CUCS-74-83

OPTIMAL PARALLEL ALGORITHMS FOR STRING MATCHING

Zvi Galil*

Tel-Aviv University

Columbia University

*Research supported by National Science Foundation Grant MCS-8303139.

OPTIMAL PARALLEL ALGORITHMS FOR STRING MATCHING

Zvi Galil*
Tel-Aviv University
Columbia University

Abstract: Let WRAM [PRAM] be a parallel computer with p processors (RAM's) which share a common memory and are allowed simultaneous reads and writes [only simultaneous reads]. The only type of simultaneous writes allowed is a simultaneous AND: several processors may write 0 simultaneously into the same memory cell. Let t be the time bound of the computer. We design below families of parallel algorithms that solve the string matching problem with inputs of size n (n is the sum of lengths of the pattern and the text) and have the following performance in terms of p, t and n:

1. For WRAM: $pt = O(n)$ for for $p \leq n/\log n$.
2. For PRAM: $pt = O(n)$ for $p \leq n/\log^2 n$.
3. For WRAM: t = constant for $p = n^{1+\epsilon}$ and any $\epsilon > 0$.
4. For WRAM: $t = O(\log n/\log \log n)$ for $p = n$.

Similar families are also obtained for the problem of finding all initial palindromes of a given string.

1. Introduction.

We design parallel algorithms in the following model: p sychronized processors (RAM's) share a common memory. Any subset of the processors can simultaneously read from the same memory location. We sometimes allow simultaneous writing in the weakest sense: any subset of processors can write the value 0 into the same memory location (i.e., turn off a switch). We denote by WRAM [PRAM] the model that allows [does not allow] simultaneous writing. We also consider (but only briefly) other models of parallel computation. We actually design a family of algorithms because we have a parameter p. The performance of the family is measured in terms of three parameters: p--the number of processors, t--the time, and n--the size of the problem instance.

It is well known that every parallel algorithm with p processors and time t can be easily converted to a sequential algorithm of time pt. Hence the analog of linear-time algorithm in sequential computation is a family of parallel algorithms with $pt = O(n)$. We therefore call such algorithms optimal. Surprisingly, while there are many problems for which linear-time algorithms are known, there are very few problems for which optimal parallel algorithms are known for a wide range of p. So few, that we list them here.

Every associative function of n variables can be computed by a PRAM in $pt = O(n)$ for $p \leq n/\log n$. (Use a binary tree, each leaf "treats" n/p inputs.) For a certain subset of these functions including the n variable OR (AND), $\Omega(\log n)$ time is needed on the PRAM [CD], so $pt = O(n)$ is unattainable for $p >> n/\log n$. Consequently, the only question left is with how few processors can we compute these functions in constant time on a WRAM. The answer depends on the specific function. The n variable OR (or AND) function can be computed by WRAM in $pt = n$ for $p \leq n$ (i.e., in time = 1 with n processors). The n variable MAXIMUM function can be computed in $pt = O(n)$ for $p \leq n/\log \log n$ and in constant time with $n^{1+\epsilon}$ processors

*Research supported by National Science Foundation Grant MCS-8303139.

(for every $\epsilon > 0$) [V], [SV].

Optimal parallel algorithms are known for merging two sorted arrays (for $p \leq n/\log n$ on a PRAM); merging can be done in constant time even by a PRAM with $n^{1+\epsilon}$ processors [SV] and in log log n with n processors [V], [BM]. Recently, optimal parallel algorithms were designed for the problem of converting an expression to its parse tree [BV] and for Selection [Vi].

What is common to all these problems except Selection is that for each one of them there is a trivial (sequential) linear-time algorithm. In this paper we design optimal parallel algorithms for string matching. The linear-time algorithm for string matching is by now very well understood, but at one time, it was quite a major discovery. Unlike the case of computing n variable functions (where it is trivial) and merging (where it is quite simple) designing optimal parallel algorithms for string matching was not immediate.

As for the problems mentioned above, we designed other parallel algorithms that perform string matching on WRAM in <u>constant</u> time with only $n^{1+\epsilon}$ processors. As in the cases above the time is proportional to $1/\epsilon$. If only n processors are available the time needed is $O(\log n/\log\log n)$.

