. O. (1987), Efficient randomized pattern matching algorithms, *IBM J. Res. Develop. 31:2*, 249-260.

[33] Karp, R. M. and Ramachandran, V. (1990), A survey of parallel algorithms for shared-memory machines, *Handbook of theoretical computer science.*

[34] Kedem, Z., Landau, G. and Palem, K. (1988), Optimal parallel suffix-prefix matching algorithm and applications. *manuscript.*

[35] Knuth, D. E., Morris, J. H. and Pratt, V. R. (1977), Fast pattern matching in strings, *SIAM J. Comput. 6*, 322-350.

[36] Kruskal, C. P. (1983), Searching, merging, and sorting in parallel computation, *IEEE trans. on computers 32*, 942-946.

[37] Lander, R. E. and Fischer, M. J. (1980), Parallel Prefix Computation, *J. ACM 27:4*, 831-838.

[38] Li, M. (1984), Lower bounds on string-matching, *TR 84-636 Department of Computer Science, Cornell University.*

[16] Colussi, L., Galil, Z. and Giancarlo, R. (1990), On the exact complexity of string matching, *Proc. 31st IEEE Symp. on Foundations of Computer Science*, 135-143.

[17] Cole, R. (1991), Tight bounds on the complexity of the Boyer-Moore string matching algorithm, *Proc. 2nd annual ACM-SIAM symp. on discrete algorithms*, 224-233.

[18] Cole, R. and Vishkin, U. (1986), Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms, *Proc. 18th ACM Symp. on Theory of Computing*, 206-219.

[19] Cole, R. and Vishkin, U. (1989), Faster optimal prefix sums and list ranking, *Inform. and Comput. 81*, 334-352.

[20] Cook, S. A., Dwork, C. and Reischuk, R. (1986), Upper and lower time bounds for parallel random access machines without simultaneous writes, *SIAM J. Comput. 15:1*, 87-97.

[21] Crochemore, M. (1989), String-Matching and Periods, In *Bulletin of EATCS*.

[22] Crochemore, M. and Perrin, D. (1989), Two way pattern matching, *JACM*, to appear.

[23] Crochemore, M. and Rytter, W. (1990), Usefulness of the Karp-Miller-Rosenberg algorithm in parallel computations on strings and arrays, manuscript.

[24] Eppstein, D. and Galil, Z. (1988), Parallel algorithmic techniques for combinatorial computation, *In Ann. Rev. Comput. Sci. 3*, 233-283.

[25] Fich, F. E., Ragde, R. L., and Wigderson, A. (1984), Relations between concurrent-write models of parallel computation, *Proc. 3rd ACM Symp. on Principles of Distributed Computing*, 179-189.

[26] Galil, Z. (1985), Optimal parallel algorithms for string matching, *Information and Control 67*, 144-157.

[5] Apostolico, A. and Giancarlo, R. (1986), The Boyer-Moore-Galil string searching strategies revisited, *SIAM J. Comput. 15:1*, 98-105.

[6] Apostolico, A., Iliopoulos, C., Landau, G. M., Schieber, B. and Vishkin, U. (1988), Parallel construction of a suffix tree with applications, *Algorithmica 3*, 347-365.

[7] Beame, P., and Hastad, J. (1987), Optimal Bound for Decision Problems on the CRCW PRAM, *Proc. 19th ACM Symp. on Theory of Computing*, 83-93.

[8] Berkman, O., Breslauer, D., Galil, Z. Schieber, B., and Vishkin, U. (1989), Highly parallelizeable problems, *Proc. 21st ACM Symp. on Theory of Computing*, 309-319.

[9] Berkman, O., Schieber, B., and Vishkin, U. (1988), Some doubly logarithmic optimal parallel algorithms based on finding nearest smallers, *manuscript*.

[10] Borodin, A. B., Fischer, M. J., Kirkpatrick, D. G., Lynch, N. A. and Tompa, M. (1979), A time-space tradeoff for sorting on non-oblivious machines, *Proc. 20th IEEE Symp. on Foundations of Computer Science*, 294-301.

[11] Borodin, A., and Hopcroft, J. E. (1985), Routing, merging, and sorting on parallel models of comparison, *J. of Comp. and System Sci. 30*, 130-145.

[12] Boyer, R. S., and Moore, J. S. (1977), A fast string searching algorithm, *Comm. ACM 20*, 762-772.

[13] Brent, R. P. (1974), The parallel evaluation of general arithmetic expressions, *J. ACM 21*, 201-206.

[14] Breslauer, D. and Galil, Z. (1990), An optimal $O(\log \log n)$ parallel string matching algorithm, *SIAM J. Comput. 19:6*, 1051-1058.

[15] Breslauer, D. and Galil, Z. (1991), A lower bound for parallel string matching, *Proc. 23nd ACM Symp. on Theory of Computation*, to appear.

string matching algorithm.

- String matching with long text strings. If the text string is much longer than the pattern, the lower bound of Section 4 does not apply. Indeed, on a comparison model where all computation is free one can do the preprocessing for Vishkin's fast algorithm in constant time using m^2 processors. If $n = m^2$ the n processors are available to preprocess the short pattern. However, it is not known if the preprocessing can be performed on a CRCW-PRAM, or how is can be done faster with less then m^2 processors on a comparison model.

- String matching with preprocessing. What are the exact bounds if preprocessing is free like in Vishkin's fast algorithms. A constant time optimal algorithm is still possible.

- String matching on CREW and EREW-PRAM. The fastest optimal CREW-PRAM deterministic algorithm is obtained by slowing down the CRCW-PRAM algorithm to $O(\log m \log\log m)$ time. What is the exact bound on these models.

7 Acknowledgements

We would like to thank Terry Boult and Thanasis Tsantilas for valuable comments and suggestions for this paper.

References

[1] Aho, A. (1990), Algorithms for finding patterns in strings, *Handbook of theoretical computer science*, 257-300.

[2] Aho, A. and Corasik, M. J. (1975), Efficient string matching: an aid to bibliographic search, *Comm. ACM 18:6*, 333-340.

[3] Apostolico, A. (1989), Optimal parallel detection of squares in strings; Part I: Testing square freedom, *CSD-TR-932, purdue.*

[4] Apostolico, A. (1990), Optimal parallel detection of squares in strings; Part II: Detecting all squares, *CSD-TR-1012, purdue.*

stages. Note that in the second stage we have at most $\frac{m}{4}$ candidates. So we can achieve double progress with the same processors: In stage 1 we can verify $ds(2)$ and $ds(3)$.

At the start of a general stage, assume the leftmost and rightmost candidates are l and r and the candidates are positions aligned with elements of A_s (relative to l or r) for which $Ds(1), \ldots, Ds(s)$ have been verified. Since $|A_s| \leq \frac{m}{2^{s+1}}$, the m processors now verify $Ds(s+1), \ldots, ds(s+2^s)$ for all the candidates. For the purpose of efficient processor assignment they will be assigned to verify 2^s positions for *all* the elements in A_s. Those assigned to a non candidate simply do nothing. As above, we define matching candidates as the candidates for which all positions were verified as matches, l and r as the leftmost and rightmost matching candidates and the candidates surviving for the next stage are the matching candidates that are aligned with A_{s+2^s} (relative to l or r). Since the new value of s is larger than 2^s, we have at most $O(\log^* m)$ stages, each of which takes constant time. The number of processor is m.

