Optimal Parallel Two Dimensional Text Searching on a CREW PRAM

Amihood Amir
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

Partially supported by NSF grants IRI-90-13055 and CCR-92-23699

Gary Benson
Department of Mathematics
University of Southern California
DRB 155
1024 W. 36th Pl.
Los Angeles, CA 90089-1113

Partially supported by NSF grant IRI-90-13055

Martin Farach
DIMACS
Box 1179
Rutgers University
Piscataway, NJ 08855

Supported by DIMACS under NSF contract STC-88-09648

Proposed running head: Parallel Two Dimensional Text Searching

Address correspondence to:

Gary Benson
Department of Mathematics
University of Southern California
DRB 155
1024 W. 36th Pl.
Los Angeles, CA 90089-1113
(213) 740-2404
gbenson@hto-03.usc.edu

Abstract

We present a parallel algorithm for two dimensional text searching over a general alphabet. This algorithm is optimal in two ways. First, the total number of operations on the text is linear. Second, the algorithm takes time $O(\log m)$ on a CREW PRAM (where m is the length of the longest dimension of the pattern), thus matching the lower bound for string matching on a PRAM without concurrent writes. On a CRCW, the algorithm runs in time $O(\log\log m)$.

Key words: Analysis of algorithms, multidimensional matching, period, parallel algorithms, string

1 Introduction

In this paper, we present the first efficient parallel algorithm for two dimensional pattern matching over a general alphabet. Our text processing phase, which has been traditionally emphasized in pattern matching algorithms (see e.g. [Vis91, Gal92]) is optimal in that its work is linear and its running time is optimal, i.e. it matches the lower bound for CREW PRAMs. Furthermore, it makes no assumptions about the character alphabet. On a CRCW PRAM, our algorithm runs optimally in $O(\log\log m)$ time. The design of such an algorithm was posed as an open problem by Vishkin [Vis85] as early as 1985.

The techniques used in this algorithm are those of two dimensional periodicity. But, we have added important new knowledge and insight to the two dimensional periodicity theory, namely the dense periodic phenomenon. Dense periodicity turns out to have broad uses. In addition to the parallel text scanning presented here, it has been used to optimally solve another open problem, that of two dimensional compressed matching [ABF94].

Note that the pattern preprocessing we use is not work optimal. Since the appearance of this algorithm [ABF93], another algorithm has been published by Cole et. al [CCG+93] that optimally preprocesses the pattern. In addition, the Cole et. al. algorithm, does $O(1)$ text searching on a CRCW PRAM. That algorithm differs from ours in that it uses properties of one-dimensional periodicity.

Before describing our algorithm we present some background on sequential and parallel string matching.

The classical *string matching problem* has as its input a *text* string T of length n and a *pattern* string P of length m. The elements in the text and pattern are taken from an alphabet set Σ. The output is all text locations i where there is a character-by-character match with the pattern, i.e. $T[i+j-1] = P[j],\ j = 1, ..., m$.

String matching is one of the most widely studied problems in computer science and has many linear time solutions, starting with Knuth, Morris and Pratt [KMP77] and Boyer and Moore [BM77]. Parallel algorithms for string matching have also been extensively studied. After a long series of papers by Vishkin, Galil and others, Galil gave a constant time, optimal string matching algorithm for the CRCW PRAM [Gal92]. Breslauer and Galil [BG90] gave an $\Omega(\log\log m)$ time lower bound for CRCW PRAM string matching which Galil avoided in his $O(1)$ algorithm by ignoring preprocessing time. An $\Omega(\log m)$ time lower bound on the CREW PRAM follows from the lower bound for computing the `OR` of m bits [CDR86].

In recent years there has been growing interest in multidimensional pattern matching, largely motivated by problems in low-level image processing [RK82]. Various algorithms exist for the *exact two dimensional matching* problem which is defined similarly to the string matching problem but the text and pattern are rectangular matrices rather than strings.

Baker [Bak78] and, independently, Bird [Bir77] used the Aho and Corasick [AC75] dictionary matching algorithm to obtain a $O(n^2 \log|\Sigma|)$ algorithm for the exact two dimensional matching problem. In [ABF92a], we showed a two dimensional matching algorithm which was linear in the text size. In [GP92] and, using different methods, in [ABF92b] the preprocessing for our algorithm was also improved to linear, thus matching the bounds for one dimensional string matching. Both of these algorithms are alphabet independent, that is, the times are linear even with an unbounded alphabet. The first work optimal parallel algorithm for two dimensional matching was given by Kedem, Landau and Palem [KLP89]. Their algorithm runs in time $O(\log m)$ on a CRCW PRAM. However, this algorithm assumes a fixed alphabet and requires space quadratic in the input size.

Our contribution is to give a work optimal algorithm that runs in $O(\log m)$ time on a CREW PRAM which works for general alphabets using linear space. The CRCW version of this algorithm runs optimally in $O(\log \log m)$ time. The pattern preprocessing runs in $O(\log m)$ time with $O(m^2)$ CRCW processors, or in $O(\log^2 m)$ time with $O(m^2/\log m)$ CREW processors.