The families of algorithms we design have several appealing features:

1. They are <u>not</u> derived from any of the variants of the linear-time sequential algorithms ([KMP], [BM]). The latter do not seem to be parallelizable, because they construct sequentially tables which are used sequentially. So, even giving the tables for free does not seem to help much. Two known algorithms are parallelizable but do not yield optimal parallel algorithms: the $O(n \log n)$ algorithm in [KMR] yields $tp = O(n \log^2 n)$ and the probabilistic linear-time algorithm in [KR] yields a <u>probabilistic</u> family with $tp = O(n \log n)$.

2. The algorithms we design are all derived from <u>one</u> algorithm: it is an algorithm for WRAM with $p = n$ and $t = \log n$ for the case that the text is twice longer than the pattern.

3. The algorithms make use of properties of periodicities in strings derived from the Periodicity Lemma which states that two different periodicities cannot coexist long enough (if they do, then there is a common refinement). Similar properties were used in a different way to design a linear-time algorithm for string matching which uses only constant (five) registers [GS]. Therefore, we have here an example for a relationship between sequential space and parallel time in the lowest level.

4. As in the algorithm in [GS], it is possible to write a very short program (for each processor), but a longer explanation is needed mainly because the algorithm uses implicitly properties of periodicities several times.

5. The algorithms use what seems to be a novel method of communication among the various processors, as will be indicated below.

<u>String matching</u> is the following problem. The input consists of two strings, x (the pattern) and y (the text), over a given alphabet of a <u>fixed</u> size. The output is a Boolean array indicating all the occurrences of x in y.

In Section 2 we prove several simple facts on periodicities of strings used by the algorithm. In Section 3 we sketch the main algorithm which is non optimal ($p = 3n$ $t = \log n$) and only deals with a special case ($|y| = 2|x| = 2n$). In Section 4 we complete the details of the algorithm. In Section 5 we show how the four families of parallel algorithms mentioned above are derived from the main algorithm. In Section 6 we briefly discuss other models of parallel computation and the problem of finding all initial palindromes of a given string.

2. <u>Periodicity in Strings</u>.

A string u is a <u>period</u> of a string w if w is a prefix of u^k for some k or equivalently if w is a prefix of uw. We call the shortest period of a string w <u>the period</u> of w. Thus a is the period of aaaaaaa while aa,aaa, etc are also periods of w. We say that w has <u>period size</u> P if the length of the period of w is P. If w is at least twice longer than its period we say that w is <u>periodic</u>.

We will consider prefixes of the pattern x of increasing length. Assume we consider a prefix u and then a prefix v. In the case that u is periodic we will say that the periodicity <u>continues</u> in v if the period of v is the same as the period of u (e.g. u = abcabcab, v = abcabcabcabcabca) and that the periodicity <u>terminates</u> otherwise (e.g. the same u, v = abcabcabcd...).

We will need some simple facts about periodicities.

<u>Fact 1</u> (The Periodicity Lemma)[LS]: If w has two periods of size P and Q and $|w| \geq P + Q$, then w has a period of size $\gcd(P,Q)$.

For a one line proof see [GS].

In the rest of this section an occurrence at j will mean an occurrence at position j in a given fixed string z.

Fact 2: If v occurs at j and $j + \hat{P}$, $\hat{P} \leq |v|/2$, then (1) v is periodic with a period of length $\hat{P}$, and (2) v occurs at $j + P$, where P is the period size of v. The first half of Fact 2 follows from the alternative definition of period. The second half of Fact 2 holds since by Fact 1 P must divide $\hat{P}$.

In the rest of this section we consider a periodic string $v = u^k u'$, $k > 1$, u the period of v, u' a proper prefix of u, and $|u| = P$. Let $L = \ell P$, $\ell = \lceil |v|/P \rceil$. The next two facts follow from a simple counting of periods.

Fact 3: If v occurs at j and $j + mP$, $m \leq k$, then $u^{k+m}u'$ occurs at j.

Fact 4: v occurs at j, $j + P$ and $j + L$ iff $u^{k+\ell}u'$ occurs at j.

Fact 5: If v occurs at j and $j + \Delta$, $\Delta \leq |v| - P$, then Δ is a multiple of P.

Proof: Otherwise $\Delta = mP + r$, $0 < r < P$, and $m < k$. Let $w = u^{(k-m)}u'$. w is a suffix of v, so it occurs at $j + mP$. It is also a prefix of v, so it occurs at $j + \Delta = j + mP + r$. By Fact 2, w has a period of size r in addition to a period of size P. By Fact 1, it has a period of size $\gcd(P,r) < P$ which divides P. Hence P cannot be the period size of v. □

We call an occurrence of v at j important if v does not occur at $j + P$.