This exponential acceleration phenomenon was called the *accelerating cascade* design principle by Cole and Vishkin [18] where by carefully choosing the parameters they were able to get an optimal $O(\log^* m)$ time parallel algorithms for another problem. For the complete description of the algorithm see Vishkin's paper [47].

6 Open questions

- String matching over a fixed alphabet. The lower bound of Section 4 assumes the input strings are drawn from a general alphabet and the only access to them is by comparisons. The lower and upper bounds for the string matching problem over a general alphabet are identical to those for comparison based maximum finding algorithm obtained by Valiant [45]. A constant time algorithm can find the maximum of integers in a restricted range [25] which suggests the possibility of a faster string matching algorithm.

- Faster randomized algorithm. The similarity to the maximum finding algorithm and the existence of a constant expected time randomized algorithm for that problem suggests the possibility of a faster randomized

with optimal number of processors.

One can use the deterministic sample to find all occurrences of the pattern in a text string in constant time and $O(n \log m)$ processors: for each position verify the deterministic sample for that position resulting in a few possible occurrences which can be verified in constant time using a linear number of processors.

We now show how to use the data structure constructed in the pattern analysis phase to search for all occurrences of the pattern starting in any position of a block of size $m/2$ of the text. We describe only an $O(\log^* m)$ version using m processors. An optimal implementation can be obtained using standard techniques.

Assume that the output of the pattern analysis phase is a sequence of compact arrays $A_0, A_1, \cdots, A_l$ where $A_0 = \{-k+1, \cdots, \frac{m}{2} - k\}$ is the set of all copies of the pattern considered at the start of the construction of the deterministic sample (relative to k, the copy that survived) and $A_i \subseteq A_0$ is the set of all copies remaining at the end of step i. These compact arrays can be generated in the same bounds of the pattern analysis described above and are used to efficiently assign processors to their tasks.

Initially all positions in the block are candidates for a potential occurrence and after each stage only part of the candidates will survive.

The algorithm starts with verifying $ds(1)$ for each candidate. This takes constant time using m processors. Call the candidates for which there is a match a matching candidate. Let l and r be the index of the leftmost and rightmost matching candidate respecively. Consider A_1 as a template around l and around r of all possible occurrences which have the same symbol in the column under $ds(1)$ relative to l or r. Note that since all other positions, even matching candidates (for which $ds(1)$ was just verified) cannot be real occurrences since they will disagree with the verified $ds(1)$ for position l or r. (The two templates cover the $\frac{m}{2}$ text positions under consideration, because there are no occurrences before l or after r and $l + \frac{m}{2} - k \geq r - k + 1$.) Thus, the candidates that survive for the next stage are those among the matching candidates aligned with a position in A_1 relative to l or r.

We can continue in this manner. In stage i there will be a set of candidates. The leftmost is l and the rightmost r and the set of candidates is aligned with the subset of A_i relative to l or r for which $Ds(1), \ldots, Ds(i)$ have been verified. (We described stage 0.) However this will take $\log m$

there are $\frac{m}{2}$ copies, for any two copies there is a witness to their mismatch.

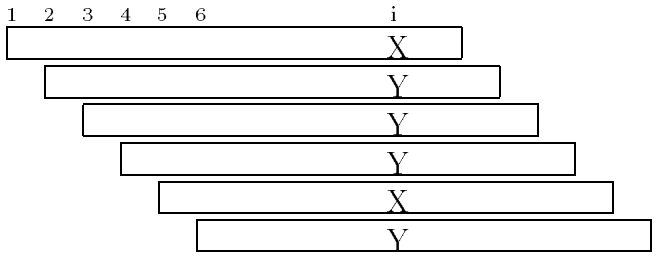

Figure 5.1. Aligned copies of the pattern and a column i.

Take the first and last copies and a witness to their mismatch. The column of the mismatch has at least two different symbols and thus one of the symbols in that column, in either the first or the last copy, appears in the column in at most half of the copies. Keep only the copies which have the same symbol in that column to get a set of at most half the number of original copies, which all have the same symbol at the witness column. This procedure can be repeated at most $\log m - 1$ times until there is a single copy left, say copy number k. Note that all columns chosen hit copy number k. The deterministic sample is the indices in copy number k of the columns considered. There are at most $\log m - 1$ such columns. If this sample is verified for position i of a text string no other occurrence at positions $i-k+1$ to $i - k + \frac{m}{2}$ is possible. □

One can find such a deterministic sample in parallel by the constructive proof of Lemma 5.1. Assume the witness information is produced by either Vishkin's [46] or Breslaur and Galil's algorithm [14] (It does not matter which algorithm since the time bound is dominated by the following steps.). There are $O(\log m)$ steps in the construction, each step counts how many symbols are equal to the witness symbol in the first and last copies, and can be implemented using Theorem 2.1 in optimal $O(\log m)$ time, or even faster by an algorithm of Cole and Vishkin [19] for prefix sums of small integers which works in optimal $O(\frac{\log m}{\log\log m})$ time.

Since the sums are taken at each round only for copies which are left, the total amount of operations performed at each round is half the number of operations of the previous round and sums up over all rounds to be linear. By Brent's Theorem the total time required for the pattern analysis is $O(\frac{\log^2 m}{\log\log m})$

characters, what implies that there would be no such occurrence. Thus, the lower bound holds also for any such string matching algorithm. □

This lower bound actually holds even if the algorithm is allowed to perform order comparison which can result in a *less than*, *equal* or *greater than* answers as shown in Breslauer and Galil's paper [15]. When the number of comparisons in each round is p and $n = O(m)$, the bound becomes $\Omega(\lceil \frac{m}{p} \rceil + \log\log_{\lceil 1+p/m \rceil} 2p)$, matching the upper bound.

5 A faster algorithm

The fast string matching algorithm of Vishkin [47] has two stages. The pattern analysis stage is slow and takes optimal $O(\frac{\log^2 m}{\log\log m})$ time while the text analysis is very fast and works in optimal $O(\log^* m)$ time. An alternative randomized implementation of the pattern analysis that works in optimal $O(\log m)$ time with very high probability will not be covered in this paper.

As we have seen before, we can assume without loss of generality that the pattern is shorter than twice its period length. Thus witnesses can be computed for all indices which are smaller than $\frac{m}{2}$.

Definition: A *deterministic sample* $DS = [ds(1), ds(2), \cdots, ds(l)]$ is a set of positions of the pattern string such that if the pattern is aligned at position i of the text and the symbols at positions $ds(1) \cdots ds(l)$ of the pattern are verified, that is $PATTERN[ds(j)] = TEXT[i + ds(j) - 1]$ for $1 \leq j \leq l$, then i is the only possible occurrence of the pattern in an interval of length $\frac{m}{2}$ around i.

