Theoretical Computer Science 88 (1991) 59–82
Elsevier

Usefulness of the Karp–Miller–Rosenberg algorithm in parallel computations on strings and arrays*

Maxime Crochemore
LITP, Institut Blaise Pascal, University of Paris 7, 2 Place Jussieu, F-75251, Paris Cedex 05, France

Wojciech Rytter
Institute of Informatics, Warsaw University, ul. Banacha 2, 00–913 Warsaw 59, Poland

Communicated by D. Perrin
Received October 1989

Abstract

Crochemore, M. and W. Rytter, Usefulness of the Karp–Miller–Rosenberg algorithm in parallel computations on strings and arrays, Theoretical Computer Science 88 (1991) 59–82.

The Karp–Miller–Rosenberg (1972) algorithm was one of the first efficient (almost linear) sequential algorithms for finding repeated patterns and for string matching. In the area of efficient sequential computations on strings it was soon superseded by more efficient (and more sophisticated) algorithms. We show that the Karp–Miller–Rosenberg algorithm (KMR) must be considered as a basic technique in parallel computations. For many problems, variations of KMR give the (known) most efficient parallel algorithms. The representation of the set of basic factors (subarrays) of a string (array) produced by the algorithm is an extremely useful data structure in parallel algorithms on strings and arrays. This gives also a general unifying framework for a large variety of problems. We show that the following problems for strings and arrays can be solved by almost optimal parallel algorithms: pattern-matching, longest repeated factor (subarray), longest common factor (subarray), maximal symmetric factor (subarray). Also the following problems for strings can be solved within the same complexity bounds: finding squares, testing even palstars and compositions of k palindromes for $k=2$, 3, 4, computing Lyndon factorization and building minimal pattern-matching automata. In the model without concurrent writes the parallel time is $O(\log(n)^2)$ (with n processors) and in the model with concurrent writes the time, for most of the problems, is $O(\log(n))$ (with n processors). For two problems related to the one-dimensional case (longest repeated factor and longest common factor) there were designed parallel algorithms using suffix trees (Apostolico et al. 1988). However, our data structure is simpler and, furthermore, for the two-dimensional case suffix

*This work was done at the Dept. Math. Informatique, Université Paris-Nord, France.

trees do not work. The complexity of our algorithms does not depend on the size of the alphabet, except for the computation of pattern-matching automata.

1. Introduction

The Karp–Miller–Rosenberg algorithm deals with identification of well-structured objects: this means that the identifier of the object A of size $2n$ can be easily computed from identifiers of few subobjects of A of size n. Examples of well-structured objects are strings, arrays and trees. The algorithm is applied to find repeated objects: two objects which are equal. This is equivalent to finding two objects with the same identifier. Hence, the identification procedure is the crucial idea of the algorithm. The computations are well structured due to the fact that the data are well structured. The identifiers of objects of size 1 are first computed, then of sizes 2, 4, 8 etc. The identifiers of objects whose size is not a power of two can be computed by decomposing these objects into objects whose sizes are powers of two. We have a recursion of depth $O(\log(n))$ with independent recursive calls. Such a type of sequential computation is ideally suited to efficient parallel computation.

In the case of strings the subobjects are factors of a string. The Karp–Miller–Rosenberg algorithm creates names for all factors whose size is a power of two; the created data structure is called the *dictionary of basic factors*. This dictionary can be computed by an efficient parallel algorithm: a parallel version of the Karp–Miller–Rosenberg algorithm. This gives, as an application, a series of efficient parallel algorithms for strings and arrays.

By an efficient parallel algorithm we mean an algorithm working in polylogarithmic time with a polynomial number of processors. In this class, especially, efficient algorithms are optimal and almost optimal parallel algorithms. An algorithm (working in polylogarithmic parallel time) is *optimal* iff its total number of elementary operations is linear. An *almost optimal algorithm* is one which is optimal within a polylogarithmic factor (polylogarithmic time, with linear number of processors). From the practical point of view, optimal or almost optimal are not very different. Our basic model of parallel computation is a parallel random access machine without write conflicts (PRAM). However, a model with write conflicts (concurrent writes) will also be discussed.

The parallel random access machine (PRAM) is a parallel version of the random access machine used as a standard model for presentation of sequential algorithms. The PRAM consists of a number of processors working synchronously and communicating through the common random access memory. Each processor is a random access machine with usual operations. The processors are indexed by the consecutive natural numbers and they synchronously execute the same central program; however, the action of a given processor depends also on its number (known to the processor). In one step a processor can access one memory location. The models differ in regard to simultaneous access of the same memory location by more than one processor. We use the following conventions: any number of processors can read from the same memory location simultaneously but write conflicts are not allowed: no two processors can attempt to write simultaneously into the same location.

The parallelism will be expressed by the following type of parallel statement:

for all x in X **in parallel do** action(x).

The execution of this statement consists of:

(a) assigning a processor to each element of X;

(b) executing in parallel by the assigned processors all those operations specified by action(x).

We are interested in parallel time $T(n)$ as well as the number $P(n)$ of processors employed by the parallel algorithm. The product $T(n)P(n)$ gives the total number of operations.

The PRAM model is best suited to work with tree-structured objects or tree-like (recursive) structured computations. We start with this type of computation. One of the basic parallel operations is the (so-called) prefix computation. Given a vector x of n values the problem is to compute all prefix products: $y[1]=x[1]$, $y[2]=x[1]\otimes x[2]$, $y[3]=x[1]\otimes x[2]\otimes x[3], \ldots$ The prefix computation means computing the values of $y[1]$, $y[2]$, $y[3], \ldots$ The operation $\otimes$ is assumed to be associative and computable in O(1) time. We use frequently the following fact (see e.g. [19]).

Lemma 1.1. *Prefix computation applied to a vector x of size n can be done in* $O(\log(n))$ *time with* $n/\log(n)$ *processors.*

Prefix computation will be used in this paper, for instance, to compute the set of maximal (in the sense of set inclusion) subintervals (if there are given $O(n)$ subintervals of $[1\,..\,n]$).

One of the basic parallel methods to construct efficient algorithms is the (so-called) *doubling technique*. Roughly speaking, it consists in computing, in subsequent steps, object of twice the size as in the previous step. The word "doubling" is often misleading because in many algorithms of this type, the size of objects grows with ratio $c>1$ and not necessarily with $c=2$. The typical use of this technique is in the proof of the following lemma (see [19]). Suppose we have a vector of size n with some of the positions marked. Denote by Minright[i] the nearest marked position to the right of position i. Similarly, Maxleft[i] denotes the nearest marked position to the left of position i.

Lemma 1.2. *If we have a vector of size n with some of the nodes marked then we can compute vectors Minright, Maxleft in* $O(\log(n))$ *time with* $n/\log(n)$ *processors.*

The doubling technique is also the crucial feature of the structure of the Karp–Miller–Rosenberg algorithm (KMR) in a parallel setting. In one stage the algorithm computes the names (identifiers) for all words of size k. In the next stage using these names it computes names of words of twice the size. To make a parallel version of KMR, it is enough to design an efficient parallel version of one stage; this essentially reduces to the parallel computation of the procedure RENUMBER(x) whose aim is to assign names to objects. If this procedure is implemented in $T(n)$ parallel time with n processors then we have a parallel version of the algorithm KMR working in $O(T(n)\log(n))$ time with the same number n of processors. This is due to the doubling technique and the fact that there are only $O(\log(n))$ stages. Essentially, the same problems as these computed by a sequential algorithm can be computed by its parallel version in $O(T(n)\log(n))$ time.