Fact 6: If there are two important occurrences of v at r and s, $r > s$, then $r - s > |v| - P$.

Proof: Assume $r - s \leq |v| - P$. By Fact 5, $r - s = mP$. By Fact 4, $u^{k+m}u'$ occurs at r, and hence v occurs at $r + P$, and the occurrence at r cannot be important. □

3. A sketch of the main algorithm.

The input is a string $z = x\,\$\,y$ of length $3n + 1$. x is the pattern, $|x| = n$, and y is the text, $|y| = 2n$. Both are over a given alphabet of fixed size which does not contain \$. The output is a Boolean array of length $3n + 1$ called SWITCH. The final value of SWITCH[i] is 1 iff an occurrence of x starts with z_i. The WRAM has $3n + 1$ processors. Processor i is responsible for z_i and SWITCH[i].

Given a string u of length ℓ we say that we test for u at (position) i (of z) if we execute AND($u_1 = z_i, \ldots, u_\ell = z_{i+\ell-1}$). Such a test finds if u occurs at i and takes one unit of time on the WRAM. The straight forward algorithm that tests for x at all i's needs n^2 processors.

Let $x^{(i)}$ be the prefix of x of size 2^i, and let $x^{(i+1)} = x^{(i)}y^{(i)}$. The algorithm consists of log n stages. After stage i SWITCH [j] = 1 if and only if $x^{(i)}$ occurs at j.

We now describe stage $i + 1$, which takes a constant (at most six) steps. The task of the stage is to test whether each occurrence of $x^{(i)}$ is followed by an occurrence of $y^{(i)}$. In case the answer is negative the corresponding 1 in SWITCH is turned off.

We divide the array SWITCH into blocks of size 2^{i-1}. We say that property i holds if each block has at most one 1. We distinguish between two cases: the regular case, and the periodic case.

The regular case is the one in which the first block of SWITCH has only one 1 (at position 1). By induction, the other blocks may have at most two 1's. In a block with two 1's, the 1 at the smaller position is turned off. (This occurrence of $x^{(i)}$ is not a beginning of an occurrence of $x^{(i+1)}$.) As a result, property i holds. There are 2^{i-1} processors responsible for the block. Hence, in two steps they can test for $y^{(i)}$ at the appropriate position if they knew which comparisons they ought to perform. We will explain below how this is done. We call it a regular step.

In the periodic case that follows a regular case the first block has two 1's; the second of which at position $P + 1$. It follows from Fact 2 that $x^{(i)}$ is periodic with period size P. In the periodic case we test whether the periodicity of $x^{(i)}$ continues in $x^{(i+1)}$. We do it in two steps using $x^{(i)}$ as a yardstick. If $x^{(i+1)}$ has the same period we similarly find all its occurrences. Then we start stage $i + 2$ in the periodic case. If $x^{(i+1)}$ does not have the same period we turn off (justifiably) many 1's in SWITCH. As a result, property i holds and we complete the stage with a regular step. Each part in the discussion above makes some use of properties of periodicities.

During the algorithm the processors need to communicate. For global communication we have a bulletin board, BB, where some announcements are posted; e.g. if the case is periodic and the size of the period. Also, the processors responsible for a block need to communicate in order to find which comparisons they ought to make in a regular step. For this purpose we have local bulletin boards, lbb's. We can use an additional array to store the lbb's. Alternatively, each lbb can be stored at the last element of its block. At the end of each stage one of every two consecutive lbb's dies and may transfer some information to the surviving one before it passes away. (See Figure 1.)

4. The Details.

The flow chart of the algorithm is given in Figure 2. In this section we give the details of each one of the seven boxes in the flow chart. The first and last stage are slightly different and are discussed at the end of the section.

We enter box 1 after a regular step in stage i. Consider blocks numbers $2j-1$ and $2j$ at the end of stage i. They contain at most one 1. The lbb of the first block dies at the end of the stage. The processor responsible for the second lbb (number $2j.2^{i-2}$) looks at the dying lbb and if it is not empty, it tries to transfer its contents to its lbb. Two 1's per block are discovered when its lbb is already nonempty.

Box 1 deals with the case $j = 1$. If two 1's are discovered in the (new) first block we are in the periodic case, which is explained below. Boxes 2, 3 deal with the case $j > 1$. If two 1's are discovered the first is turned off by the processor responsible for the surviving lbb. (It is the processor that discovers the two 1's.)