Our approach has some similarities to the fast parallel string matching algorithms. Most of those algorithms use some notion of a *witness*, an idea which was introduced in [Vis85]. We also make use of witnesses, though in a different way. The one dimensional string algorithms rely on the fact that periodic strings can be broken down into aperiodic substrings. The algorithm can then proceed by finding occurrences of the smaller aperiodic string. We take a different approach. Amir and Benson showed in [AB92] that periodicity in two dimensions has a much richer structure than does periodicity in strings. In particular, there are four classes of periodicity in rectangular arrays. Here, we achieve our efficient algorithm by dividing the four classes into two groups and showing how each group is handled.

The paper is constructed as follows. In section 2, we give a brief overview of two dimensional periodicity and define dense lattice periodic arrays. In section 3, we formally describe the problem and give an outline of our algorithm. In section 4, we describe the use of witness tables. In section 5, we describe the first phase of the text processing in which we reduce the number of potential locations in which the pattern may occur. Finally, in section 6 we describe how to verify which of the surviving candidates represent actual occurrences of the pattern in the text.

2 Preliminaries

Central to our method are ideas about two-dimensional periodicity developed in [AB92]. There, periodicity in two-dimensional arrays was defined. It was shown that there are four classes of periodicity for rectangular arrays. For the present work, we do not need four classes. Rather we separate arrays into two classes, those that are *dense lattice periodic* and those that are not. We call patterns in the first class *dense* and those in the second class *sparse*.

Below, we give a formal definition of dense lattice periodic arrays, but first, we introduce some terminology from [AB92]. An array or pattern is divided into four *quadrants* (figure 1). Any location in a quadrant where a second copy of the pattern could originate is termed a *source*. In quadrant I (upper left) for example, the origin, $P[0,0]$, is a source. Each source $P[r_1, c_1]$ in quadrant I, defines a periodic vector $v_1 = r_1\mathbf{y} + c_1\mathbf{x}$ where $\mathbf{y}$ is the vector in the direction of increasing row index and $\mathbf{x}$ is the vector in the direction of increasing column index. We denote the vector by its

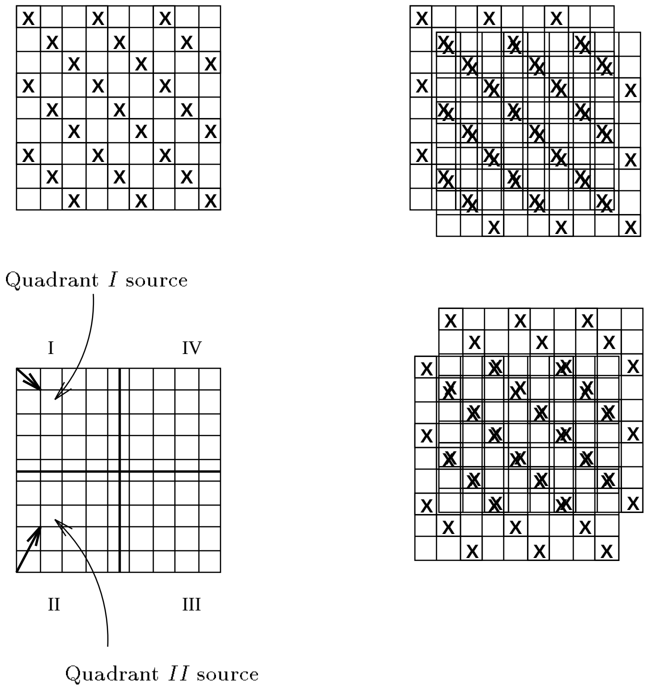


Figure 1:
Each overlap without mismatch defines a *source* and a *periodic vector*.

coefficients, thus $\mathbf{v}_1 = (r_1, c_1)$. The smallest vector (in a lexicographic ordering) is the *basis vector* for quadrant I. A basis vector $\mathbf{v}_2 = (r_2, c_2)$ for quadrant II (lower left) is similarly defined.

Definition: A pattern is *dense lattice periodic* if the coefficients of its basis vectors meet the following two restrictions (figure 2):

1. All of $|r_1|, |r_2|, |c_1|, |c_2| < \lceil \frac{m}{2} \rceil$.
2. Either $|r_1| + |r_2| < \lceil \frac{m}{2} \rceil$ or $|c_1| + |c_2| < \lceil \frac{m}{2} \rceil$.

3 Problem Description

The *Exact Two-dimensional Matching Problem* is defined as follows:

Input: $n \times n$ square text matrix $T[1 \ldots n, 1 \ldots n]$ and $m \times m$ square pattern matrix $P[0 \ldots m-1, 0 \ldots m-1]$.

Output: All locations, $[i, j]$, in T where P occurs, *i.e.*, $T[i+r, j+c] = P[r, c]$, $r, c = 0, ..., m-1$.

For simplicity, we have defined the problem in terms of square arrays, but *our algorithm works for any rectangular array.* We present below an overview of our algorithm. In the discussion that follows, the term *candidate* refers to a possible copy of the pattern in the text. The term *candidate source* or merely *source* refers to the *origin* $P[0,0]$ of a candidate. Two candidates are *compatible* if they can overlap without any mismatches. A candidate is *consistent with the text* if each element of the text in the candidate matches the corresponding pattern element.