The complexity of computing RENUMBER(x) depends heavily on the model of parallel computation used. It can be computed in time $O(\log(n))$ without concurrent writes and in constant time with concurrent writes. In the latter case one needs a memory bigger than the total number of operations (auxiliary table with n^2 entries or, by making some arithmetic tricks, with $n^{1+\varepsilon}$ entries). This looks slightly artificial, although entries of auxiliary memory do not have to be initialized. The details related to the distribution of processors are also very technical in the case of the concurrent-write model. Therefore, we present our algorithms using a model without concurrent writes. This increases the time by a logarithmic factor. This logarithmic factor gives also a big margin of time for technical problems related to the assignment of processors to the elements to be processed. We indicate shortly how to remove this logarithmic factor when concurrent writes are used. The main difference is the implementation of the procedure RENUMBER and computation of equivalent classes (classes of objects with the same name).

2. The Karp–Miller–Rosenberg algorithm and the dictionary of basic factors (basic subarrays).

Given a string t we say that two positions are *k-equivalent* iff the factors of length k starting at these positions are equal. Such an equivalence is best representated by assigning at each position a name to the factor of length k starting at this position. We shall compute names of all factors of a given length k for $k=1, 2, 4, \ldots$ We consider only factors whose length is a power of two. Such factors are called *basic factors*. The *name of a factor* is its rank in the lexicographic ordering of factors of a given length. We also call these names k-letters. Factors of length k and their corresponding k-letters can be thought of as the same object. For each k-letter r we also require (for further applications) a link pos$[r, k]$ to any one position at which an occurrence of the k-letter r starts.

We consider only factors starting at positions $[1\,..\,n-1]$. The nth position contains the special endmarker #. The endmarker has highest rank in the alphabet. We can generally assume without loss of generality (w.l.o.g.) that $n-1$ is a power of two. The tables "name" and "pos" are together called the *dictionary of basic factors*. This dictionary is our basic data structure.

	i =	1	2	3	4	5	6	7	8	
	text =	a	b	a	a	b	b	a	a	#######
k=1	name[i,k]=	1	2	1	1	2	2	1	1	
k=2	name[i,k]=	2	4	1	2	5	4	1	3	
k=4	name[i,k]=	3	6	1	4	8	7	2	5	

	r =	1	2	3	4	5	6	7	8
k=1	pos[r,k]=	1	2	undefined					
k=2	pos[r,k]=	3	1	8	2	5	undefined		
k=4	pos[r,k]=	3	7	1	4	8	2	6	5

Fig. 1. The dictionary of basic factors: the tables of names and positions of k-letters. The k-letter at position i is the word $t[i\,..\,i+k-1]$; its name is its rank according to lexicographic ordering of all factors of length k (order of symbols is: $a<b<\#$). Integers k's are powers of two. The tables can be stored in O(n log(n)) memory.

We demonstrate our data structure on the following example string $t=$ *abaabbaa*#. A further six #'s are appended to guarantee that each factor x of length 8 starting in $[1\,..\,8]$ is well defined. In Fig. 1 tables "name" and "pos" are presented for our example text *abaabbaa* # # # # # # #. In particular, the entries of pos[$*$, 4] give the lexicographically sorted sequence of factors of length 4. This is the sequence of factors of length 4 starting at positions 3, 7, 1, 4, 8, 2, 6, 5. Hence, the lexicographic ordering factors of length 4 is:

aabb, *aa*# #, *abaa*, *abba*, *a*# # #, *baab*, *baa*# *bbaa*.

In the case of arrays, basic factors are $k*k$ subarrays, where k is a power of two. In this situation, name[(i,j), k] is the name of a $k*k$ subarray t' of a given array t with upper-left corner at position (i,j). We will discuss mostly the construction of dictionaries of basic factors for strings; the construction in the two-dimensional case is an easy extension of that for the one-dimensional data.

We first present a simple application to make some acquaintance with dictionary of basic factors. The table PREF is a close "relative" of the failure table P commonly used in efficient string-matching sequential algorithms and automata. The name "failure" comes from the way of using the table: it is used in situations when a failure (mismatch) occurs.

$$\mathrm{PREF}[i]=\max\{j: p[i\,..\,i+j-1] \text{ is a prefix of } p\},$$

where p is a pattern and the failure table P is defined as follows:

$$P[i]=\max\{0\leqslant j<i: p[1\,..\,j] \text{ is a suffix of } p[1\,..\,i]\}.$$

This table PREF and related table SUF are crucial tables in the Main–Lorentz square-finding algorithm, which we later parallelize.

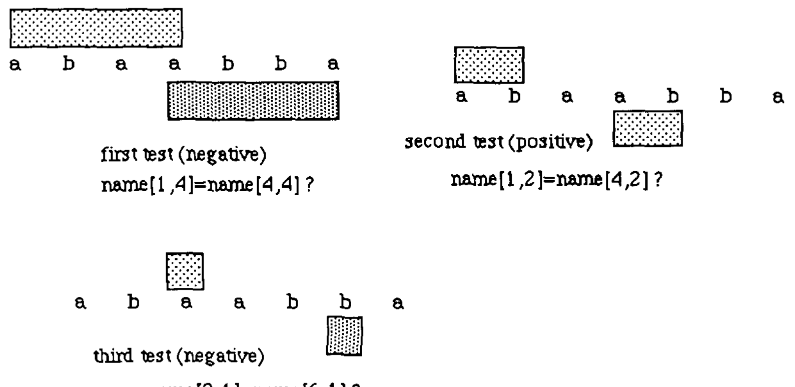

Fig. 2. The computation of PREF[4] using a binary search (log(8) tests). PREF[4] = 2.

Theorem 2.1. *The table* PREF *can be computed in* O(log(n)) *time with n processors without using concurrent writes if the dictionary of basic factors is already computed.*

Proof. Essentially, it is enough to show how to compute table PREF within our complexity bounds. One processor is assigned to each position i and computes PREF[i] using a kind of binary search, as indicated in Fig. 2. This completes the proof. □

Let x be a vector or an array of total size n containing elements of some linearly ordered set. The procedure RENUMBER assigns new values to the entries of x. Let val(x) be the set of values of all entries of x. Let x' be the value of x after performing RENUMBER(x). We require:

(a) if $x[i]=x[j]$ then $x'[i]=x'[j]$, and

(b) val(x') is a subset of $[1..n]$.

There are two variations of the procedure depending on whether the following condition is also satisfied:

(c) $x'[i]$ is the rank of $x[i]$ in the set val(x).

We require also that the procedure computes as a side-effect the vector POS: if q is in val(x') then POS[q] is any position i such that $x'[i]=q$.

Lemma 2.2. (1) *The procedure* RENUMBER *satisfying* (a), (b) *and* (c) *can be computed in* O(log(n)) *time with n processors on an exclusive-write* PRAM;

(2) *Assume that* val(x) *consists of integers or pairs of integers in the range* $[1..n]$. *Then the procedure* RENUMBER *satisfying* (a) *and* (b) *can be computed in* O(1) *time with n processors on a concurrent-write* PRAM. *In this case the size of auxiliary memory is bigger than the total number of operations: it is* $O(n^{1+\varepsilon})$. *However, auxiliary tables do not have to be initialized.*

Proof. (1) The main part of the procedure is the parallel sort. We explain the action of the RENUMBER on the following example: $x=[(1,2),(3,1),(2,2),(1,1),(2,3),(1,2)]$. We create the vector x' of composite entries $x'[i]=(x[i],i)$. The entries of x' are lexicographically sorted. The parallel merge sort algorithm of Cole [8] can be applied ($O(\log(n))$ time, n processors). So for our example we get the sequence

$$((1,1),4),\,|((1,2),1),\,((1,2),6),\,|\,((2,2),3),\,|\,((2,3),5)\,|\,((3,1),2).$$

We partition this sequence into groups of equal elements except for their last component. These groups are consecutively numbered. This can be easily done using a prefix computation. Then taking the last component i of each element, we set $x[i]$ to the number associated with its group. So, in our example $i=4$ is in the first group, $i=1,6$ are in the second group, etc. We have five groups:

$$x[4]:=1,\quad x[1]:=2,\quad x[6]:=2,\quad x[3]:=3,\quad x[5]:=4,\quad x[2]:=5.$$

This renumbering can be done in $O(1)$ parallel time once we know to which group each element belongs. Doing so, the whole procedure RENUMBER has the same complexity as sorting n elements. This completes the proof of point (1).