Deterministic samples are useful since one can always find a small one.

Lemma 5.1: For any pattern of length m, a deterministic sample of size $\log m - 1$ exists.

Proof: We show how to find a deterministic sample of length $\log m - 1$. If this sample is verified for position i of the text then i is the only possible occurrence in an interval of length $\frac{m}{2}$ around i.

Consider $\frac{m}{2}$ copies of the pattern placed under each other, each shifted ahead by one position with respect to the previous one. Thus copy number k is aligned at position k of copy number one. Call the symbols of all copies aligned over position number i of the first copy *column* i (see figure 5.1). Since we assume that the pattern is shorter than twice its period length and

most $\lceil \frac{m}{k_{i+1}} \rceil \leq \frac{2m}{k_{i+1}}$ elements. By Lemma 4.5 and simple algebra the number of fixed elements satisfies:

$$\begin{aligned} f_{i+1} &\leq f_i + \frac{2m}{k_{i+1}} \frac{4mK_i \log m}{K_{i+1}} \\ &\leq K_i \left[1 + \left(\frac{m}{K_{i+1}} \right)^2 16 \log m \right] \\ &\leq m^{1-4^{-(i-1)}} (1 + m^{2 \cdot 4^{-i}} 16 \log m) \\ &\leq m^{1-4^{-i}} = K_{i+1}. \ \square \end{aligned}$$

Theorem 4.7: Any comparison-based parallel algorithm for finding the period length of a string $S[1..m]$ using m comparisons in each round requires $\frac{1}{4} \log \log m$ rounds.

Proof: Fix an algorithm which finds the period of S and let the adversary described above answer the comparisons. After $i = \frac{1}{4} \log \log m$ rounds $f_{i+1}, k_{i+1} \leq m^{1-4^{-\frac{1}{4} \log \log m}} = \frac{m}{\sqrt{\log m}} \leq \frac{m}{2}$. The adversary can still fix S to have a period length k_{i+1} by fixing each remaining residue class modulo k_{i+1} to the same symbol, different symbol for each class. Alternatively, the adversary can fix all unfixed symbols to different symbols. Note that this choice is consistent with all the the comparisons answered so far by invariants 3 and 4, and the string does not have any period length smaller than $\frac{m}{2}$. Consequently, any algorithm which terminates in less than $\frac{1}{4} \log \log m$ rounds can be fooled. $\square$

Theorem 4.8: The lower bound holds also for any comparison-based string matching algorithm when $n = O(m)$.

Proof: Fix a string matching algorithm. We present to the algorithm a pattern $P[1..m]$ which is $S[1..m]$ and a text $T[1..2m-1]$ which is $S[2..2m]$, where S is a string of length $2m$ generated by the adversary in the way described above (We use the same adversary that we used in the previous proof; the adversary sees all comparisons as comparisons between symbols in S.). After $\frac{1}{4} \log \log 2m$ rounds the adversary still has the choice of fixing S to have a period length smaller than m, in which case we will have an occurrence of P in T, or to fix all unfixed symbols to completely different

for different residue classes). The adversary answers comparisons between fixed symbols based on the value they are fixed to. All other comparisons involve two positions in different residue classes modulo k_{i+1} (and at least one unfixed symbol) and are always answered as unequal.

Since k_{i+1} is a multiple of k_i, the residue classes modulo k_i split; each class splits into $\frac{k_{i+1}}{k_i}$ residue classes modulo k_{i+1}. Note that if two indices are in different residue classes modulo k_i, then they are also in different residue classes modulo k_{i+1}; if two indices are in the same residue class modulo k_{i+1}, then they are also in the same residue class modulo k_i.

We show that the invariants still hold.

1. The candidate we chose for k_{i+1} was in the required range.

2. Residue classes which were fixed before split into several residue classes, all are fixed. Any symbol fixed at this round causes its entire residue class modulo k_{i+1} to be fixed to the same symbol.

3. Equal answers of previous rounds are not affected since the symbols involved were fixed to the same value by the invariants held before. Equal answers of this round are either between symbols which were fixed before to the same value or are within the same residue class modulo k_{i+1} and the entire residue class is fixed to the same value.

4. **a.** Unequal answers of previous rounds are between different residue classes modulo k_{i+1} since residue classes modulo k_i split. Unequal answers of this round are between different residue classes because comparisons within the same residue class modulo k_{i+1} are always answered as equal.

   b. Unequal answers which involve symbols which were fixed before this round are consistent because fixed values dictate the answers to the comparisons. Unequal answers which involve symbols that are fixed at the end of this round and at least one was fixed at this round are consistent since a new symbol is used for each residue class fixed.

5. We prove inductively that $f_{i+1} \leq K_{i+1}$. We fix at most $\frac{4mK_i \log m}{K_{i+1}}$ residue classes modulo k_{i+1}. There are k_{i+1} such classes and each class has at

$$\frac{1}{4}\frac{K_{i+1}}{k_i \log \frac{K_{i+1}}{k_i}} \geq \frac{K_{i+1}}{4K_i \log m}.$$

Each one of these candidates also satisfies the condition of Lemma 4.2 since $k_i \leq K_i$, $pk_i \geq \frac{K_{i+1}}{2}$ and

$$p^2 \geq \frac{1}{k_i}\frac{K_{i+1}^2}{4K_i} = \frac{1}{k_i}\frac{m^{2-2\cdot 4^{-i}}}{4m^{1-4^{-(i-1)}}} = \frac{m}{k_i}\frac{1}{4}m^{2\cdot 4^{-i}} \geq \frac{m}{k_i}. \ \square$$

Lemma 4.4: There exists a candidate for k_{i+1} in the range $\frac{1}{2}K_{i+1} \ldots K_{i+1}$ that forces at most $\frac{4mK_i \log m}{K_{i+1}}$ comparisons.

Proof: By Lemma 4.3 there are at least $\frac{K_{i+1}}{4K_i \log m}$ such candidates which are prime multiples of k_i and satisfy the condition of Lemma 4.2. By Lemma 4.2 each of the m comparisons is forced by at most one of them. So the total number of comparisons forced by all these candidates is at most m. Thus, there is a candidate that forces at most $\frac{4mK_i \log m}{K_{i+1}}$ comparisons. $\square$

Lemma 4.5: For m large enough and $i \leq \frac{1}{4}\log\log m$, $1 + m^{2\cdot 4^{-i}} 16 \log m \leq m^{3\cdot 4^{-i}}$.

Proof: For m large enough,

$$\log\log(1 + 16\log m) < \frac{1}{2}\log\log m = (1 - \frac{2}{4})\log\log m$$

$$\log(1 + 16\log m) < 4^{-\frac{1}{4}\log\log m}\log m$$

$$1 + 16\log m < m^{4^{-\frac{1}{4}\log\log m}} \leq m^{4^{-i}},$$

from which the lemma follows. $\square$

Lemma 4.6: Assume the invariants hold at the beginning of round i and the adversary chooses k_{i+1} to be a candidate which forces at most $\frac{4mK_i \log m}{K_{i+1}}$ comparisons. Then the adversary can answer the comparisons in round i so that the invariants also hold at the beginning of round $i + 1$.