Algorithm Overview:

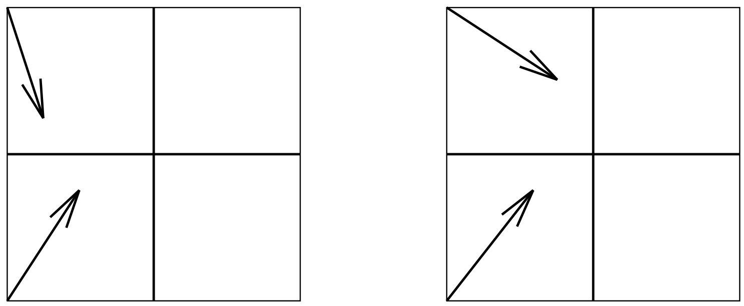
As in many pattern matching algorithms our algorithm consists of a *pattern preprocessing* part and a *text scanning* part.

Pattern Preprocessing: The pattern is analyzed as described in [AB92]. From this analysis we determine whether the pattern is dense or sparse and we obtain a witness table (described below).

Text Processing: Performed in two phases:

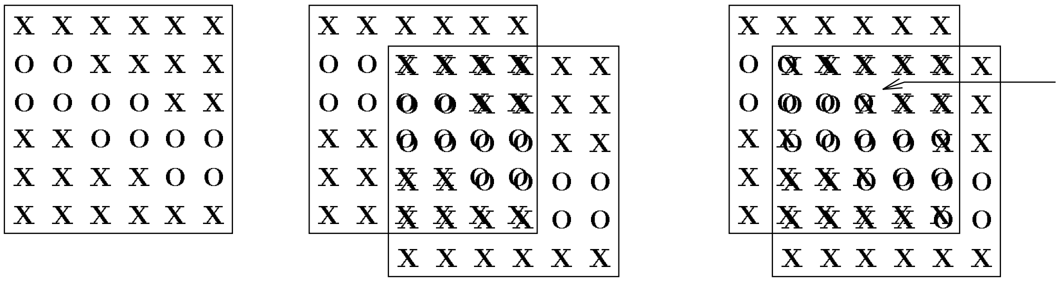
1) *Block Compatibility:* The text is partitioned into disjoint blocks of size $\frac{m}{2} \times \frac{m}{2}$. Within any block, only compatible candidate sources remain. This phase differs in its details depending upon whether the pattern is dense or sparse.

2) *Candidate Verification:* We select a constant number of pattern elements with which to compare each text element. Each text element is then compared with its assigned pattern elements and we propagate the results of the tests to the candidate sources, thereby verifying which of the candidates are actual occurrences of the pattern.



a **b**

Figure 2:
a) Basis vectors for a dense lattice periodic pattern. Here $c_1 + c_2 < \lceil \frac{m}{2} \rceil$.
b) A pattern that is not dense lattice periodic.



a b c

Figure 3:
a) Pattern P b) Two compatible copies. Here $Witness[1,2] = [6,6]$.
c) Two incompatible copies. Here $Witness[1,1] = [1,2]$ where a pair of mismatched elements (arrowed) is: $P[1,2] = X \neq P[1+1, 1+2] = P[2,3] = 0$.

4 Pattern Preprocessing

We begin with the preprocessing part of the algorithm. Its input is an $m \times m$ pattern P. The output of this step is 1) a classification of the pattern as being dense or sparse and 2) a table $Witness[-\frac{m}{2} \dots \frac{m}{2}, 0 \dots \frac{m}{2}]$.

4.1 Witness Idea

The *witness* idea was introduced by Vishkin [Vis85]. Suppose we are given two overlapping candidate patterns in the text. It may turn out that the two candidates can not coexist because they disagree in the area of overlap. The witness table gives us a location in their overlap where the candidates disagree on the character. By looking at the text character at that location, at least one of the candidates can be eliminated. This is the *duel paradigm* [Vis85].

More formally, our table $Witness$ indicates whether two copies, P_1 and P_2, of the pattern can overlap without mismatch. Let the two copies be offset so that the origin $P[0,0]$ of P_2 overlaps $P[i,j]$ of P_1. If the area of overlap does not mismatch, we call the candidates *compatible* and indicate this by setting $Witness[i,j] = [m,m]$. Otherwise, some pair of elements in the overlap mismatches. The candidates are incompatible. Let $Witness[i,j] = [a,b]$. Then one such mismatched pair of overlapping positions is $P[a,b]$ in P_2 and $P[i+a, j+b]$ in P_1. See figure 3 for an example.

We apply the witness information in the following way (figure 4). Suppose we have two candidate patterns P_1 and P_2 in the text T. Perform a duel with P_1 and P_2. Suppose that P_1 has origin at $T[r,c]$ and that P_2 has origin at $T[r+i, c+j]$. (This is the same offset as described above.) Then by lookup in the $Witness$ table, we find that these patterns are *incompatible* and that a witness exists in location $P[a,b]$ relative to the origin of P_2. We now examine $T[r+i+a, c+j+b]$. The character that occurs there mismatches either $P[a,b]$, ruling out P_2 or it mismatches $P[i+a, j+b]$, ruling out P_1 or it mismatches and rules out both.