(2) Assume that val(x) consists of pairs of integers in the range $[1..n]$. In fact, RENUMBER will be used by us mostly in such cases. We use an $n*n$ auxiliary table BB. This table is called the bulletin board, see [16, 4]. The processor at position 1 writes its name into the entry (p,q) of table BB, where $(p,q)=x[1]$. We have many concurrent writes here because many processors attempt to write their positions into the same entry of the bulletin board. We assume that one (e.g. the one with smallest position) of them succeeds and writes its position i into the entry $BB[p,q]$. Then for each position j the processors set $x[j]$ to the value of $BB[p,q]$, where (p,q) is the old value of $x[j]$. The computation is graphically illustrated in Fig. 3. The time is constant. In this moment we use quadratic memory; however, by applying some arithmetic tricks (see [4]), it can be reduced to $O(n^{1+\varepsilon})$. This completes the proof. □

Theorem 2.3. *The dictionary of basic factors (basic subarrays) of a given string (array) can be computed with n processors in* $O(\log(n)^2)$ *time on an exclusive-write* PRAM *and*

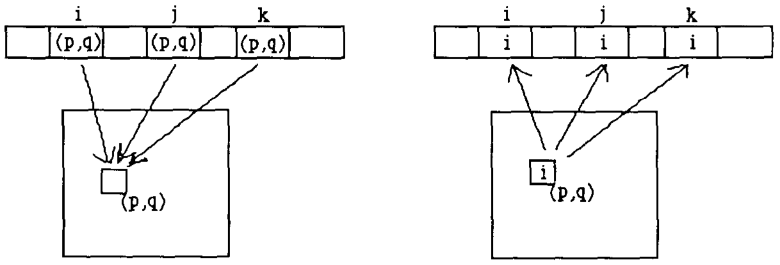

Fig. 3. The use of the bulletin board. $x[i]=x[j]=x[k]=(p,q)$. Then the values of x at positions i,j,k are changed, in two parallel steps (with concurrent writes), to the same position i.

in O(log(n)) *time on a concurrent-write* PRAM (*in the latter case the auxiliary memory of size* O($n^{1+\varepsilon}$) *is used*).

Proof. We start with an algorithm for strings. The crucial fact is now the following simple observation:

$$(*)\qquad \text{name}[i, 2k] = \text{name}[j, 2k] \text{ iff } (\text{name}[i, k] = \text{name}[j, k]) \text{ and } (\text{name}[i+k, k]\ \text{name}[j+k, k]).$$

Algorithm KMR

```
/* a parallel version of the Karp–Miller–Rosenberg algorithm, computation of
the dictionary of basic factors of $t$; the last symbol of $t$ is #, $|t| = n$, $n-1$ is a power
of two */
$x := t$;
RENUMBER($x$); /* recall that RENUMBER updates $x$ and computes table POS */
begin
  for $i := 1 \,..\, n$ do in parallel {name[$i$, 1] := $x[i]$; pos[1, $i$] := POS[$i$];}
  $k := 1$;
  while $k < n-1$ do
    {for $i := 1 \,..\, n-2k+1$ do in parallel $x[i] := (x[i], x[i+k])$;
     delete the last $2k-1$ entries of $x$;
     RENUMBER($x$);
     for $i := 1 \,..\, n-2k+1$ do in parallel {name[$i$, $2k$] := $x[i]$; pos[$2k$, $i$] := POS[$i$];}
     $k := 2k$;}
end.
```

The correctness of the algorithm follows from fact (∗). The number of iterations is logarithmic and the dominating operation is the procedure RENUMBER. The thesis follows now from the preceding lemma.

Let name[(i, j), p] be the name of a $p*p$ subarray of a given array t with upper-left corner at position (i, j). In the two-dimensional case there is a fact analogous to (∗) (see Fig. 4).

$$\begin{aligned}(**)\qquad & \text{name}[(i,j), 2p] = \text{name}[(k,l), 2p] \\ & \Leftrightarrow (\text{name}[(i,j), p] = \text{name}[(k,l), p] \text{ and} \\ & \quad \text{name}[(i+p, j), p] = \text{name}[(k+p, l), p] \text{ and} \\ & \quad \text{name}[(i+p, j+p), p] = \text{name}[(k+p, l+p), p] \text{ and} \\ & \quad \text{name}[(i, j+p), p] = \text{name}[(k, l+p), p]).\end{aligned}$$

Using fact (∗∗), the algorithm above can be easily modified to compute the dictionary of basic subarrays. Hence, the algorithm for the two-dimensional case is essentially the same as that for the one-dimensional case. This completes the proof. □

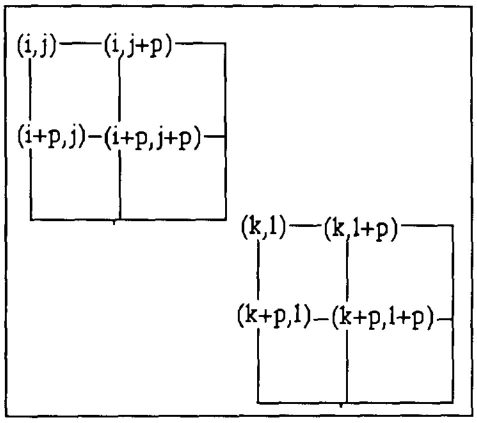

Fig. 4. A subarray of size $2p*2p$ repeats. (The subarrays can overlap.)

3. Almost optimal parallel algorithms for strings.

We start the applications of the previous algorithms with the problem of searching for squares in strings. A square is a nonempty word of the form ww. It is a nontrivial problem to find a square factor within a word in sequential linear time. A simple application of failure functions gives a quadratic sequential algorithm and also a parallel NC-algorithm. To do so, we can compute a failure function P_i for each suffix $x[i..n]$ of the word x. Then there is a square in x, prefix of the suffix $x[i..n]$, iff $P_i[j] \geqslant (j-i+1)/2$ for some $j>i$. Computing a linear number of failure functions leads to a quadratic sequential algorithm and to a parallel NC-algorithm. However, with such an approach, the parallel computation requires a quadratic number of processors. We show how the divide-and-conquer method used in the sequential case saves time and space in parallel computation.

The main part of the known most efficient sequential algorithms for finding a square in a word is an operation called *test* (see [24] or [10]). This operation applies to square-free words u and v and tests whether the word uv contains a square (the square must begin in u and end in v). This operation is a composition of two smaller ones *righttest* and *lefttest*. The first (second) operation tests whether uv contains a square whose centre is in v (u). The operation "test" may be implemented by using two auxiliary tables related to string-matching technique. For a given word $v = v[1]..v[m]$ and position k ($0 \leqslant k \leqslant m$) let PREF[$k$] be, as in Section 2, the length of the longest prefix of v which occurs at k in v (i.e. which is a prefix of $v[k+1]..v[m]$). Let also $\mathrm{SUF}_u[k]$ be the size of longest suffix of $v[1..k]$ which is also a suffix of u. This table SUF_u can be computed in the same way as PREF, e.g. by computing PREF for u^R and v^R.

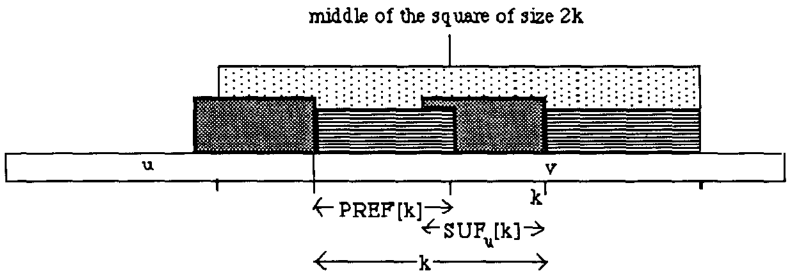

Fig. 5. A square of size $2k$ occurring in uv. The suffix of $v[1..k]$ of length $\mathrm{SUF}_u[k]$ is a suffix of u; the prefix of $v[k+1..m]$ of length $\mathrm{PREF}[k]$ is a prefix of v. $\mathrm{PREF}[k]+\mathrm{SUF}_u[k]\geq k$.