To understand box 4, the regular step, consider Figure 1. If the occurrence starts at $z_{\Delta+1}$, then the lbb contains Δ. Processor j in the group that corresponds to the block makes two comparisons: $x_k = z_{k+\Delta}$? for $k \in \{j+2^i, j+2^i+2^{i-1}\}$. If one of the answers is negative it turns off the 1 at SWITCH $(\Delta+1)$. This is the only place where the concurrent write is used.

The test in box 1 is actually handled differently. Since SWITCH(1) = 1, the processor responsible for the second lbb of stage i (the first of stage $i+1$) looks at its lbb. If it is nonempty it contains P; i.e., $x^{(i)}$ is periodical with period size P. The processor posts P on BB.

During the periodic loop (boxes 5,6) the lbb's are not updated and are not used. BB will contain P and $L \equiv \ell P$, where $\ell = \lceil 2^i/P \rceil$. When we enter box 5 from box 1 $\ell \in \{2,3\}$ $(2^{i-2} < P + 1 \leq 2^{i-1})$. Updating L in the loop is easy: $L \leftarrow$ if $2L - P > 2^{i+1}$ then $2L - P$ else $2L$.

Let $\hat{x}^{(i+1)}$ be the prefix of x of size $2^i + L$ $(|x^{(i+1)}| = 2^{i+1} \leq |\hat{x}^{(i+1)}| < 2^{i+1} + P)$. In box 5 we test whether the periodicity continues in $\hat{x}^{(i+1)}$ by using $x^{(i)}$ as a yardstick. (Fact 4 $v = x^{(i)}$, $j = 1$, $\hat{x}^{(i+1)} = u^{k+\ell}u'$); the first processor tests whether SWITCH(P+1) = 1 and SWITCH(L+1) = 1. Recall that P and L are posted. The first test is redundant when we come from box 1. Similarly, in Box 6, we find the occurences of $\hat{x}^{(i+1)}$ as follows (Fact 4 $v = x^{(i)}, \hat{x}^{(i+1)} = u^{k+\ell}u'$): processor p_j that sees 1 at SWITCH(j) checks whether SWITCH(P+j) = 1 and SWITCH(L+j) = 1. If one of the tests fails p_j turns off the 1.

Recall that an occurrence of $v = x^{(i)}$ at j is called important if $x^{(i)}$ does not occur at $j+P$. Since SWITCH(1) = 1 and one of SWITCH(P+1), SWITCH(L+1) is zero, at least one of the occurrences of $x^{(i)}$ at positions $j \leq L + 1 - P$ is important. By Facts 5,6, either the occurrence at 1 is important or there is exactly one important occurrence at some j $1 \leq j \leq L+1-P$.

When we test if the periodicity continues, first, p_1 checks SWITCH(P+1). If it is zero, then the occurrence at 1 is important and p_1 posts 0 on BB. Otherwise it tests SWITCH(P+L). If it is 1, the periodicity continues. Otherwise, each processor p_j tests (using SWITCH) whether there is an important occurrence at j. The unique p_j that succeeds posts $j-1$ on BB.

Next, each processor p_r with SWITCH(r) = 1 uses SWITCH and the posted value of $j-1$ to check whether there is an important occurrence of $x^{(i)}$ at $r+j-1$. If there is no such an occurrence it turns off the 1 at SWITCH(r). This is justified because $\hat{x}^{(i+1)}$ cannot occur at r, since in $\hat{x}^{(i+1)}$ there is an important occurrence of $x^{(i)}$ at j. At this point property i holds by Fact 6.

Before executing the regular step (box 4) the lbb's are restored. Each processor p_r with SWITCH(r) = 1 writes r-1 in its lbb. By Claim 2, no conflict occurs. To be able to do it, each processor knows in each stage where is its lbb. This information can be easily precomputed or updated dynamically.

The first stage is very simple. Processor p_j tests whether $z_j z_{j+1} = x_1 x_2$. If the test succeeds, p_j turns on SWITCH(j) and makes the j-th lbb for the second stage point to the 1. Recall that the size of the blocks in the second stage is 1.

We now discuss the changes needed for the last stage, but first we need to elaborate on the other stages. Consider stage i+1, and an occurrence of $x^{(i)}$ at $j \leq n$. Assume $j+2^{i+1} > n+1$, so $x^{(i+1)}$ cannot occur at j simply because it is too long, and the \$ does not match any symbol of x. In case the first mismatch from the left is the \$ the algorithm will not turn off the 1 at SWITCH (j). (It is as though the \$ and the following symbols always match the symbols compared to them. As a result, a 1 in SWITCH may stand for an overhanging occurrence.