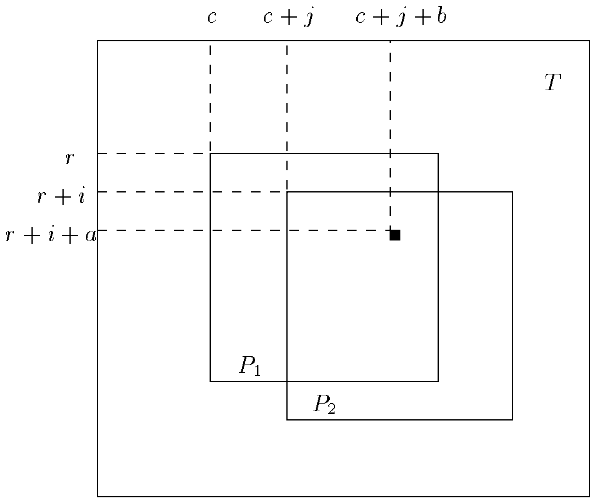


Figure 4:
The character at $T[r+i+a, c+j+b]$ rules out one or both candidate patterns.

The witness table construction can be done in time $O(\log m)$ using $O(m^2)$ CRCW processors or in $O(\log^2 m)$ time using $O(m^2/\log m)$ CREW processors [AB92]. A more recent optimal parallel algorithm is described in [CCG+93].

5 Block Compatibility

In the first phase of the text processing, we find compatible candidates using the dueling paradigm. Initially, we assume that every location in the text is a candidate source. In the *block compatiblity step*, we partition the text into disjoint blocks of size $\frac{m}{2} \times \frac{m}{2}$ and reduce the number of candidates so that any two candidates within a block are compatible. We eliminate candidates only when they are inconsistent with the text. If the pattern is dense, the blocks are built up in stages from smaller blocks. As we combine blocks, we maintain the property of compatibility of sources within a block. If the pattern is *sparse*, then we treat the columns of the text independently, dividing them into disjoint strips of length $\frac{m}{2}$, by joining smaller strips and maintaining compatibility of the candidates within the strips. Finally, $\frac{m}{2}$ of the strips of length $\frac{m}{2}$ are joined into a block, and all the candidates within the block are simultaneously tested for compatibility with each other.

Definition: A *k-block* B is a subbarray of the text T of size $2^k \times 2^k$. Specifically, *k-block* $B[i,j]$ for $k = 0, \ldots, \log m - 1$, $i, j = 0, \ldots, \frac{n}{2^k} - 1$ is:

$$T[i \cdot 2^k + 1 \ldots (i+1) \cdot 2^k, j \cdot 2^k + 1 \ldots (j+1) \cdot 2^k]$$

Algorithm A *Block Compatibility*

Input: Text T, pattern P and *Witness* table.

Output: $Cand[i,j]$ where $Cand[i,j] = \mathtt{T}$ if $T[i,j]$ is a candidate source and $Cand[i,j] = \mathtt{F}$ otherwise, such that for each $(\log m - 1)$-*block* of text (size $\frac{m}{2} \times \frac{m}{2}$), the set of candidates within the block are compatible and no candidate has been eliminated except when it is inconsistent with the text.

In the next two sections, we describe the different procedures for Algorithm A based on whether the pattern is dense or sparse.

5.1 Sparse Case

We stated earlier that a sparse pattern is one that is not dense lattice periodic. In this section, though, we will treat as sparse patterns that do not meet the criteria of the following lemma. We have kept the definition of dense patterns as stated earlier because the lemmas we prove in the following section are of interest in their own right and because the definition has already proven useful in another context [ABF94].

Lemma 1 *If, a pattern is periodic in both its first row and first column, then the pattern is dense lattice periodic.*

Proof: If the pattern is periodic in its first row, then there is a quadrant I periodicity vector $\mathbf{v}_1 = r_1\mathbf{y} + c_1\mathbf{x}$ with $r_1 = 0$ and $c_1 < \lceil \frac{m}{2} \rceil$. Similarly, if the pattern is periodic in its first column, then there is a quadrant II periodicity vector $\mathbf{v}_2 = r_2\mathbf{y} + c_2\mathbf{x}$ with $r_2 < \lceil \frac{m}{2} \rceil$ and $c_2 = 0$. The vectors meet the restricitions for dense lattice periodic patterns. ▮

By lemma 1, patterns that are sparse cannot be periodic in both the first column and the first row. This means either that overlapping sparse patterns aligned on their first column with a vertical separation of less than $\frac{m}{2}$, or overlapping sparse patterns aligned on their first row with a horizontal separation of less than $\frac{m}{2}$, have a mismatch. But, now, we will also consider dense lattice periodic patterns that are not periodic in both the first row and first column to be sparse. From the pattern preprocessing, we can determine if a pattern is periodic as stated above and if not, in which direction, either row or column, the pattern is not periodic. For the remainder of this section we will assume that the pattern is not column periodic.

We want to produce $(\log m - 1) - \mathit{blocks}$ containing compatible candidates. We will describe the procedure for a single block B. All blocks are done in parallel. The procedure consists of two parts. First (A1) we eliminate all but, at most, one candidate in each column of B. Second (A2), we simultaneously compare all the candidates in the columns of B for compatibility.

Procedure A1 *Column Sparseness*

The idea is the following. Build a binary tree with the elements in the column as the leaves. Starting at the leaves, look up the witness for consecutive pairs of candidates. At most one candidate

Procedure A2: Join Strips

Step 1: Test Compatibility

for each $(\log m - 1)$-*block* B **pardo**

for each candidate pair C_α, C_β in B **pardo**

- Let $[i, j]$ be the offset of the candidates.
  ($i.e.$ origin of C_α is at $T[r, c]$ and origin of C_β is at $T[r+i, c+j]$.)
- If $Witness[i, j] = [a, b] \neq [m, m]$ (incompatible)
  then let σ (witness character)$=T[r+i+a, c+j+b]$.
- If $\sigma \neq P[a, b]$ then $Mark[\beta, \alpha] = 1$. (C_β loses.)
- If $\sigma \neq P[i+a, j+b]$ then $Mark[\alpha, \beta] = 1$. (C_α loses.)