Lemma 3.1. *Tables* PREF *and* SUF_u *being computed, functions righttest*(u, v), *lefttest*(u, v) *and test*(u, v) *can be computed in* $O(\log(n))$ *time with* $n/\log(n)$ *processors (in constant time with n processors in the concurrent-write model).*

Proof. Given tables PREF and SUF_u, the computation of righttest(u, v) reduces to the comparison of k and $\mathrm{PREF}[k]+\mathrm{SUF}_u[k]$. A square of length $2k$ is found iff the latter quantity is greater than or equal to k, as shown in Fig. 5. The evaluation of righttest is done by inspecting in parallel all lengths $k=1, 2, \ldots, m$. Hence, in $O(\log(n))$ time (and constant time with the concurrent-writes model), the time to collect the boolean value, we can compute righttest. By grouping the values of k in intervals of length $\log(n)$ we can even reduce the number of processors to $n/\log(n)$. The same holds for lefttest and, thus, for test. This completes the proof. □

A recursive algorithm for testing occurrences of squares can be easily constructed with the help of the function test.

```
Algorithm SQUARE:
/* checks if word $x=x[1]..x[n]$ contains a square. It is assumed w.l.o.g. that $n$ is
a power of two */

begin
if $n>1$ then
  {for $i\in\{1, n/2+1\}$ do in parallel
    check recursively whether $x[i..i+n/2-1]$ contains a square;
    /* if the algorithm has not already stopped then */
    if (test($x[1..n/2]$, $x[n/2+1..n]$)) then return true;}
return false; /* if the value true has not been already returned */
end.
```

Theorem 3.2. *The algorithm* SQUARE *tests the squarefreeness of a word* $x[1]..x[n]$ *in* $O(\log(n)^2)$ *parallel time using n processors in a model without concurrent writes.*

Proof. The complexity of the algorithm SQUARE essentially comes from the computation of basic factors (Theorem 2.3). Each recursive step during the execution of the algorithm takes $O(\log(n))$ time with n processors as shown by Lemma 3.1 and Theorem 2.1. The number of steps is $\log(n)$. This gives the result. □

Lemma 3.3. *The failure table P can be computed in $O(\log(n)^2)$ time with n processors without using concurrent writes (in $O(\log(n))$ time if the dictionary of basic factors is already computed).*

Proof. We can assume that the table PREF is already computed. Let us consider pairs $(i, i+\mathrm{PREF}[i]-1)$. These pairs correspond to intervals of the form $[i..j]$. The first step is to compute all such intervals which are maximal in the sense of set inclusion order. It can be done with the use of a parallel prefix computation. For each k compute

$$\mathrm{maxval}(k)=\max(\mathrm{PREF}[1], \mathrm{PREF}[2]+1, \mathrm{PREF}[3]+2, \ldots, \mathrm{PREF}[k-1]+k-2).$$

Then we "turn off" all positions k such that $\mathrm{maxval}(k) \geqslant \mathrm{PREF}[k]+k-1$. We are left with maximal subintervals $(i, \mathrm{RIGHT}[i])$. Let us (in one parallel step) mark all right ends of these intervals and compute the table $\mathrm{RIGHT}^{-1}[j]$ for all right ends j of these intervals. For the other values of j, $\mathrm{RIGHT}^{-1}[j]$ is undefined. Again, using a prefix computation for each position k we can compute minright$[k]$ to be the first marked position to the right of k. Then, in one parallel step, we set

$P[k]:=0$ if minright$[k]$ is undefined, and

$P[k]:=\max(0, k\text{-}\mathrm{RIGHT}^{-1}[\mathrm{minright}[k]]+1)$ otherwise (see Fig. 6).

This completes the proof. □

One may observe that if the table PREF is given then even $n/O(\log(n))$ processors are sufficient to compute table P, because our main operations are prefix computations.

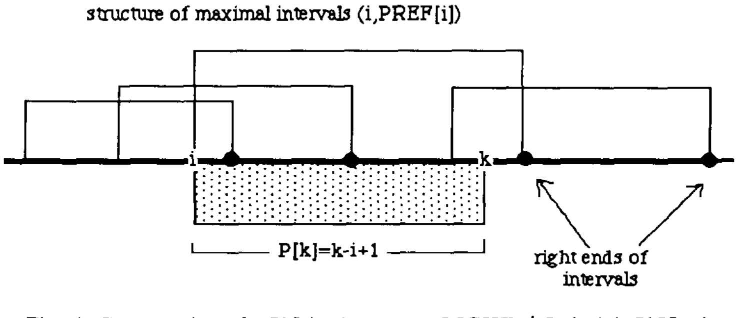

Fig. 6. Computation of $P[k]$ in the case $i=\mathrm{RIGHT}^{-1}$ $[\mathrm{minright}[k]] \leqslant k$.

Corollary 3.4. *The periods of all the prefixes of a word can be computed in* $O(\log(n)^2)$ *time with n processors without using concurrent writes (in* $O(\log(n))$ *time if the dictionary of basic factors is already computed).*

Proof. The period of prefix $x[1\,.\,.\,i]$ is $i-P[i]$. □

We now give another consequence of Lemma 3.3 connected to the previous result. Given a word x, the minimal pattern-matching automaton for x is the minimal deterministic automaton which recognizes the language A^*x. The construction of such automata is a basic technique in string-matching problems. Its sequential construction is straightforward, but it is usually related in the literature to the failure function P (see [2]). We will use this fact to develop a parallel algorithm.

Theorem 3.5. *Assume that the dictionary of basic factors is computed and the alphabet has size* $O(1)$. *Then we can compute the minimal pattern-matching automata for one string, or for a finite set of strings, in* $O(\log(n))$ *time with n processors on an exclusive-write* PRAM.

Proof. We prove only the one-pattern case. The case with many patterns can be done in essentially the same way, although trees are to be used (instead of one-dimensional tables, see [1]). The KMR algorithm works for trees as well.

Let x be a string (pattern) of length n. We can assume that the failure table P for x is already known. We refer the reader to [2] for the sequential construction of pattern-matching automata. Our algorithm is essentially a parallel version of that construction. The minimal pattern-matching automaton for x has $\{0, 1, \ldots, n\}$ as set of states. The initial state is 0 and n is the only accepting state.

Define first the transition function δ for occurrences of symbols of x only. Let $\delta[a, i]=i+1$ for $a=x(i+1)$ and $\delta[a, 0]=0$ for $a\neq x(1)$.

For each symbol a of the alphabet, define a modified failure table as follows:

for $i<n$, $P_a[i]=\text{if } (x[i+1]=a) \text{ then } i \text{ else } P[i]$,
and $P_a[n]=P[n]$.

Let $P_a^*[i]=P_a^k[i]$, where k is such that $P_a^k[i]=P_a^{k+1}[i]$. Tables P_a will be treated here as functions which could be iterated indefinitely. We can easily compute all tables $P_a^*[i]$ in $O(\log(n))$ time with n processors using the doubling technique ($\log(n)$ repetitions of $P_a[i]:=P_a^2[i]$). Then the transition table of the automaton is constructed as follows:

```
for each letter $a$ and position $i$ such that $\delta[a, i]$ is not already defined
do in parallel $\delta[a, i]:=\delta[a, P_a^*[i]]$.
```

This completes the proof. □

The next application of algorithms of Section 2 is to algorithmic questions related to palindromes. We consider in this paper only palindromes of even length. Let PAL denote the set of such nonempty even palindromes (words of the form ww^R).