Proof: By Lemma 4.4 such k_{i+1} exists. For each comparison that is forced by k_{i+1} and is of the form $S[l] = S[j]$ where $l \equiv j \mod k_{i+1}$ the adversary fixes the residue class modulo k_{i+1} to the same new symbol (a different symbol

3. If a comparison was answered as equal then both symbols compared were fixed to the same value.

4. If a comparison was answered as unequal, then

   a. it was between different residues modulo k_i;

   b. if the symbols were fixed then they were fixed to different values.

5. The number of fixed symbols f_i satisfies $f_i \leq K_i$.

Note that invariants 3 and 4 imply consistency of the answers given so far. Invariants 2, 3 and 4 imply that k_i is a possible period length: if we fix all symbols in each unfixed residue class modulo k_i to a new symbol, a different symbol for different residue classes, we obtain a string consistent with the answers given so far that has a period length k_i.

We start at round number 1 with $k_1 = K_1 = 1$. It is easy to see that the invariants hold initially. We show how to answer the comparisons of round i and how to choose k_{i+1} so that the invariants still hold. All multiples of k_i in the range $\frac{1}{2}K_{i+1} \ldots K_{i+1}$ are candidates for the new k_{i+1}. A comparison $S[l] = S[j]$ must be answered as equal if $l \equiv j \mod k_{i+1}$. We say that k_{i+1} *forces* this comparison.

Theorem 4.1 (see [43]): For large enough n, the number of primes between 1 and n denoted by $\pi(n)$ satisfies, $\frac{n}{\ln n} \leq \pi(n) \leq \frac{5}{4}\frac{n}{\ln n}$.

Corollary: The number of primes between $\frac{1}{2}n$ and n is greater than $\frac{1}{4}\frac{n}{\log n}$.

Lemma 4.2: If $p, q \geq \sqrt{\frac{m}{k_i}}$ and are relatively prime, then a comparison $S[l] = S[k]$ is forced by at most one of pk_i and qk_i.

Proof: Assume $l \equiv k \mod pk_i$, $l \equiv k \mod qk_i$ for $1 \leq l, k \leq m$. Then also $l \equiv k \mod pqk_i$. But $pqk_i \geq m$ and $1 \leq l, k \leq m$ which implies $l = k$, a contradiction. □

Lemma 4.3: The number of candidates for k_{i+1} which are prime multiples of k_i and satisfy $\frac{1}{2}K_{i+1} \leq k_{i+1} \leq K_{i+1}$ is greater than $\frac{K_{i+1}}{4K_i \log m}$. Each such candidate satisfies the condition of Lemma 4.2.

Proof: These candidates are of the form pk_i for prime p. The number of such prime values of p can be estimated using the corollary to Lemma 4.1. It is at least

Consider a CRCW-PRAM that solves the string matching problem over a general alphabet. In this case the PRAM can perform comparisons, but not computation, with its input symbols. Thus, its execution can be partitioned into comparison rounds followed by computation rounds. Therefore, a lower bound for the number of rounds in the parallel comparison model immediately translates into a lower bound for the time of the CRCW-PRAM. If the pattern is given in advance and any preprocessing is free, then this lower bound does not hold, as Vishkin's $O(\log^* m)$ algorithm shows. The lower bound also does not hold for CRCW-PRAM over a fixed alphabet strings. Similarly, finding the maximum in the parallel decision tree model has exactly the same lower bound [45], but for small integers the maximum can be found in constant time on a CRCW-PRAM [25].

We start by proving a lower bound for a related problem of finding the period length of a string. Given a string $S[1..m]$ we prove that $\Omega(\log\log m)$ rounds are necessary for determining whether it has a period length smaller than $\frac{m}{2}$. Later we show how this lower bound translates into a lower bound for string matching.

We show a strategy for an adversary to answer $\frac{1}{4}\log\log m$ rounds of comparisons after which it still has the choice of fixing the input string S in two ways: in one the resulting string has a period of length smaller than $\frac{m}{2}$ and in the other it does not have any such period. This implies that any algorithm which terminates in less rounds can be fooled.

We say that an integer k is a possible period length if we can fix S consistently with answers to previous comparisons in such a way that k is a period length of S. For such k to be a period length we need each residue class modulo k to be fixed to the same symbol, thus if $l \equiv j \mod k$ then $S[l] = S[j]$.

At the beginning of round i the adversary will maintain an integer k_i which is a possible period length. The adversary answers the comparisons of round i in such a way that some k_{i+1} is a possible period length and few symbols of S are fixed. Let $K_i = m^{1-4^{-(i-1)}}$. The adversary will maintain the following invariants which hold at the beginning of round number i:

1. k_i satisfies $\frac{1}{2}K_i \leq k_i \leq K_i$.

2. If $S[l]$ was fixed then for every $j \equiv l \mod k_i$ $S[j]$ was fixed to the same symbol.

$2k_i\frac{k_i}{k_{i-1}} = 2m$ processors.

2. If we succeed in producing witnesses for all the indices in the first block (all but the first for which there is no witness), compute witnesses in each following block of the same size using the optimal duel algorithm described above for the text processing. This takes $O(\log\log m)$ time only for the first stage. In the following stages, we will have at most $\sqrt{m}$ indices for which we have no witness, and duels can be done in O(1) time.

3. If we fail to produce a witness for some $2 \leq j \leq k_i$, it follows that $PATTERN(1 \cdots 2k_i)$ is periodic with period length p, where $p = j - 1$ and j is the smallest index of an yet-to-be-computed witness. By Lemma 2.1 all yet-to-be-computed indices within the first block are of the form $kp + 1$. Check periodicity with period length p to the end of the pattern. If p turns out to be the length of the period of the pattern, the pattern analysis is done and we can proceed with the text analysis. Otherwise, the smallest witness found is good also for all the indices of the form $kp + 1$ which are in the first k_i–block, and we can proceed with the duels as in 2.

If p processors are available and $m \leq p \leq m^2$, this algorithm can be modified to work in $O(\log\log_{\frac{2p}{m}} m)$ time. If the number of processors is smaller than $\frac{m}{\log\log m}$ the algorithm can be slowed down to work in $\frac{m}{p}$ time. When the number of processors is larger than n^2 the naive algorithm solves the problem in constant time. All these bounds can be summerized in one expression: $O(\lceil\frac{m}{p}\rceil + \log\log_{\lceil 1+p/m\rceil} 2p)$.

4 A lower bound

In this section we describe the lower bound of Breslauer and Galil [15] for a model which is similar to Valiant's parallel comparison tree model [45]. We assume the only access the algorithm has to the input strings is by comparisons which check whether two symbols are equal or not. The algorithm is allowed m comparisons in each round, after which it can proceed to the next round or terminate with the answer. We give a lower bound on the minimum number of rounds necessary in the worst case.