In the last stage, if property i holds, or if the periodicity terminates (and as a result of including overhanging occurrences it means that it terminates before the \$) we execute a regular step without any change. The only change is in the case that the periodicity continues. While in the other stages it means that the periodicity continues to $\hat{x}^{(i+1)}$, in the last stage it continues only to the \$. We find ourself in this case when $L + 2^i \geq n$ ($|\hat{x}^{(i+1)}| \geq |x|$). We call an occurrence of $x^{(i)}$ at j <u>special</u> if $j + 2^i \leq n$ and $j + P + 2^i > n + 1$ (if the next occurrence of $x^{(i)}$, at j+P is the first overhanging occurrence). As with important occurrences the unique p_j that finds a special occurrence at j posts j-1 on BB. (Note that j = mP + 1 for some m, $x = u^m u^k u'u''$, $u'u''$ a prefix of u^2.) Then each p_r that sees a 1 at SWITCH(r) checks whether SWITCH(r+j-1) = 1 and if not it turns off the 1. If the test succeeds it checks whether SWITCH(r+j-1+P) = 1. If the test succeeds we know that x occurs at r (since the tests imply that $u^{m+k+1}u'$ occurs at j). If the test fails we still do not know the answer. Note that in this case the occurrence at $x^{(i)}$ at r+j-1 is important and by Fact 6 if we restrict attention to occurrences at r's such that the occurrence at r+j-1 is important, then property i holds. So we activate the lbb's and use a regular step to test whether such occurrences of $x^{(i)}$ extend to occurrences of x.

5. <u>The Four Families</u>.

<u>5.1 Using only n/log n processors</u>.

The main algorithm can be implemented with only n/log n processors using the four Russians trick [AHU] to pack log n symbols into one number.

Each processor is responsible for s consecutive symbols in z and in SWITCH, where s = c log n and c depends on the alphabet size: processor p_r will be responsible for z_j, SWITCH(j) $j \in A_r$ $\equiv \{(r-1)s+1,\ldots,rs\}$. First, each p_r packs each substring of z of length s that starts with z_j, $j \in A_r$, into a new symbol $\hat{z}_j$. Then it compares each $\hat{z}_j$, $j \in A_r$, with $\hat{z}_1$ and if they are equal it sets SWITCH(j) = 1. This has the effect of the first t = log s stages and takes O(s) = O(log n) time.

Assume the next ((t+1)-st) stage is in the regular case. The other stages are as in the main algorithm. The only difference is that in each regular step the packed symbols $\hat{z}_i$ are used.

If the (t+1)-st stage is periodical, then the period size P < s/2, and we need also to pack the bits in SWITCH. Each p_r packs the s consecutive segments of SWITCH starting with each SWITCH(j) $j \in A_r$. When the periodicity continues and we test for occurrences of $\hat{x}^{(i+1)}$ we can handle all the 1's in a packed symbol of SWITCH simultaneously using some simple bit vector operations on the packed symbols. Even if we disallow bit vector operations, the n/log n processors can prepare (in time O(log n)) a table to implement these operations.

5.2 The general case.

We now have an algorithm with $tp_0 = O(n)$ for $p_0 = n/\log n$. This immediately yields a family with $tp = O(n)$ for $p \leq n/\log n$ because of the well known downward translation. In general, if $tp_0 = f(n)$, then we have a family with $tp = f(n)$ for $p \leq p_0$, because having only p processors, each one will simulate p_0/p processors and the time will be slowed down by a factor of p_0/p.

We still have to deal with the case in which $|x|$ and $|y|$ are unrelated. Let $n = |x|+|y|$ (the length of the input) and $m = |x|$. If $p \leq 2n/m$ we divide y into $p/2$ equal parts. Let the i-th piece be the concatenation of the i-th and (i+1)st parts. There are p pieces and we assign one processor per piece. The size of a piece $S = 2|y|/(p/2)$ satisfies $4n/p \geq S \geq 2n/p \geq m$. Each processor looks for all occurrences of x in its piece in time $O(S) = O(n/p)$. Hence in this case, when we have a small number of processors, we have an optimal algorithm simply because we still solve the problem sequentially.

If $p > 2n/m$ ($p \leq n/\log m$) we break y into overlapping pieces of size $2m$. The number s of such pieces satisfies $n/m \leq s \leq 2n/m < p$. We assign p/s ($\leq m/\log m$) processors per piece. By the first paragraph above, all the occurrences in a piece can be found in time t such that $t \cdot p/s = O(m)$, or $tp = O(ms) = O(n)$.