Step 2: Eliminate Candidates

for each $(\log m - 1)$-*block* B **pardo**

for each candidate C_α **pardo**

- Let $T[i, j]$ be the source for candidate C_α.
  $Cand[i, j] =$**T** if $(\bigvee_\beta Mark[\alpha, \beta] = 0)$ else **F**.

Figure 5:

survives to the next stage because the pattern is not periodic in a column. Continue up the tree, performing duels between surviving candidates. At the top, at most one candidate survives. This simple description has complexity $O(\log m)$ time and $O(m)$ processors. Breslauer and Galil [BG90] showed that for a non-periodic string of length m, the algorithm above can be modified to find the surviving candidate within a text segment of length $m/2$ in linear work in $O(\log \log m)$ time on a CRCW PRAM since it is analagous to computing the maximum of $m/2$ numbers. Using a similar method to reduce the work, we find a single surviving candidate in each column in $O(\log m)$ time with $O(m/\log m)$ CREW processors.

Procedure A2 *Join Strips*

There is now one suvivor per column (of length $\frac{m}{2}$), so we have $\frac{m}{2}$ candidates in each $(\log m - 1)$-*block*. In constant time, we now perform all pairwise compatibility checks (duels) within each block. Finally, we determine which candidates survive all such comparisons. Let $Mark$ be an $\frac{m}{2} \times \frac{m}{2}$ table to hold the results of the duels. All entries in $Mark$ are initialized to zero. See figure 5.

Theorem 1 *Procedures Column Sparseness and Join Strips are correct and run in linear work and space in time $O(\log m)$ on a CREW PRAM and in time $O(\log \log m)$ on a CRCW PRAM.*

Proof: The correctness of the procedures follows from the preceding comments. As pointed out above, Procedure Column Sparseness executes within the desired bounds. In Procedure Join Strips, there are $\frac{m}{2}$ candidates per block of size $\frac{m}{2} \times \frac{m}{2}$. We perform all $O(m^2)$ duels in $O(1)$ time and $O(m^2)$ space to store the marks. Then, each candidate must perform an `OR` over the answers of its duels to see if it is eliminated. Once again, the `OR` is computable optimally within the desired bounds [JáJ92]. ∎

5.2 Dense Case

When the pattern is dense, a text block of size $k \times k$ may contain as many as k^2 candidates so the column sparseness will not help us here. We use a different technique. As in Procedure Column Sparseness, the method is analagous to computing the maximum of a list of numbers. Our goal is to eliminate enough candidates so that within each $(\log m - 1)$-*block*, all candidates are compatible.

Procedure A3 *Blocks Merge*

At each stage k, we will effectively do all pairwise duels between members of sets of k-*blocks*. The size of the sets of merged blocks will be noted below, but depends on the type of processor used. The algorithm for finding the maximum (`MAX`) of a list of numbers [JáJ92] works by building a tree as described in procedure Column Sparseness in the previous section. It proceeds by finding the maximum of larger and larger blocks and relies on the fact that each block has a single maximal value. Since each of our blocks may contain more than one candidate, if we adopt the `MAX` algorithm, we need to show how to compare all candidates within two blocks in constant work and how to eliminate a block of candidates in constant time.

Having presented these subprocedures, we can then simply apply the standard `MAX` algorithm [JáJ92] which, on a CREW PRAM combines 2 blocks per stage, giving a running time of $O(\log m)$, and on a CRCW PRAM combines 2^{2^i} blocks in stage i, giving a running time of $O(\log\log m)$.

5.2.1 Comparing Blocks in constant work

For each $k \times k$ block, we have only $\frac{k^2}{\log m}$ CREW or $\frac{k^2}{\log\log m}$ CRCW processors available. Therefore, if we require that every candidate look at a witness, we would need super-constant time per stage. Instead, we use representative candidates from each block and find the witnesses (if they exist) for those candidates. We will treat the representative of a block as the `MAX` value of this computation. Initially, the representative is the single candidate in its 0-*block*. In order to specify the representative candidate to be carried up the tree after a duel, we define $>$ over two representatives as follows:

Definition: For two representatives X and Y, let $X \gg Y$ mean that X occurs first in a lexicographic ordering of the indices, first by row and then by column. Then, $X > Y$ if one of the following three conditions holds:

1. X and Y are compatible and $X \gg Y$.

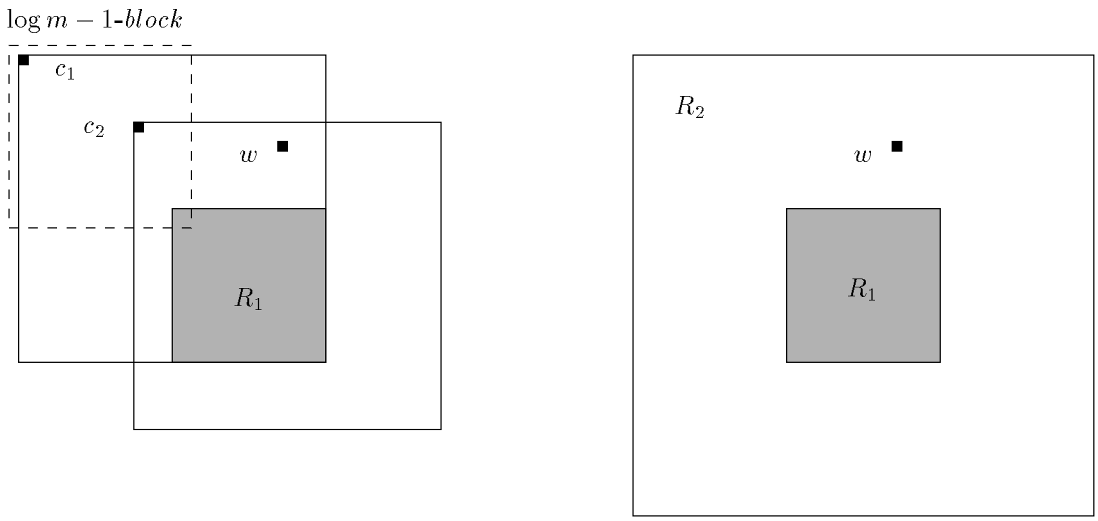


Figure 6:
Any witness w occurs within $\frac{m}{2}$ rows and or columns of R_1.

2. X is incompatible with Y and X wins a duel with Y.

3. X is incompatible with Y, neither is consistent with the text and $X \gg Y$.

Example: Let candidate X occur at $T[4, 8]$ and candidate Y occur at $T[4, 12]$. Note that $X \gg Y$. If X and Y are compatible, then $X > Y$. If Y wins a duel with X, then $Y > X$. If both X and Y are found to be inconsistent with the text through a duel, then $X > Y$.

We want a witness that can eliminate all the candidates in either of the two blocks. But, the witness w we get from the witness table may not fall within the common overlap of all the candidates in the two blocks. In this case, it is *always* possible to find an alternate witness w' that *does* fall within that overlap. In the following two lemmas, we show that the region of common overlap of all the candidates for the two blocks has dimensions $\lceil \frac{m}{2} \rceil \times \lceil \frac{m}{2} \rceil$, the witness w lies within $\lceil \frac{m}{2} \rceil$ rows and or columns of this region and that such a region must always contain an alternate witness.

Lemma 2 *If two candidate c_1 and c_2, have sources within the same $(\log m - 1)$-block, then the witness w for those two candidates (if it exists) lies within $\lceil \frac{m}{2} \rceil$ rows and/or columns of the common overlap of all the candidates in the block, which has size $\lceil \frac{m}{2} \rceil \times \lceil \frac{m}{2} \rceil$.*

Proof: (Figure 6.) The common overlap for all possible candidates with sources in the $\log m$-*block* $B[i, j]$ is $T[(i+1) \cdot \lceil \frac{m}{2} \rceil \ldots (i+2) \cdot \lceil \frac{m}{2} \rceil, (j+1) \cdot \lceil \frac{m}{2} \rceil \ldots (j+2) \cdot \lceil \frac{m}{2} \rceil] = R_1$. Note that R_1 has size $\lceil \frac{m}{2} \rceil \times \lceil \frac{m}{2} \rceil$. The witness w for any pair of candidates c_1 and c_2 occurs somewhere in the region $T[i \cdot \lceil \frac{m}{2} \rceil \ldots (i+3) \cdot \lceil \frac{m}{2} \rceil, j \cdot \lceil \frac{m}{2} \rceil \ldots (j+3) \cdot \lceil \frac{m}{2} \rceil] = R_2$, depending upon the exact locations of c_1 and c_2 and their witness. Any point in R_2 is within $\lceil \frac{m}{2} \rceil$ rows and/or columns of R_1. ∎

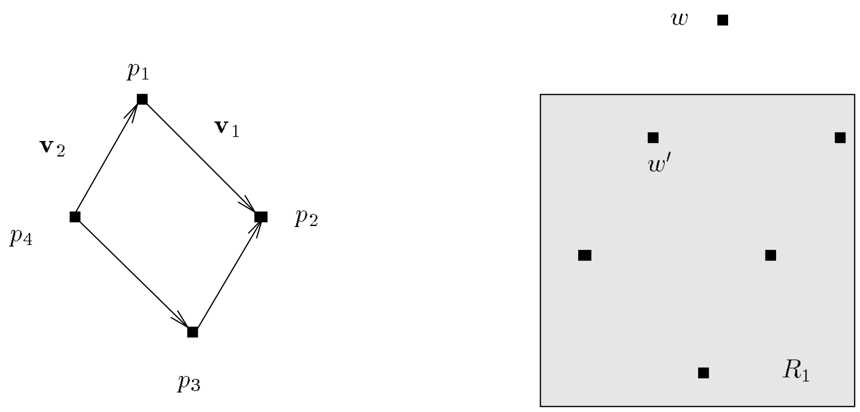


Figure 7:
Region R_1 cannot fit within the rectangular area bounded by a unit cell.