3 An $O(\log\log m)$ time algorithm

The $O(\log\log m)$ time algorithm of Breslauer and Galil [14] is similar to Vishkin's algorithm from the previous section. The method is based also on an algorithm for finding the maximum suggested by Valiant [45] for a comparison model and implemented by Shiloach and Vishkin [44] on a CRCW-PRAM.

As before, we have two stages. The first stage, the pattern analysis, computes the witness information which is used in the text analysis to find the actual occurrences. Let $\mathcal{P} = |u|$ be the period length of the pattern. As before if the pattern has a period length longer than half its length, we define $\mathcal{P} = \lceil \frac{m}{2} \rceil$.

Partition the text into blocks of size $\mathcal{P}$ and consider each one separately. In each block consider each position as a possible occurrence. Assuming we had $\mathcal{P}^2$ processors for each such block a duel can be made between all pairs of possible occurrences resulting with at most one occurrence in each block. Since we have only n processors, partition the blocks into groups of size $\sqrt{\mathcal{P}}$ and repeat recursively. The recursion bottoms out with one processor per block of size 1. When done we are left with one possible occurrence at most in each group of size $\sqrt{\mathcal{P}}$, thus $\sqrt{\mathcal{P}}$ possible occurrences all together. Then in constant time make all duels as described above. We are left with a single possible occurrence (or none) in each block of size $\mathcal{P}$ and proceed with counting the consecutive occurrences of the period described in section 2.

To make the text analysis run in optimal $O(\log\log m)$ time we start with an $O(\log\log \mathcal{P})$ time sequential algorithm which runs in parallel in all sub-blocks of length $\log\log \mathcal{P}$ leaving only $\frac{\mathcal{P}}{\log\log \mathcal{P}}$ possible occurrences in each block by performing duels. Then proceed with the procedure above starting with the reduced number of possible occurrences.

The pattern analysis can be done also in optimal $O(\log\log m)$ time. We describe here only an m processor algorithm. It works in stages and it takes at most $\log\log m$ stages. Let $k_i = m^{1-2^{-i}}$, $k_0 = 1$. At the end of stage i, we have at most one yet-to-be-computed witness in each block of size k_i. The only yet-to-be-computed index in the first block is 1.

1. At the beginning of stage i we have at most k_i/k_{i-1} yet-to-be-computed witnesses in the first k_i–block. Try to compute them using the naive algorithm on $PATTERN(1 \cdots 2k_i)$ only. This takes constant time using

Consider the first block of size 2^{i+1}. It has at most one other yet-to-be-computed witness, say $WITNESS[k]$. We first try to compute this witness by comparing two copies of the pattern shifted with respect to each other by $k-1$ positions. This can be easily done in constant time on an arbitrary CRCW-PRAM with m processors. If a witness is not found, then k is the period length of the pattern and the pattern analysis terminates. If a witness was found, a duel is performed in each block of size 2^{i+1} between the two yet-to-be-computed witnesses in each such block. It results in each block of size 2^{i+1} having at most one yet-to-be-computed witness. After $O(\lfloor \log m \rfloor)$ stages the witness information is computed for the first half of the pattern and the algorithm can proceed with the text analysis.

The optimal implementation of the pattern analysis is very similar to Galil's [26] original algorithm. Each iteration of the pattern analysis described above has actually two steps: the first step trys to verify a period length using a naive algorithm which compares long strings; if fails, a witness is found and it is used in a step which is identical to the actual string analysis phase.

Suppose the naive algorithm would be applied at stage i just to verify a period length of a prefix of the pattern of length 2^{i+1} instead of the whole pattern. If a mismatch is found it can be used as a witness as described before. If no mismatch has been found, continue to a *periodic* stage $i+1$ which will try to verify the same period length of a prefix of double length. At some point either a mismatch is found or the period length is verified for the whole string and the pattern analysis is terminated. If a mismatch was found, it follows from Lemma 2.1 that the first mismatch can be used as a witness value for all uncomputed witnesses in the first block; and the algorithm can catch up to stage $i+1$ (with the current value of i) by performing duels.

Galil's [26] original algorithm had only one stage which consisted of similar two steps; application of a naive algorithm to verify a period length of a prefix of the pattern of increasing length and elimination of close possible occurrences which would imply a short period length. The main difference is that Galil's algorithm had to compare long strings also in the steps Vishkin's algorithm uses the witness information. So n operations are performed at each round making the algorithm non-optimal. Galil [26] suggests an improvement for a finite alphabet which packs $O(\log m)$ symbols in a single integer and thus uses less processors to perform the comparisons, making an optimal implementation possible in $O(\log m)$ time.

At stage i, consider a block of size 2^{i+1} which consists of two blocks of size 2^i. It has at most two possible occurrences of the pattern, one in each block of size 2^i. A duel is performed between these two possible occurrences, leaving at most one in the 2^{i+1} block.

At the end of $\lfloor \log \mathcal{P} \rfloor$ stages, we are left with at most $\frac{2n}{|u|}$ possible occurrences of u which can be verified in constant-time using n processors. Note that the total number of operations performed is $O(n)$ and the time is $O(\log m)$. By Theorem 1.1 an optimal implementation is possible. Moreover, it is even possible to implement this phase on a CREW-PRAM within the same time bound. It is the pattern analysis phase which requires a CRCW-PRAM.

The pattern analysis phase is similar to the text analysis phase. It takes $\lfloor \log m \rfloor$ stages. The description below outlines a logarithmic time implementation using m processors.

The $WITNESS$ array which we used in the text processing stage is computed incrementally. Knowing that some witnesses are already computed in previous stages, one can easily compute more witnesses. Let i and j be two indices in the pattern such that $i < j \leq \lceil m/2 \rceil$. If $s = WITNESS[j - i + 1]$ is already computed then we can find at least one of $WITNESS[i]$ or $WITNESS[j]$ using a duel on the pattern as follows:

- If $s + i - 1 \leq m$ then $s + i - 1$ is also a witness either for i or for j.
- If $s + i - 1 > m$ then either s is a witness for j or $s - j + i$ is a witness for i (see figure 2.2).

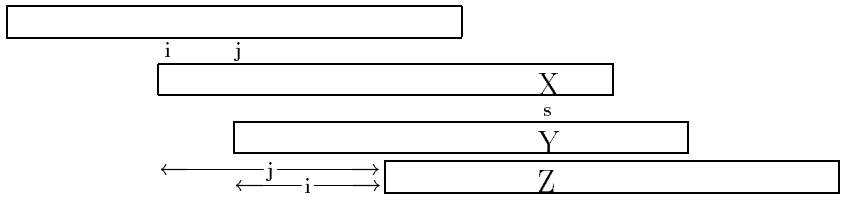

Figure 2.2. $X \neq Y$ and therefore we cannot have $Z = X$ and $Z = Y$.