Rad$[i]$ is the radius of the maximal even palindrome centred at position i;

$$\mathrm{Rad}[i]=\max\{k:p[i-k+1\,..\,i]=(p[i+1\,..\,i+k])^{\mathrm{R}}\},$$

where R is the operation of reversing the text. If k is the maximal integer satisfying $p[i-k+1\,..\,i]=(p[i+1\,..\,i+k])^{\mathrm{R}}$ then we say that i is the centre of the maximal palindrome $p[i-k+1\,..\,i+k]$. In many cases we will identify palindromes with their corresponding subintervals of $[1\,..\,n]$.

Lemma 3.6. *Assume that the dictionaries of basic factors for the word and its reverse are computed. Then the table* Rad *of maximal radii of palindromes can be computed in* $O(\log(n))$ *time with n processors.*

Proof. The proof is essentially the same as computation of table PREF: a variant of parallel binary search is used. □

Denote by Firstcentre$[i]$ (Lastcentre$[i]$) the tables of first (last) centres to the right of i (including i) of palindromes containing position i.

Lemma 3.7. *If the table* Rad *is computed then the table Firstcentre can be computed in* $O(\log(n))$ *time with n processors on the exclusive-write model.*

Proof. We first describe an $O(\log(n)^2)$ time algorithm. It uses the divide-and-conquer approach. A notion of half-palindrome is introduced as shown in Fig. 7. The value Firstcentre$[k]$ is the first (to the right of k including k) right end of a half-palindrome containing k.

The structure of the algorithm is recursive:

> The word is divided into two parts $x1$ and $x2$ of equal sizes. In $x1$ we disregard the half-palindromes with right end in $x2$. Then, we compute in parallel the table Firstcentre for $x1$ and $x2$ independently. The computed table, at this stage, gives correct final value for positions in $x2$. However, the part of the table for $x1$ may be incorrect due to the fact that we disregard for $x1$ half-palindromes ending in $x2$. To cope with this, we apply procedure UPDATE($x1$, $x2$).

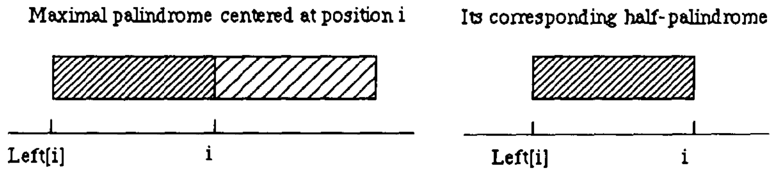
Maximal palindrome centered at position i
Its corresponding half-palindrome
Left[i]
i
Left[i]
i

Fig. 7.

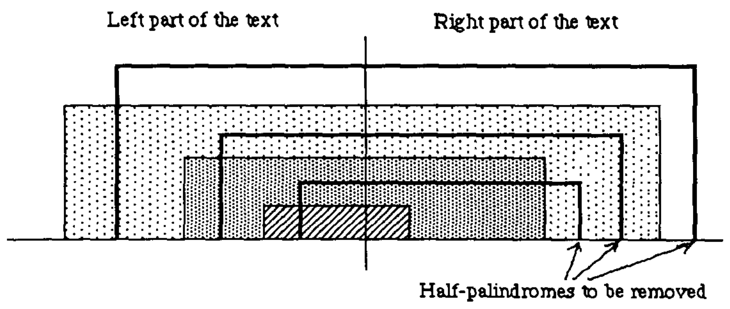

Fig. 8.

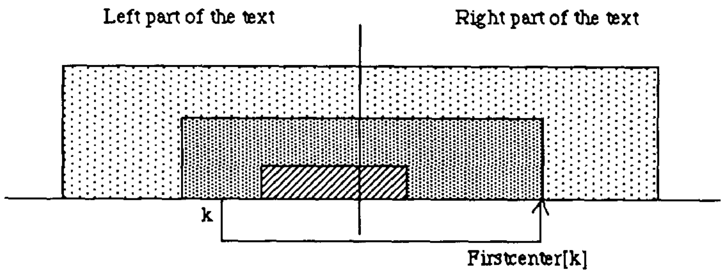

Fig. 9.

The table is updated looking only at previously disregarded half-palindromes. The structure of these half-palindromes is shown in Fig. 8.

The half-palindromes can be treated as subintervals $[i..j]$. We introduce a (partial) ordering on half-palindromes as follows. $[i..j] \leqslant [k..l]$ iff ($i \leqslant k$ and $j \leqslant l$).

```
procedure UPDATE($x1$, $x2$):
/* only half-palindromes starting in $x1$ and ending in $x2$ are considered */
remove all half-palindromes but minimal ones;
/* it can be easily done in log($n$) time by parallel prefix computation */
/* we are left with the situation presented in Fig. 9 */
we compute for each position $k$ the first to the left starting position $s[k]$ of a half-
palindrome;
/* see Lemma 1.2 */
begin
for each position $k$ in $x1$ do in parallel /* see Fig. 9 */
    Firstcentre[$k$]:= min(Firstcentre[$k$], right end of half-palindrome starting at
        $s[k]$);
end procedure.
```

At the beginning of the whole algorithm Firstcentre$[k]$ is set to $+\infty$. One can see that procedure UPDATE takes $O(\log(n))$ time using $|x1|+|x2|$ processors. The depth of the recursion is logarithmic; thus, the whole algorithm takes $\log(n)^2$ time.

We now briefly describe how to obtain $O(\log(n))$ time with the same number of processors. We look at the global recursive structure of the previous algorithm. The initial level of the recursion is the zero level, the last one has depth $\log(n)-1$. The crucial observation is that we can simultaneously perform procedures UPDATE at all levels of the recursion. However, at the qth level of the recursion, instead of minimizing the value of Firstcentre[k], we store the values computed at this level in a newly introduced table Firstcentre$q$: for each position k considered at this level the procedure sets Firstcentre$q$[k] to the right end of half-palindrome starting at $s[k]$ (see the text of UPDATE). Hence, procedure UPDATE is slightly modified; at level q it computes the table Firstcentre$q$. Assume that all entries of all tables are initially set to infinity.

After computing all tables Firstcenter$q$ we assign a processor to each position k which computes sequentially (for its position) the minimum of Firstcentre$q$[k] over recursion levels $k=0, 1, \ldots, \log(n)-1$. This, in total, takes $O(\log(n))$ time with n processors. However, we compute $O(\log(n))$ tables Firstcentre$q$, each of size n, so we have to be sure that n processors suffice to compute all these data in $O(\log(n))$ time. The computation of Firstcentre$q$, at a given level of recursion can be done by simultaneous applications of parallel prefix computations (to compute the first marked element to the left of each position k) for each pair $(x1, x2)$ of factors (at the qth level we have 2^q such pairs). The prefix computations at a given level take together $O(n)$ elementary sequential operations. Hence, the total number of operations for all levels is $n\ O(\log(n))$. It is easy to compute Firstcentre$q$ for a fixed q using n processors. This requires $O(n\log(n))$ processors for all levels q; however, we have only n processors.

At this moment we have an algorithm working in $O(\log(n))$ time whose total number of elementary operations is $O(n\log(n))$. We can apply Brent's lemma: if $M(n)/P(n)=T(n)$ then the algorithm performing the total number of $M(n)$ operations and working in parallel time $T(n)$ can be implemented to work in $O(T(n))$ time with $P(n)$ processors (see e.g. [19]).

In our case $M(n)=O(n\log(n))$ and $T(n)=O(\log(n))$. Brent's lemma is a general principle; its application depends on the detailed structure of the algorithm as to whether it is possible to redistribute processors efficiently. However, in our case we deal with computations of very simple structure: multiple applications of parallel prefix computations. Hence, n processors suffice to make all computations in $O(\log(n))$ time. This completes the proof. $\square$

Theorem 3.8. *Even palstars can be tested in* $O(\log(n))$ *time with n processors on the exclusive-write model if the dictionary of basic factors is already computed.*

Proof. Let "first" be a table whose ith value is the first position j in t such that $t[i\,..\,j]$ is an even nonempty palindrome; it is zero if there is no such prefix even palindrome. Define first[n]$=n$. This table can be easily computed using the values of Firstcentre[i]. Then the sequential algorithm tests even palstars in the following natural way: it finds the first prefix even palindrome and cuts it; then such a process is repeated as long as possible; if we are done with an empty string then we return "true": the initial word is an even palstar. The correctness of the algorithm is proved in [21]. We make a parallel version of the algorithm.