Lemma 3 *If* $\mathbf{v}_1 = r_1\mathbf{y} + c_1\mathbf{x}$ *and* $\mathbf{v}_2 = r_2\mathbf{y} + c_2\mathbf{x}$ *are the basis vectors of a dense lattice periodic pattern, and a lattice with such basis vectors is laid over the text* T*, then every block of size* $\lceil \frac{m}{2} \rceil \times \lceil \frac{m}{2} \rceil$ *must contain a lattice point.*

Proof: (Figure 7.) Consider a unit cell of the lattice on the text T. Label the nodes of the cell, in clockwise order, p_1, p_2, p_3 and p_4, where p_1 is the upper left corner of the cell. By the definition of basis vectors from [AB92], $\mathbf{p_1p_2} = \mathbf{v}_1 = r_1\mathbf{y} + c_1\mathbf{x}$ and $\mathbf{p_4p_1} = \mathbf{v}_2 = r_2\mathbf{y} + c_2\mathbf{x}$. Because the pattern is lattice periodic, all of $|r_1|, |r_2|, |c_1|, |c_2| < \lceil \frac{m}{2} \rceil$. Therefore, for an $\lceil \frac{m}{2} \rceil \times \lceil \frac{m}{2} \rceil$ block of text to fit within the lattice cell without containing a lattice point, it must fit within the rectangular area bounded by such a cell. (It can not, for example, squeeze between two adjacent nodes.) But, the largest such rectangular region has at most $\lceil \frac{m}{2} \rceil - 2$ columns if $|c_1| + |c_2| < \lceil \frac{m}{2} \rceil$ or at most $\lceil \frac{m}{2} \rceil - 2$ rows if $|r_1| + |r_2| < \lceil \frac{m}{2} \rceil$. ∎

By lemma 3, an alternate witness for w must lie within R_1. Additionally, it must lie on a point of the lattice defined by the basis vectors of the pattern, $\mathbf{v}_1$ and $\mathbf{v}_2$, and with origin at w. In order to find an alternate witness w', we will store the locations of the lattice points in an array *Alt*. Then, *Alt* will indicate the position of an alternate witness, permitting lookup in constant time. Since w always lies within $\lceil \frac{m}{2} \rceil$ rows and/or columns of R_1 (by lemma 2), the size of *Alt* is linear in the size of the pattern.

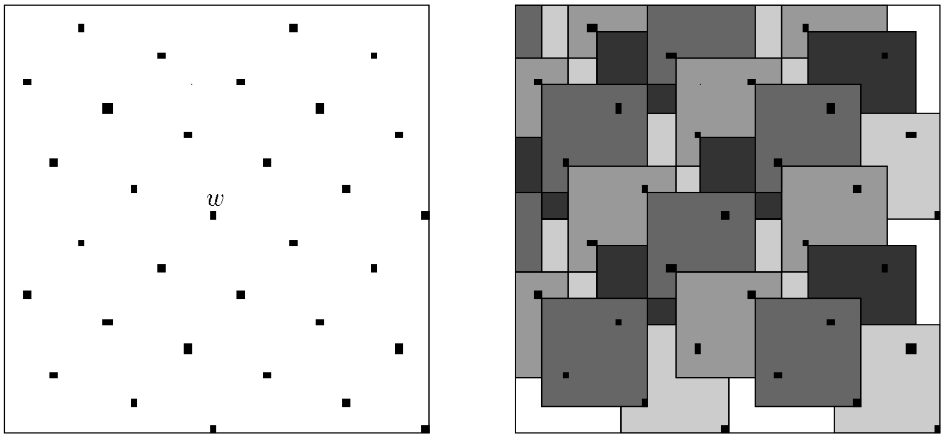
Let

$$r_{i,j} = i \cdot r_1 + j \cdot r_2$$

$$c_{i,j} = i \cdot c_1 + j \cdot c_2$$

be the coordinates of the lattice points reached from w along i vectors $\mathbf{v}_1$ and j vectors $\mathbf{v}_2$. We will precompute the locations of all lattice points within an $2m \times 2m$ block around w.

Procedure A3.1 *Alternate Witness Table*



a b

Figure 8:
a) The lattice points in a $2m \times 2m$ square around w.
b) The regions of propagation for the alternate witnesses. Whenever the upper left corner of R_1 falls within a shaded region, a specific alternate witness is specified.

For all $i, j = -m, \ldots, m$ pardo:

If $-m \leq r_{i,j} \leq m$ and $-m \leq c_{i,j} \leq m$
then $Alt[r_{i,j}, c_{i,j}] = [r_{i,j}, c_{i,j}]$.

Example:

Let $\mathbf{v}_1 = (2, 3)$, $\mathbf{v}_2 = (-1, 4)$ and $m = 10$.

Then, some values assigned by Procedure A3.1 are:
$Alt[0, 0] = [0, 0]$
$Alt[2, 3] = [2, 3]$
$Alt[-2, -3] = [-2, -3]$
$Alt[1, 7] = [1, 7]$
$Alt[-1, 4] = [-1, 4]$

The location of the lattice points with respect to w are now stored in Alt. In the remainder of this procedure, we fill out the remaining entries in Alt by propagating the locations of the lattice points up and left. The method is similar to that described in section 6 and will not be further described here. A typical result is illustrated in figure 8.