The pattern analysis proceeds as follows: At stage i the pattern is partitioned into consecutive blocks of size 2^i. Each block has at most one yet-to-be-computed witness. The first block never has $WITNESS[1]$ computed.

be at least one mismatch. Vishkin [46] takes one of these mismatches and calls it a *witness* to the fact that i is not a period length. More formally, let k be the index of one such mismatch, then

$$PATTERN[k] \neq PATTERN[k-i].$$

We call this k a witness, and define

$$WITNESS[i+1] = k.$$

Using this witness information Vishkin suggests a method which he calls a *duel* to eliminate at least one of two close possible occurrences. Suppose i and j are possible occurrences and $0 < j-i < |u|$. Then, $r = WITNESS[j-i+1]$ is defined. Since $PATTERN[r] \neq PATTERN[r+i-j]$, at most one of them is equal to $TEXT[i+r-1]$ (see figure 2.1), and at least one of the possible occurrences can be ruled out (As in a real duel sometimes both can be ruled out.).

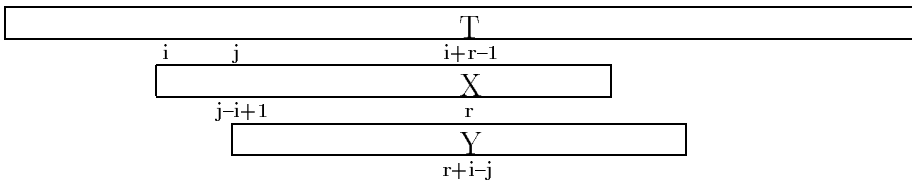

Figure 2.1. $X \neq Y$ and therefore we cannot have $T = X$ and $T = Y$.

Vishkin's algorithm [46] consists of two phases. The first is the pattern analysis phase in which the witness information is computed to help later with a text analysis phase which finds the actual occurrences.

We start with a description of the text analysis phase. Let $\mathcal{P} = |u|$ be the period length of the pattern. The pattern analysis phase described later computes witnesses only for the first half of the pattern. If the pattern has a period which is longer than half its length, we define $\mathcal{P} = \lceil \frac{m}{2} \rceil$.

The text analysis phase works in stages. There are $\lfloor \log \mathcal{P} \rfloor$ stages. At stage i the text string is partitioned into consecutive blocks of length 2^i. Each such block has only one possible start of an occurrence. We start at stage 0 where the blocks are of size one, and each position of the string is a possible occurrence.

$j - i$. Therefore, we cannot have occurrences of w at positions j and i if $0 < j - i < |u|$ and u is the period of the pattern. Clearly there are no more then $\frac{n}{|u|}$ occurrences of the pattern in a string of length n.

If the pattern is longer than twice its period length then instead of matching the whole pattern w we look only for occurrences of u^2, its period repeated twice. (Note that u^2 has the same period length as w by Lemma 2.2.) This case where the pattern is longer than twice its period is called the periodic case.

Assuming we could eliminate many of the occurrencesof u^2 and have only $n/|u|$ possible occurrences left, we can use the constant-time algorithm described above to verify these occurrences using only $2n$ processors. Then, by counting the number of consecutive matches of u^2, we can match the whole pattern.

Vishkin [46] shows how to count the consecutive matches in optimal $O(\log m)$ time on an EREW-PRAM using ideas which are similar to prefix computation. Breslauer and Galil [14] show how it can be done in constant optimal time on a CRCW-PRAM (and thus in optimal $O(\log m)$ time on an EREW-PRAM). Assume without loss of generality that the text is of length $n \leq \frac{3}{2}m$ and the pattern is $u^k v$ where u is its period length. Call an occurrence of u^2 at position i an *initial occurrence* if there is no occurrence of u^2 at position $i - |u|$ and a *final occurrence* if there is no occurrence at position $i + |u|$. There is at most one initial occurrence which can start an actual occurrence of the pattern: the rightmost initial occurrence in the first $\frac{m}{2}$ positions. Any initial occurrence in a position greater than $\frac{m}{2}$ cannot start an occurrence of the pattern since the text is not long enough. Any initial occurrence on the left cannot start an occurrence of the pattern since u, the period length of the pattern, is not repeated enough times. The corresponding final occurrence is the leftmost final occurrence to the right of the initial occurrence. By substructing the positions of the initial occurrence from the final occurrences and verifying an occurrence of v starting after the final occurrence, one can tell how many times the period is repeated and what are the actual occurrences of the pattern.

For the rest of the description we assume without loss of generality that the pattern is shorter than twice its period length, what is called the non periodic case.

Suppose u is the period of the pattern w. If we compare two copies of w shifted with respect to each other by i positions for $0 < i < |u|$, there must

It proceeds with $O(\log m)$ steps computing $n(i,2)$, $n(i,4)$, $\cdots n(i,2^j)$ for $j \leq \lfloor \log_2 m \rfloor$, by merging names of two 2^j-equivalence classes into a names of 2^{j+1}-equivalence classes. In another $O(\log m)$ steps it computes $n(i,m)$ by merging a subset of the names of power-of-two equivalence classes computed before, and reports all indices which are in the same m-equivalence class of the starting index of the pattern.

This algorithm requires $O(m^2)$ space which can be reduced to $O(m^{1+\epsilon})$ for a time tradeoff as described in the suffix tree construction algorithm of Apostolico et al. [6].

In the rest of this section we will describe the algorithm of Vishkin [46], on which the faster algorithms, described later, are based.

As we have seen before, if we have nm processor CRCW-PRAM, we can solve the string matching problem in constant time using the following method:

- First, mark all possible occurrences of the pattern as 'true'.
- To each such possible beginning of the pattern, assign m processors. Each processor compares one symbol of the pattern with the corresponding symbol of the text. If a mismatch is encountered, it marks the appropriate beginning as 'false'.

Assuming that we can eliminate all but l of the possible occurrences we can use the method described above to get a constant time parallel algorithm with lm processors. Both Galil [26] and Vishkin [46] use this approach. The only problem is that one can have many occurrences of the pattern in the text, even much more than the $\frac{n}{m}$ needed for optimality in the discussion above.

To overcome this problem, we introduce the notion of the period used in these two papers. A string u is called a *period* of a string w if w is a prefix of u^k for some positive integer k or equivalently if w is a prefix of uw. We call the shortest period of a string w *the period* of w. For example, the period of the string *abacabacaba* is *abac*. The string *abacabac* is also a period, so is the string *abacabacab*.

Lemma 2.2 (Lyndon and Schutzenberger [41]): If w has two periods of length p and q and $|w| \geq p + q$, then w has a period of length $\gcd(p,q)$.