Compute table $\text{first}^*[i]=\text{first}^k[i]$, where $\text{first}^k[i]=\text{first}^{k+1}[i]$, using a doubling technique. Now the text is an even palstar iff $\text{first}^*[1]=n$. This can be tested in one step. This completes the proof. □

Unfortunately the natural sequential algorithm described above for even palstars does not work for arbitrary palstars. However, in this more general case a similar table first can be defined and computed in essentially the same way as for even palindromes. Then the palstar recognition can be easily reduced to the following reachability problem: is there a path from position 1 to n. The positions are nodes of a directed graph whose maximal outdegree is three (see [17]). It is easy to solve such a reachability problem in linear sequential time; however, we do not know how to solve it by an almost optimal parallel algorithm.

In fact, the case of even palstars could also be viewed as a reachability problem: its simplicity is related to the fact that in this case the outdegree is one.

Lemma 3.9. *The table Lastcentre can be computed in* $O(\log(n))$ *time with n processors on the exclusive-write model.*

Proof. We compute the maximal (with respect to inclusion) half-palindromes. We mark their leftmost positions and then the table Lastcentre is computed according to Fig. 10. The position k has to find only the first (to the left, including k) marked position. This completes the proof. □

Theorem 3.10. *Compositions of k palindromes* for $k=2$, 3, or 4 *can be tested in* $O(\log(n))$ *time with* $n/\log(n)$ *processors on exclusive-write model if the dictionary of basic factors is computed.*

Proof. The parallel algorithm is an easy parallel version of the sequential algorithm in [17], which applies to $k=2, 3, 4$. The key point is that if a text t is a composition uv of even palindromes then there is a composition $u'v'$ such that u' is a maximal prefix palindrome or v' is a maximal suffix palindrome of t. The maximal prefix (suffix) palindrome for each position of the text can be computed efficiently using the table

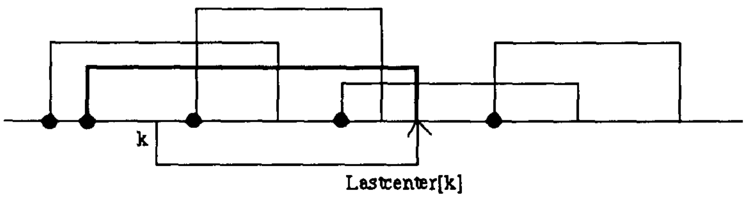

Fig. 10.

Lastcentre (in the case of suffix palindromes the table is computed for the reverse of the string).

Given the tables Rad and Lastcentre (also for the reversed string) the question "is the text $t[1\,..\,i]$ a composition of two palindromes", can be answered in constant time using one processor. The computation of logical "or" of questions of this type can be done by a parallel prefix computation. The time is $O(\log(n))$, and $n/O(\log(n))$ processors suffice (see Lemma 1.1). This completes the proof. □

Several combinatorial algorithms on graphs and strings consider strings on an ordered alphabet. A basic algorithm often used in this context is the computation of maximal suffixes, or of the minimal nonempty suffixes, according to the alphabetic ordering. These algorithms are strongly related to the computation of the Lyndon factorization of a word, that is recalled now. A Lyndon word is a nonempty word which is minimal among its nonempty suffixes. Chen, Fox and Lyndon (see [23]) proved that any word x can be uniquely factorized as $l_1 l_2\,..\,l_h$ such that $h \geqslant 0$, the l_i's are Lyndon words and $l_1 \geqslant l_2 \geqslant \cdots \geqslant l_h$. It is known that l_h is the minimal nonempty suffix of x. In the next parallel algorithm, we will use the following characterization of the Lyndon word factorization of x.

Lemma 3.11. *The sequence of nonempty words $(l_1, l_2, \ldots, l_h)$ is the Lyndon factorization of the word x iff $x = l_1 l_2\,..\,l_h$ and the sequence $(l_1 l_2\,..\,l_h, l_2 l_3\,..\,l_h, \ldots, l_h)$ is the longest sequence of suffixes of x in decreasing alphabetical order.*

Proof. We only prove the "if" part and leave the "only if" part to the reader.

Let $s_1 = l_1 l_2\,..\,l_h$, $s_2 = l_2 l_3\,..\,l_h, \ldots, s_h = l_h$, $s_{h+1} = \varepsilon$. As a consequence of the maximality of the sequence, one may note that, for each $i = 1, \ldots, h$, any suffix w longer than s_{i+1} satisfies $w > s_i$. To prove that $(l_1, l_2, \ldots, l_h)$ is the Lyndon factorization of x, we have only to show that each element of the sequence is a Lyndon word. It is the case for l_h since, again by the maximality condition, l_h is the minimal nonempty suffix of x and, thus, is less than any of its nonempty suffixes. Assume, *ab absurdo*, that v is both a nonempty proper prefix and suffix of l_i, and let w be such that $vw = s_i$. Since vs_{i+1} is a suffix of x longer than s_{i+1}, we have $vs_{i+1} > s_i$. The latter expression implies that $s_i > w$, which is a contradiction. This proves that no nonempty proper suffix v of l_i is a prefix of l_i. But since, again $vs_{i+1} > s_i$, we must have $v > s_i$ and also $v > l_i$. So, l_i is a Lyndon word. This completes the proof. □

Theorem 3.12. *The maximal suffix of the text and the Lyndon factorization can be computed in* $O(\log(n)^2)$ *time with n processors on the exclusive-write* PRAM.

Proof. We first apply the KMR algorithm of Section 3 using the exclusive-write model to the word $x\#\,..\,\#$. Assume (contrary to the previous sections) that the special character $\#$ is the minimal symbol. Essentially, it makes no difference to algorithm KMR. We then get the ordered sequence of basic factors of x given by tables pos. After padding enough dummy symbols to the right, each suffix of x becomes also a basic factor. Hence, for each position i, we can easily get $\mathrm{Rank}[i]$: the rank of the ith suffix $x[i]\,..\,x[n]$ in the sequence of all suffixes. One may note that the padded character # has no effect on the alphabetical ordering since # is less than any other character.

Let us define $L[i]$ to be that position j such that $j \leqslant i$ and $\mathrm{Rank}[j] = \min\{\mathrm{Rank}[j']: 1 \leqslant j' \leqslant i\}$. By the above characterization of the Lyndon word factorization of x, $L[i]$ gives the starting position of the Lyndon factor containing i. $L[i]$ can be computed by prefix computation. Define an auxiliary table $G[i] = L[i]-1$. Now it is enough to compute the sequence $G[n]$, $G^2[n]$, $G^3[n]$, ..., $G^h[n]=0$. In fact, such sequence gives the required list of positions, starting at position n. It is easy to mark all positions of $[0\,..\,n]$ contained in this list in $\log(n)^2$ parallel time (see Section 1). These positions decompose the word into its Lyndon factorization. This completes the proof. □

4. Almost optimal parallel algorithms for strings and arrays

We consider three problems whose algorithmic solution for one-dimensional and two-dimensional cases are essentially the same. These problems consist in finding a longest factor (largest subarray) which

(1) repeats (occcurs at least twice),
(2) is a common factor (subarray) of the given two texts (arrays), or
(3) is symmetric.

Denote by $\mathrm{Cand1}(s)$, $\mathrm{Cand2}(s)$ and $\mathrm{Cand3}(s)$ a function whose value is any factor (subarray) of size s which satisfies, respectively, condition (1), (2) and (3). If there is no such factor (subarray) then the value of the function is *nil*. The values of the function are candidates of size s: we want to find a non-nil candidate with maximum size s.