5.3 On the PRAM.

Consider the main algorithm. The only case of concurrent write is the regular step: the 2^{i-1} processors of a block compute an AND. If we do not allow concurrent write, we can no longer execute one stage in constant time. The algorithm on the PRAM takes time $O(\log^2 n)$, because each stage takes $O(\log n)$ time.

Fortunately, we can implement this algorithm with only $n/\log^2 n$ processors. Each processor is responsible for $\log^2 n$ symbols or for $\log n$ packed symbols. In a regular step, the processors in a block make $\log n$ comparisons of packed symbols (in time $\log n$). They record only whether all the comparisons succeed. Then using the implicit tree structure, they 'and' their results in time $O(\log n)$.

The discussion above yields an algorithm on a PRAM with $p = n/\log^2 n$ and $t = O(\log^2 n)$. The rest is as in subsection 5.2. The algorithm can be implemented without simultaneous reads.

5.4 Having many processors.

Assume $|y| = 2|x| = 2n$. As was noted above, with n^2 processors we can solve string matching in constant ($t = 2$) time on the WRAM. We show below that if $p = n^{1+1/k}$ we can solve string matching in time $O(k)$. This immediately gives the third and fourth families: for the third, take $\varepsilon = 1/k$ and the constant is k. For the fourth, take $k = \log n/\log\log n$. In this case $p = n \log n$, but by packing symbols we reduce p to n.

In this subsection we use a stronger version of WRAM. In case of a write conflict the processor with the minimum number is the one that writes. At the moment, if it is not known whether such a WRAM can be simulated by our weaker type without time loss. However, in our case, such simulation is possible.

Assume one subset of p processors tries to write simultaneously into a register and the processor with the minimal number succeeds. It was observed in [FRW] that our weaker model of WRAM can do the same in four steps: the processors are partitioned into $\sqrt{p}$ groups of size $\sqrt{p}$. In the first step each group computes whether one of its members wants to write. The result is a Boolean array of size $\sqrt{p}$. In the second step the 1's in that array that are not first are turned off. This is possible because there are $\sqrt{p}$ processors for each 1. Now, the processors in the corresponding group find in a similar way the minimal in the group. Such a simulation will easily be extended to our case.

When we have n or more processors we can use them to have $x^{(i+1)}$ more than twice larger than $x^{(i)}$ and as a result, to have less than $\log n$ stages. Specifically, let $p = 3n^{1+1/k}$. The processors are divided into $3n$ groups of $n^{1/k}$ processors. Each group contains one principal processor, and is responsible for one symbol of z and SWITCH. The length of $x^{(i)}$ is $n^{i/k}$. In the first stage (finding all occurrences of $x^{(1)}$) the i-th group looks for an occurrence at i.
The size of the blocks for stage $i + 1$ is $|x^{(i)}|/2 = n^{i/k}/2$. A regular step is simple, since we have enough processors: the number of processors in the groups

corresponding to a block is $n^{(i+1)k}/2 = |x^{(i+1)}|/2$.

The parts concerning periodicity are slightly different, because the size of blocks much more than doubles from one stage to the next. To test for periodicity, each principal processor in the first block that sees 1 writes its group number minus 1 on the same place of BB. The one with the minimal group number succeeds, and posts the period size P.

Let $L_i = \lfloor x^{(i)}/P \rfloor P$. L_i can be easily maintained and is available in stage i + 1. Note that $L_{i+1} \leq 2n^{1/k} L_i$. To test if the periodicity continues, the first group checks whether $SWITCH(1+jL_i)=1$ for $j=1,\ldots,2n^{1/k}$. (In this case $\hat{x}^{(i+1)} = 2n^{1/k}L_i+x^{(i)}$, so $x^{(i+1)} < |\hat{x}^{(i+1)}| < 3x^{(i+1)}$.)

If the test succeeds, a similar test is used to test which occurrence of $x^{(i)}$ is extended to an occurrence of $\hat{x}^{(i+1)}$. If the test fails, using the stronger form of concurrent writing the first group finds the first j with $SWITCH(1+jL_i) = 0$. The value of j is posted on BB, and next $SWITCH(r) = 1$ is not turned off only if the r-th group finds that $SWITCH(r+jL_i) = 0$, and for all $k < j$ $SWITCH(r+kL_i) = 1$.