If a pattern w occurs in positions i and j of some string and $0 < j - i < |w|$ then the occurrences must overlap. This implies that w has a period of length

matching can be adopted also for parallel string matching. Although the original algorithm worked in $O(n \log n)$ time, Kedem, Landau and Palem [34] were able to obtain an optimal $O(\log m)$ time parallel algorithm using a similar method. Chrochemore and Rytter [23] independently suggested a parallel implementation in $O(\log m)$ time using n processors. Another parallel algorithm which uses a similar method is the suffix tree construction algorithm of Apostolico et al. [6] which can also be used to solve the string matching problem. All these parallel algorithms need an arbitrary CRCW-PRAM and are for fixed alphabets; they also need a large memory space. It seems that this method cannot be used to obtain faster than $O(\log n)$ string matching algorithms, however it is applicable to other problems [23].

We describe a logarithmic time implementation of the Karp, Miller and Rosenberg [31] method for an n-processor arbitrary CRCW-PRAM. Consider the input as one string of length $l = n + m$ which is a text of length n concatenated with a pattern of length m. Two indices of the input string are called k-equivalent if the substring of length k starting at those indices are equal; this is in fact an equivalence relation on the set of indices of the input string. The algorithm assigns unique names to each index in the same equivalence class. The goal is to find all indices which are in the same m-equivalence class of the index where the pattern starts.

We denote by $n(i, j)$ the unique name assigned to the substring of length j starting at position i of the input string; assume $n(i, j)$ is defined only for $i + j \leq l + 1$ and the names are integers in the range $1 \cdots l$. Suppose $n(i, r)$ and $n(i, s)$ are known for all positions i of the input string. One can easily combine these names to obtain $n(i, r + s)$ for all positions i in constant time using l processors as follows: Assume a two dimensional array of size $l \times l$ is available; assign a processor to each position of the input string. Note that the string of length $r + s$ starting at position i is actually the string of length r starting at position i concatenated with the string of length s starting at position $i + r$. Each processor will try to write the position number it is assigned to in the entry at row $n(i, r)$ and column $n(i + r, s)$ of the matrix. If more then one processors attempts to write the same entry, assume an arbitrary one succeeds. Now $n(i, r + s)$ is assigned the value written in row $n(i, r)$ and column $n(i + r, s)$ of the matrix. That is, $n(i, r + s)$ is an index of the input string, not necessarily i, which is $(r + s)$-equivalent to i.

The algorithm start with $n(i, 1)$ which is the symbol at position i of the string, assuming the alphabet is the set of integers between 1 and l.

$$f(s_1 s_2 \cdots s_i) = f(s_1) f(s_2) \cdots f(s_i)$$

Most of the work in the algorithm is performed using a well known method for parallel prefix computation summarized in the following theorem:

Theorem 2.1 (Folklore, see also [37]): Suppose a sequence of n elements $x_1, x_2, \cdots, x_n$ are drawn from a set with an associative operation $*$, computable in constant time. Let $p_i = x_1 * x_2 * \cdots x_i$, usually called a prefix sum. Then an EREW-PRAM can compute all $p_i \quad i = 1 \cdots n$, in $O(\log n)$ time using $\frac{n}{\log n}$ processors.

Karp and Rabin's algorithm multiplies the matrices representing the pattern to get a single matrix which is called the *fingerprint* of the pattern. By Theorem 2.1 this can be done by a $\frac{m}{\log m}$-processor EREW-PRAM in $O(\log m)$ time. The text string is also converted to the same representation and matches can be reported based only on comparison of two matrices; the fingerprint of the pattern and the fingerprint of each text position. To compute the fingerprint of a text position j, which is the product of the matrix representation of the substring starting at position j and consisting of the next m symbols, first compute all prefix products for the matrix representation of the text and call them P_i. Then compute the inverse of each P_i; the inverse exists since each P_i is a product of invertible matrices. The fingerprint for a position j, $2 \leq j \leq n - m + 1$ is given by $P_{j-1}^{-1} P_{j+m-1}$; the finger print of the first position is P_m. By Theorem 2.1 the prefix products also take optimal $O(\log m)$ time on an EREW-PRAM. Since the remaining work can be done in constant optimal time, the algorithm works in optimal $O(\log m)$ total time.

However, there is a problem with the algorithm described above. The entries of those matrices may grow too large to be represented in a single register; so the numbers are truncated modulo some random prime p. All computations are done in the field $\mathcal{Z}_p$ which assures that the matrices are still invertible.

This truncated representation does not assure uniqueness, but Karp and Rabin show that the probability of their algorithm erroneously reporting a nonexisting occurrence is very small if p is chosen from a range which is large enough. This algorithm is in fact the only parallel algorithm which works in optimal logarithmic time on an EREW-PRAM; all the algorithms we describe later need a CRCW-PRAM.

The method used by Karp, Miller and Rosenberg [31] for sequential string

in $O(\log m)$ time using $\frac{n}{\log m}$ processors. This algorithm works for fixed alphabet and was later improved by Vishkin [46] for general alphabet. Optimal algorithms by Karp and Rabin [32] and other algorithms based on Karp, Miller and Rosenberg's [31] method [23, 34] also work in $O(\log m)$ time for fixed alphabet. Breslauer and Galil [14] obtained an optimal $O(\log\log m)$ time algorithm for general alphabet. Vishkin [47] developed an optimal $O(\log^* m)$[1] time algorithm. Unlike the case of the other algorithms this time bound does not account for the preprocessing of the pattern. The preprocessing in Vishkin's algorithm takes $O(\frac{\log^2 m}{\log\log m})$. Vishkin's super fast algorithm raised the question whether an optimal constant-time algorithm is possible. This question was partially settled in a recent paper by Breslauer and Galil [15] showing an $\Omega(\log\log m)$ lower bound for parallel string matching over a general alphabet. The lower bound proves that a slower preprocessing is crucial for Vishkin's algorithm.

This paper is organized as follows. In Section 2 we describe the logarithmic time algorithms. Section 3 is devoted to Breslauer and Galil's $O(\log\log m)$ time algorithm. Section 4 covers the matching lower bound. Section 5 outlines the ideas in Vishkin's $O(\log^* m)$ algorithm. In some cases we will describe a parallel algorithm that achieves the claimed time bound using n processors. The optimal version, using $O(\frac{n}{t})$ processors, can be derived using standard techniques. Many questions are still open and some are listed in the last section of this paper.

2 Logarithmic time algorithms

The simplest parallel string matching algorithm is probably the randomized algorithm of Karp and Rabin [32]. The parallel version of their algorithm assumes the alphabet is binary and translates the input symbols into a 2×2 non-singular matrices. The following representation is used, which assures a unique representation for any string as a product of the matrices representing it.

$$f(0) = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \qquad\qquad f(1) = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

[1]The function $\log^* m$ is defined as the smallest k such that $\log^{(k)} m \leq 2$, where $\log^{(1)} m = logm$ and $\log^{(i+1)} m = \log\log^{(i)} m$.

sometimes use the more powerful priority model.

For the algorithms discussed in this paper we assume that the length of the text string is $n = 2m$, where m is the length of the pattern string. This is possible since the text string can be broken into overlapping segments of length $2m$ and each segment can be searched in parallel.