Generally, assume that we have a function $\mathrm{Cand}(s)$ satisfying the following monotonicity property:

$$\mathrm{Cand}(s+1) \neq \mathrm{nil} \;\Rightarrow\; \mathrm{Cand}(s) \neq \mathrm{nil}.$$

Denote

$$\mathrm{Maxcand} = \{\mathrm{Cand}(s) : s \text{ is maximum such that } \mathrm{Cand}(s) \neq \mathrm{nil}\}.$$

Assume also that $\mathrm{Cand}(0)$ is some special value.

Lemma 4.1. *Assume that the function* $\mathrm{Cand}(s)$ *can be computed with* $P(n)$ *processors in* $T(n)$ *parallel time. Then the value of* Maxcand *can be computed with the same number of processors in* $\mathrm{O}(\log(n)T(n))$ *time.*

Proof. We can assume w.l.o.g. that n is a power of two and $\mathrm{Cand}(n) = \mathrm{nil}$. A variant of binary search can be applied. We look at $\mathrm{Cand}(n/2)$, if it is nil then we try $\mathrm{Cand}(n/4)$;

otherwise, we look at Cand($n/2+n/4$), and so on. In this way, after a logarithmic number of operations we compute Maxcand. This completes the proof. □

Assume that the dictionary of basic factors is computed. We can easily test equality of two factors whose size is a power of two; however, what about factors whose sizes are not powers of two. It so happens that we can test equality of such factors also in constant time. We define identifiers for factors (subarrays) whose size s is not a power of two. In the case of strings let the identifier of the factor of length s starting at i be Ident(i, s) = (name(i, k), name($i+s-k$, k)), where k is the highest power of 2 less than s. In the case of arrays the identifier of the $s*s$ subarray with left upper corner positioned at $v1$ is Ident($v1$, s) = (name($v1$, k), name($v2$, k), name($v3$, k), name($v4$, k)), as can be seen in Fig. 11.

For each i and s we can compute Ident(i, s) in constant time, if the dictionary of basic factors is computed. The equality of two factors (subarrays) can be checked in constant time. The key point is that identifiers are of constant size. The identifiers can be used to look for factors (subarrays) whose sizes are not powers of two. A typical example is the pattern-matching problem.

Theorem 4.2. *The pattern-matching problem for strings* (*arrays*) *can be solved with n processors in* $O(\log(n)^2)$ *time on the exclusive-write model and in* $O(\log(n))$ *time on the concurrent-write model.*

Proof. If p is a pattern and t is a text then we create the dictionary of basic factors common to p and t. The identifier ID of the pattern is computed. Then we look, in parallel, for a factor p' of t (of size $|p|$) whose identifier equals ID. This completes the proof. □

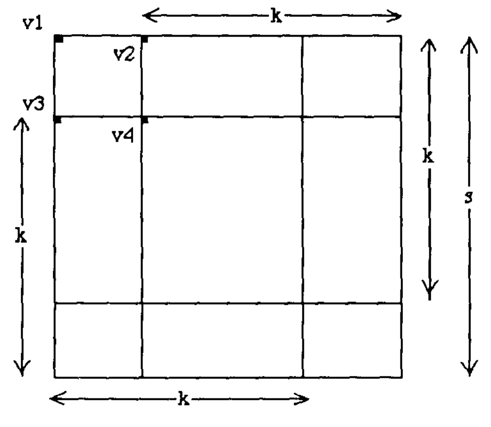

Fig. 11. Ident($v1$, s) = (name($v1$, k), name($v2$, k), name($v3$, k), name($v4$, k)), where k is the highest power of 2 less than s.

Observe that none of the linear-time sequential algorithm for two-dimensional pattern-matching is well parallelizable (see [5, 6]). Next we apply Lemma 4.1 to Cand1, Cand2 and Cand3.

Theorem 4.3. *The longest repeated factor (largest repeated subarray) can be computed with n processors in* $O(\log(n)^2)$ *time on the exclusive-write model and in* $O(\log(n))$ *time on the concurrent-write model.*

Proof. It is enough to compute Cand1(s) with n processors in $O(\log(n))$ time on the exclusive-write model and in $O(1)$ time on the concurrent-write model. Given identifiers of all factors (subarrays) for each position of the text (array), it is an easy matter to compute in $O(\log(n))$ time two positions with the same identifier. We can sort pairs (ident(i, s), i) lexicographically. Any two such consecutive pairs with the same identifier part in the sorted sequence will give the required positions. The model is the exclusive-write PRAM. With concurrent writes we can use an auxiliary table (bulletin board) and proceed similarly as in the proof of Lemma 2.2. No initialization of the bulletin board is required. This completes the proof. □

Theorem 4.4. *The longest common factor of two strings (largest common subarray of two arrays) can be computed with n processors in* $O(\log(n)^2)$ *time on the exclusive-write model and in* $O(\log(n))$ *time on the concurrent-write model.*

Proof. In this case we compute, at the beginning, the common dictionary for both texts (arrays) X, Y. The further proof is essentially the same as the proof of Theorem 4.3. There are small technical differences. We sort pairs (ident(i, s), i) lexicographically. Now i's are positions in X and Y. We can partition the sorted sequence into segments consisting of pairs having the same first component (identifier). In these segments it is now easy to look for two positions i, j such that i is a position in X and j is a position in Y. This completes the proof. □

Theorem 4.5. *The longest symmetric factor (largest symmetric subarray) can be computed with n processors in* $O(\log(n)^2)$ *time on the exclusive-write model and in* $O(\log(n))$ *time on the concurrent-write model.*

Proof. In the case of strings the algorithm can be designed using the table Rad of radii of maximal palindromes. Hence, we have to prove only the two-dimensional case. Let X be an $n*n$ array of symbols or elements of any linearly ordered set (with constant time comparison of elements). It is enough to compute Cand3(s) with n processors in $O(\log(n))$ time on the exclusive-write model and in $O(1)$ time on the concurrent-write model.

Let us compute the array Y which results by reflecting each entry of X with respect to the centre. Denote by reflect(i, j, s) the left upper corner in Y of $s*s$ subarray B which results from the subarray A by reflection with respect to the centre (see

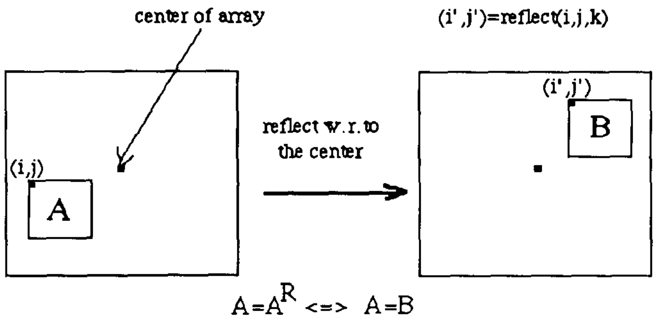

Fig. 12. The $k*k$ subarray A is symmetric iff ident$((i,j),k)$=ident$((i',j'),k)$.

Fig. 12). Now we can compute the common dictionary of basic subarrays of arrays X and Y. The subarray A is symmetric iff $A = B$, which can be checked in constant time using s-identifiers at position (i,j) in X and position reflect(i,j,k) in Y. The function reflect is easily computable in constant time. Once we know which $s*s$ subarrays A are symmetric, we can easily choose one of them (if there is any) as a value of Cand3(s). Hence, Cand(s) can be computed within the required bounds of the complexity. This completes the proof. □

5. Concluding remarks

If the size of the alphabet is fixed then all the presented algorithms which work in O(log(n)) time with n processors on a concurrent-write PRAM can be transformed to a strictly optimal algorithm (O(log(n)) time, n/log(n) processors) by using the "four Russians" method (see [2]). The following theorem could be proved.

Theorem 5.1. *There are optimal parallel algorithms (on a concurrent-write* PRAM*) for two-dimensional string matching, computing maximal symmetric subarrays, testing even palstars and constructing minimal pattern-matching automata in the case of fixed-size alphabets.*

The algorithms use auxiliary tables with $n^{1+\varepsilon}$ entries: the tables do not have to be initialized.