The stronger type of concurrent write is used only within groups, and the memory locations are different for different groups. The simulation mentioned above (for one group) can be obviously extended to our case. We left out the details of allocating of processors. For fixed k this task is immediate because we can assume that $n = 2^{kr}$ for some r. In the general case ($|x|$ and $|y|$ unrelated) the number of processors needed is only $nm^{1/k}$ and with $p = n$ the time bound is $O(\log m/\log\log m)$.

6. Conclusion

We can implement the main algorithm in other models for parallel computation:

1. Boolean circuits of size $O(n \log^2 n)$ and depth $O(\log^2 n)$.
2. Fixed connection networks (the k-dimensional cube) and even networks with fixed degree (CCC's [PV]) in $pt = O(n \log n)$.

The details of these implementation are straightforward. Both use shifting networks as building blocks.

There are some questions unresolved:

1. Can we solve string matching on WRAM with n processors in constant ($O(\log\log n)$) time?
2. Can we solve string matching deterministically on PRAM with $n/\log n$ (or even n) processors in $O(\log n)$ time? (The parallel version of [KR] has $p = n$, $t = O(\log n)$ but is probabilistic.)
3. Can we find optimal parallel algorithms for string matching on fixed connection networks?

Finally, families of parallel algorithms corresponding to all the families mentioned above can be derived for finding all initial palindromes of a given string w. The reduction of the latter problem to string matching [FP] does not help, because it makes use of the table of the KMP algorithm. It is not clear how to compute efficiently this table in parallel. Instead we look for w in w^{rev}, recording in SWITCH also overhanging occurrences. The main algorithm discovers the initial palindromes of length ℓ, $2^{i-1} < \ell \leq 2^i$, in stage i.

Acknowledgement: I am indebted to Uzi Vishkin for suggestions that have led to several improvements.

References:

[AHU] A.V. Aho, J.E. Hopcroft and J.D. Ullman, The design and analysis of computer algorithms, Addison Wesley, Reading MA, 1974.

[BH] A. Borodin and J.E. Hopcroft, Routing, merging and sorting on parallel models of computation, Proc. 14th ACM STOC (1982), pp. 338-344.

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

[BV] I. Bar-On and U. Vishkin, Optimal parallel generation of a computation tree form, Manuscript, Department of Computer Science, Courant Institute, October 1983.

[CD] S.A. Cook and C. Dwork, Bounds on the time for parallel RAM's to compute simple functions, Proc. 14th ACM STOC (1982), pp. 231-233.

[FP] M.J. Fischer and M.S. Paterson, String-matching and other products, in Complexity and Computation, SIAM-AMS Proceedings 7 (R.M. Karp, Ed.), pp. 113-125, American Mathematical Society, Providence, R.I., 1974.

[FRW] F.E. Fich, R.L. Ragde and A. Wigderson, Relations between concurrent-write models of parallel computation, Manuscript, November 1983.

[GS] Z. Galil and J.I. Seiferas, Time-space-optimal string matching, JCSS 26 (1983), pp. 280-294.

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

[KMR] R.M. Karp, R.E. Miller, and A.L. Rosenberg, Rapid identification of repeated patterns in strings, trees and arrays, Proc. 4th ACM STOC (1972), pp. 125-136.

[KR] R.M. Karp and M.O. Rabin, Efficient randomized pattern-matching algorithms, a manuscript.

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

[PV] F.P. Preparata and J. Vuillemin, The cube-connected-cycles: a versatile network for parallel computation, Proc. 20th IEEE FOCS (1979), pp. 140-147.

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

[V] L.G. Valiat, Parallelism in comparison problems, SIAM J. on Computing 4 (1975), pp. 348-355.

[Vi] U. Vishkin, An optimal parallel algorithm for selection, Manuscript, Department of Computer Science, Courant Institute, December 1983.

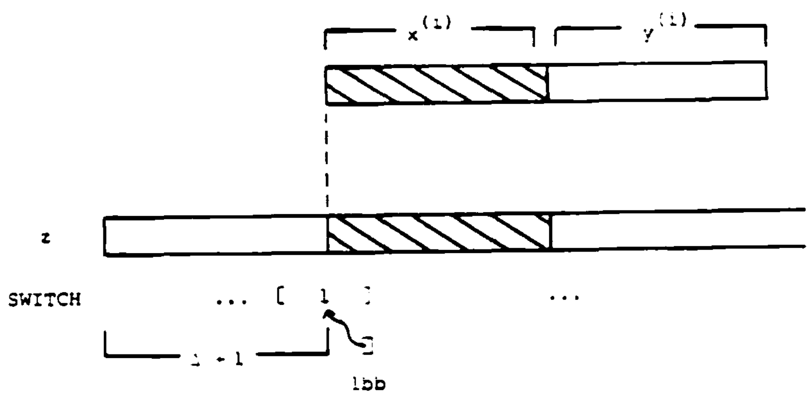

Figure 1. An occurrence of $x^{(i)}$ in z followed by a potential occurrence of $y^{(i)}$; a block in SWITCH and its lbb.