Lower bounds for some basic parallel computational problems can be applied to string matching. A lower bound of $\Omega(\frac{\log n}{\log \log n})$ for computing the parity of n input bits on a CRCW-PRAM with any polynomial number of processors [7] implies that one cannot count the number of occurrences faster than $O(\frac{\log n}{\log \log n})$. Another lower bound of $\Omega(\log n)$ for computing a Boolean AND of n input bits on any CREW-PRAM [20] implies an $\Omega(\log n)$ lower bound for string matching in this parallel computation model.

These lower bounds make the possibility of sublogarithmic parallel algorithms for any problem very unlikely. However several problems are known to have such algorithms [8, 9, 11, 36, 44, 45] including string matching. In fact, a very simple algorithm can solve the string matching problem in constant time using nm processors on a CRCW-PRAM: similarly to the naive sequential algorithm, consider each possible start of an occurrence. Assign m processors to each such position to verify the occurrence. Verifying an occurrence is simple; perform all m comparisons in parallel and any mismatch changes a value of the $MATCH$ array to indicate that an occurrence is impossible.

The following theorem will be used throughout the paper.

Theorem 1.1 (Brent [13]): Any PRAM algorithm of time t that consists of x elementary operations can be implemented on p processors in $\lceil x/p \rceil + t$ time.

Using this theorem for example, we can slow down the constant-time algorithm describe above to run in $O(s)$ time on $\frac{nm}{s}$ processors.

In the design of a parallel algorithm, one is also concerned about the total number of operations performed, which is the time-processor product. The best one can wish for is the number of operations performed by the fastest sequential algorithm. A parallel algorithm is called *optimal* if that bound is achieved. Therefore, in the case of the string matching problem, an algorithm is optimal if the time-processor product is linear in the length of the input strings.

An optimal parallel algorithm discovered by Galil [26] solves the problem

the string matching problem. Namely, the exact number of comparisons necessary in the case of a general alphabet. Surprisingly, the upper bound of about $2n$ comparisons, the best known before [5, 35], was improved to $\frac{4}{3}n$ comparisons by Colussi, Galil and Giancarlo [16]. In a recent work Cole [17] proved that the number of comparisons performed by the original Boyer-Moore algorithm is about $3n$.

In this paper we will focus on parallel algorithms for the string matching problem. Many other related problems have been investigated and are out of the scope of this paper [1, 27]. For an introduction to parallel algorithms see surveys by Karp and Ramachandran [33] and Eppstein and Galil [24].

In parallel computation, one has to be more careful about the definition of the problem. We assume the the input strings are stored in memory and the required output is a Boolean array $MATCH[1..n]$ which will have a 'true' value at each position where the pattern occurs and 'false' where there is no occurrence.

All algorithms considered in this paper are for the parallel random access machine (PRAM) computation model. This model consists of some processors with access to a shared memory. There are several versions of this model which differ in their simultaneous access to a memory location. The weakest is the exclusive-read exclusive-write (EREW-PRAM) model where at each step simultaneous read operation and write operations at the same memory location are not allowed. A more powerful model is the concurrent-read exclusive-write (CREW-PRAM) model where only simultaneous read operations are allowed. The most powerful model is the concurrent-read concurrent-write (CRCW-PRAM) model where read and write operations can be simultaneously executed.

In the case of the CRCW-PRAM model, there are several ways of how write conflicts are resolved. The weakest model, called the *common* CRCW-PRAM assumes that when several processors attempt to write to a certain memory location simultaneously, they all write the same value. A stronger model called the *arbitrary* CRCW-PRAM assumes an arbitrary value will be written. An even stronger model, the *priority* CRCW-PRAM assumes each processor has a priority and the highest priority processor succeeds to write. Most of the CRCW-PRAM algorithms described in this paper can be implemented in the common model. In fact these algorithms can be implemented even if we assume that the same constant value is always used in case of concurrent writes. However, to simplify the presentation we will

of the pattern can start only at these positions. The algorithm checks each of these positions for an occurrence of the pattern. Since it can take up to m comparisons to verify that there is actually an occurrence, the time complexity of this algorithm is $O(nm)$. Note that the only operations involving the input strings in this algorithm are comparisons of two symbols.

The only assumption we made about the alphabet in the algorithm described above is that alphabet symbols can be compared and the comparison results in an equal or unequal answer. This assumption, often referred to as the *general alphabet* assumption, is the weakest assumption we will have on the alphabet, and, as we have seen, is sufficient to solve the string matching problem. However, although the definition of the string matching problem does not require the alphabet to be ordered, an arbitrary order is exploited in several algorithms [3, 4, 21, 22] which make use of some combinatorial properties of strings over an ordered alphabet [40]. This assumption is reasonable since the alphabet symbols are encoded numerically, which introduces a natural order. Other algorithms use a more restricted model where the alphabet symbols are small integers. Those algorithms usually take advantage of the fact that symbols can be used as indices of an array [2, 6, 42, 48] or that many symbols can be packed together in one register [26]. This case is usually called *fixed alphabet*.

Many sequential algorithms exist for the string matching problem and are widely used in practice. The better known are those of Knuth, Morris and Pratt [35] and Boyer and Moore [12]. These algorithms achieve $O(n + m)$ time which is the best possible in the worse case and the latter algorithm performs even better on average. Another well known algorithm which was discovered by Aho and Corasik [2] searches for multiple patterns over a fixed alphabet. Many variations on these algorithms exist and an excellent survey paper by Aho [1] covers most of the techniques used.

All these algorithms use an $O(m)$ auxiliary space. At a certain time it was known that a logarithmic space solution was possible [28], and the problem was conjectured to have a time-space trade off [10]. This conjecture was later disproved when a linear-time constant-space algorithm was discovered [29] (see also [21]). It was shown that even a 6-head two-way finite automaton can perform string matching in linear time. It is still an open problem whether a k-head one-way finite automaton can perform string matching. The only known cases are for $k = 1, 2, 3$ [30, 38, 39] where the answer is negative.

Recently, few papers have been published on the exact complexity of

Parallel string matching algorithms*

Dany Breslauer†
Columbia University

Zvi Galil
Columbia University and
Tel-Aviv University

CUCS-002-92

Abstract

The string matching problem is one of the most studied problems in computer science. While it is very easily stated and many of the simple algorithms perform very well in practice, numerous works have been published on the subject and research is still very active. In this paper we survey recent results on parallel algorithms for the string matching problem.

1 Introduction

You are given a copy of Encyclopedia Britanica and a word and requested to find all the occurrences of the word. This is an instance of the string matching problem. More formally, the input to the string matching problem consists of two strings $TEXT[1..n]$ and $PATTERN[1..m]$; the output should list all occurrences of the pattern string in the text string. The symbols in the strings are chosen from some set which is called an *alphabet*. The choice of the alphabet sometimes allows us to solve the problem more efficiently as we will see later.

A naive algorithm for solving the string matching problem can proceed as follows: Consider the first $n-m+1$ positions of the text string. Occurrences

*Work partially supported by NSF Grant CCR-90-14605.
†Partially supported by an IBM Graduate Fellowship.