Idea of the proof. We are compressing substrings (subarrays) of size O(log(n)) and replacing them by their names. Essentially the same methods as in [16, 25] can be used. □

Remark. In the exclusive-write model we number subwords according to their lexicographic ordering. In the model with concurrent writes (if we want to reduce the time by a logarithmic factor) the numbering of words does not necessarily need to reflect their lexicographic order. Essentially, the lexicographic order is not necessary for most of the applications presented in this paper: like repeated factors and longest common factor. However, the lexicographic numbering problem is interesting in its own right; it helps also in computing minimal (maximal) suffixes and Lyndon factorization.

In most applications the table pos (part of the dictionary) is not needed. However, it helps in finding quickly an object with a given name. Below we show a more essential application of this table. Assume that we have computed the dictionary of basic factors for the text t of size n and that we receive the text p of size m which is a power of two, where m is much smaller than n. Then we want to check whether p is factor of t and (if p is a factor of t) to compute the dictionary of basic factors for p. However, we require that the dictionaries for t and p are consistent; i.e. the same words must have the same names in both of them.

Theorem 5.2. *Assume that we are given a text p whose size m is a power of two and the dictionary for t for a text t of length n with names of subwords corresponding to the lexicographic ordering. Then we can check whether p is a factor of t and (if yes) compute dictionary for p consistent with dictionary for t in* $O(\log(n)\log(m))$ *time with only m processors of an exclusive-write* PRAM.

Proof. We perform essentially the same algorithm as in the proof of Theorem 2.3. However, the procedure RENUMBER is now different. We have a composite pair (x, y) of names, where x, y are names of k-factors. This pair corresponds to a factor of size $2k$ starting at position i and we want to give a name (a number) to it. Now observe that the table $\text{pos}(*, 2k)$ gives a lexicographic ordering of all such pairs (x, y).

The pair (x', y') corresponding to $j = \text{pos}(r, 2k)$ is $(\text{name}(j, k), \text{name}(j+k, k))$. Hence, for a fixed pair (x, y) (identifying factor at position i) one processor can compute in $O(\log(n))$ time, by a binary search in a linearly ordered set, the integer r such that (x, y) corresponds to $\text{pos}(r, 2k)$. Then $\text{name}(i, 2k)$ is set to r. If there is no such integer then we report that p is not a factor of t. Hence, the procedure RENUMBER can be computed in $O(\log(n))$ time with only m processors and the names are consistent with dictionary for t. We have $\log(m)$ iterations. Hence, the total time is $O(\log(n)\log(m))$. This completes the proof. □

Theorem 5.3. *Assume that we are given a text p whose size m is a power of two and the text t of length n. Also assume that there are given the dictionary for t together with all bulletin boards computed in the computation of this dictionary on a concurrent-write PRAM. Then we can check whether p is a factor of t and (if yes) compute the dictionary for p consistent with the one for t in* $O(\log(m))$ *time with only m processors of an exclusive-write* PRAM.

Proof. The procedure RENUMBER at a given stage of the computation of the dictionary for p is now simplified. To compute the name of (p, q) we have only to look at the entry (p, q) of the bulletin board corresponding to this stage of the computation of the dictionary for p. There are no write conflicts now; we are only reading from the (ready) bulletin boards. This completes the proof. □

The theorem can be strengthened to cover the general case (when m is not a power of two), however, a more complicated algorithm using suffix tress should be applied [4]. We state the following problems as open problems:

(1) To design an almost optimal parallel algorithm to test general palstars – compositions of even and odd nontrivial (at least two-letter) palindromes. In the sequential case there is also a big difference in the difficulty of computing even palstars and general palstars in linear time; see [17];

(2) To find an optimal parallel algorithm to test squarefreeness (free alphabets of arbitrary size);

(3) To design an almost optimal parallel algorithm which computes the maximal repeated factor (subarray) and a square factor in case of alphabets with "don't care" symbol. The "don't care" symbol is a special universal symbol which matches any other symbol (including itself). For two strings x, y we write $x \approx y$ iff x and y coincide except (may be) at positions containing the "don't care" symbol. The factor x repeats iff there is a factor y occurring at a different position such that $x \approx y$. The "square" can now be redefined as factor of the form xy with $x \approx y$. The "philosophy" of the KMR algorithm does not apply to this case: the relation $\approx$ is not an equivalence relation.

References

[1] A. Aho and M. Corasick, Efficient string-matching: an aid to bibliographic search, *Comm. ACM* **18** (1975) 333–340.

[2] A. Aho, J. Hopcroft and J. Ullman, *The Design and Analysis of Computer Algorithms* (Addison-Wesley, 1974).

[3] A. Apostolico, On context-constrained squares and repetitions in a string, *RAIRO Inform. Theor. Appl.* **18** (1984) 147–159.

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

[5] T. Baker, A technique for extending rapid exact string matching to arrays of more than one dimension, *SIAM J. Comput.* **7** (1978) 533–541.

[6] R.S. Bird, Two-dimensional pattern-matching, *Inform. Process. Lett.* **6** (1977) 168–170.

[7] R. Boyer and J. Moore, A fast string searching algorithm, *Comm. ACM* **20** (1977).

[8] R. Cole, Parallel merge sort, *Found. Comput. Sci.* (1987).

[9] M. Crochemore, An optimal algorithm for computing the repetitions in a word, *Inform. Process. Lett.* **12** (1981).

[10] M. Crochemore, Transducers and repetitions, *Theoret. Comput. Sci.* **45** (1986) 63–86.

[11] M. Crochemore, Longest common factor of two words, in: *CAAP'87*, 23–36.

[12] M. Crochemore, String matching and periods, *EATCS Bulletin* **39** (1989) 149–153.

[13] M. Crochemore and D. Perrin, Two-way pattern matching, *J. ACM*, to appear.

[14] J. Duval, Factorizing words over an ordered alphabet, *J. Algorithms* **4** (1983) 363–381.

[15] Z. Galil, Open problems in stringology, in: A. Apostolico and Z. Galil, eds., *Combinatorial algorithms on words* (Springer, Berlin, 1985) 1–12.

[16] Z. Galil, Optimal parallel algorithm for string matching, *Inform. and Control* **67** (1985) 144–157.

[17] Z. Galil and J. Seiferas, A linear time on-line recognition algorithm for palstars, *J. ACM* **25** (1978) 102–11.

[18] Z. Galil and J. Seiferas, Time space optimal string matching, *J. Comput. System Sci.* **26** (1983) 280–294.
[19] A. Gibbons and W. Rytter, *Efficient Parallel Algorithms* (Cambridge University Press, 1988).
[20] R. Karp, R. Miller and A. Rosenberg, Rapid identification of repeated patterns in strings, arrays and trees, in: *Proc. of ACM Symp. on Theory Comput.* **4** (1972) 125–136.
[21] D. Knuth, J. Morris and V. Pratt, Fast pattern matching in strings, *SIAM J. Comput.* **6** (1977) 322–350.
[22] G. Landau, B. Schieber and U. Vishkin, Parallel construction of a suffix tree, in: *ICALP'87*, 314–325.
[23] M. Lothaire, *Combinatorics on Words* (Addison-Wesley, 1983).
[24] M. Main and R. Lorentz, An $O(n \log n)$ algorithm for finding all repetitions in a string *J. Algorithms* (1984) 422–432.
[25] M. Main and R. Lorentz, Linear time recognition of square-free strings, in: A. Apostolico and Z. Galil, eds., *Combinatorial algorithms on words* (Springer, Berlin 1985) 271–278.
[26] G. Manacher, A new linear time on-line algorithm for finding the smallest initial palindrome of the string, *J. ACM* **22** (1975) 345–351.
[27] W. Rytter, On the parallel transformations of regular expressions to nondeterministic finite automata, *Inform. Process. Lett.* **31** (1989) 103–109.
[28] U. Vishkin, Optimal parallel pattern matching in strings, *Inform. and Control* **67** (1985) 91–113.