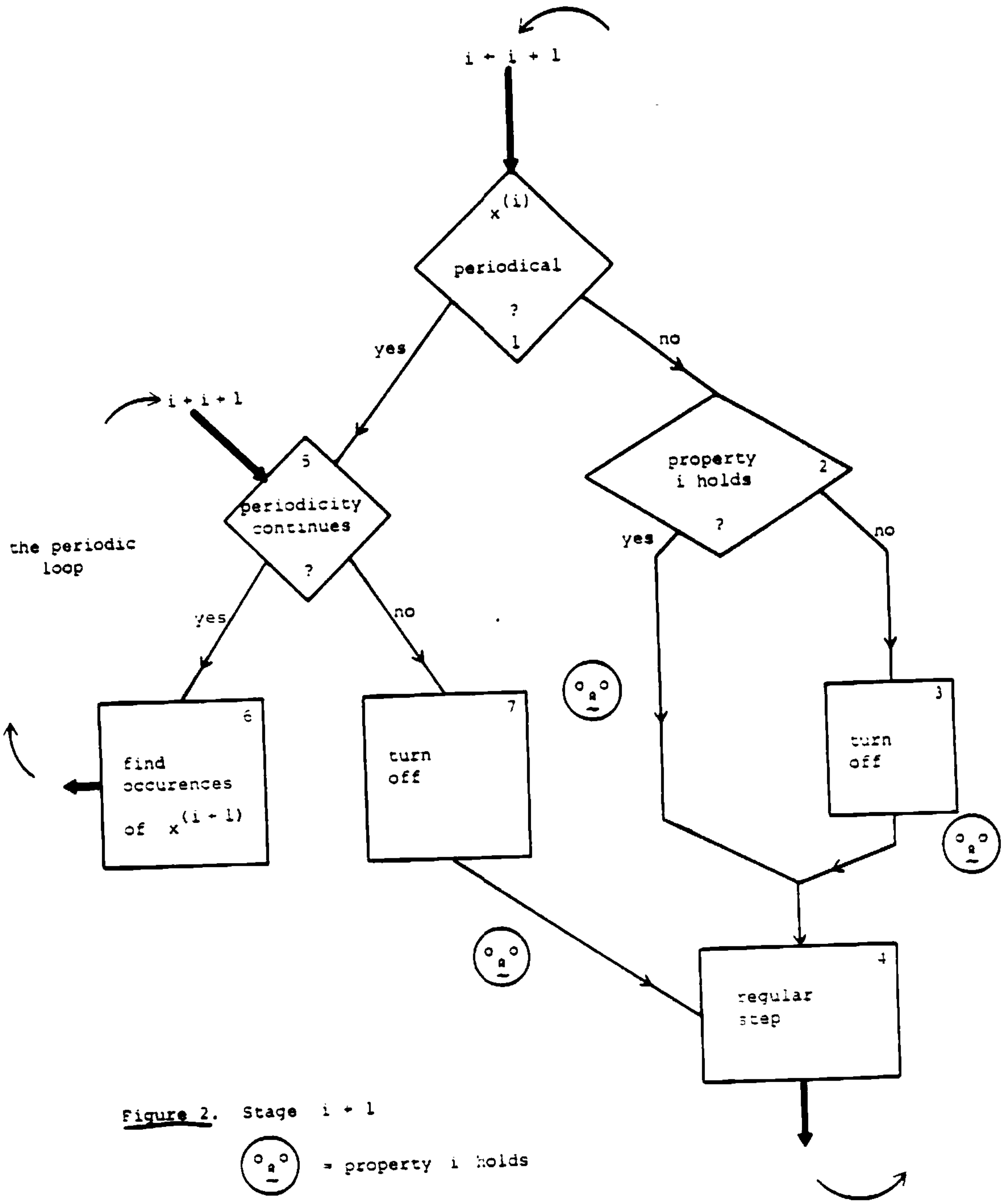

Figure 2. Stage $i + 1$

= property i holds