When we find the witness for two blocks $B_1 = B(i, j)$ and $B_2 = B(s, t)$, The upper left corner of region R_1 is at $T[(max(i, s) + 1) \cdot 2^k, (max(j, t) + 1) \cdot 2^k]$. If w is at $T[r_w, c_w]$ then the alternate witness w' is at $Alt[(max(i, s) + 1) \cdot 2^k - r_w, (max(j, t) + 1) \cdot 2^k - c_w]$.

5.2.2 Eliminating Candidates

When we run the `MAX` algorithm on the blocks, we only have constant time per comparison to eliminate all the surviving candidates per block. Since this can in general take too long, we mark blocks for elimination rather than eliminate the candidates immediately. We simply keep a tree which mimics the computation tree, i.e. if two blocks are compared in some stage i, they are sibling nodes at height i from the leaves. Now, if some node needs deleting, by which we mean that all the candidates in the block represented by the node do not survive, we mark the node for deletion.

Such a deletion tree is of size $O(m^2)$. Once constructed, we need only start at the root and propagate the deletion bits down to the leaves. On either a CRCW or CREW PRAM, the processor allocation problem is staightforward and we can finish the computation in time proportional to the depth of the tree, i.e. within the same bounds as the `MAX` algorithm.

We summarize the discusion above with the following theorem.

Theorem 2 *Procedure Blocks Merge is correct and runs in linear work and space in time* $O(\log m)$ *on a CREW PRAM and in time* $O(\log\log m)$ *on a CRCW PRAM.*

6 Verification

Within each $(\log m - 1)$-*block*, all remaining candidates are now mutually compatible. For each candidate, our final goal is to compare all candidate text elements with their corresponding elements in the pattern. If none of the comparisons results in a mismatch, then the candidate is an actual occurrence of the pattern.

Note that because patterns may overlap, the same text element may appear in many candidates. Yet, we do not want to test each text element more than a constant number of times. What comes to our aid is the fact that compatible candidates agree on their areas of overlap. This leads to the following crucial observation: Given a set of compatible candidates and a single text element, t, which they all overlap, the consistency of all the candidates with the text at t can be determined by comparing t to a *single* pattern element. If the comparison is a match, then *all* the candidates are consistent with the text at t. If the comparison is a mismatch, then all the candidates are inconsistent with the text.

What remains now is to show how to select a pattern element to be compared with each text element and how to inform the candidates of the result of the test (propagation).

We use sets of candidates within each $(\log m - 1)$-*block*. Each text element $T[r, c]$ may be contained by several candidates, the *relevant* candidates. Let the *home block*, $B[b(r), b(c)]$, be the $(\log m - 1)$-*block* containing $T[r, c]$, where the block index $b(i) = \lfloor i/\frac{m}{2} \rfloor$. Note that $T[r, c]$ may have relevant candidates in each of nine blocks, namely $B[b(r) - s, b(c) - t]$, $s, t = 0 \ldots 2$. We call these nine blocks the *relevant* blocks (figure 9).

Compatible candidates that are relevant to the same text element must agree on the expected

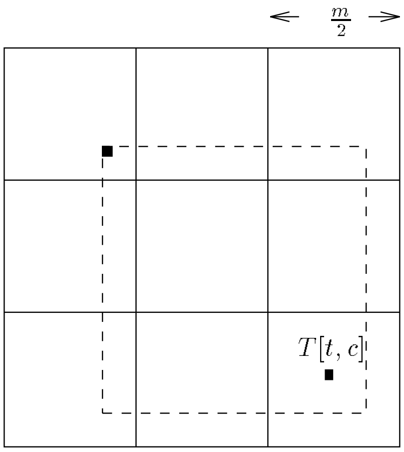


Figure 9:
Relevant candidates for $T[r,c]$ can have origins within nine $\log m$-*blocks*, including the home block. Here one relevant candidate is shown.

character in that element. Thus every element $T[r,c]$ can be labeled with a matrix $M_{r,c}[s,t] \in \{\texttt{true}, \texttt{false}\}$ where $M_{r,c}[s,t] = \texttt{true}$ means that $T[r,c]$ equals the unique pattern symbol expected by all relevant candidates in block $B[b(r)-s, b(c)-t]$ and $M_{r,c}[s,t] = \texttt{false}$ otherwise. Thus, every text element needs to be compared to a *single* pattern element per relevant block, and every candidate source that contains an element with the appropriate entry of M set to `false` is not a pattern appearance and can be discarded.

We proceed independently for each block.

Algorithm B *Candidate Verification for block* $B[i,j]$

> Step B.1: For every text location $T[r,c]$ with a relevant candidate in $B[i,j]$ record a *pattern coordinate pair* $\langle x,y\rangle$, where $\langle x,y\rangle$ are the coordinates of the pattern element $P[x,y]$ with which $T[r,c]$ should be compared.

There may be several options for some locations, namely, the position of the text element relative to each of its relevant candidates. However, any will do since all candidate sources within block $B[i,j]$ are now compatible. If a location is not contained in any candidate it is left unmarked. We will later see how this step is implemented.

> Step B.2: Compare each text location $T[r,c]$ with $P[x,y]$, where $\langle x,y\rangle$ is the pattern coordinate pair assigned to $T[r,c]$ in the previous step. If $T[r,c] = P[x,y]$, then $M_{r,c}[b(r)-i, b(c)-j] \leftarrow$ `true`, else `false`.

Step B.3: Flag with a *discard* every candidate that contains a `false` location within its bounds.

This flagging is done by the same method as in step B.1.

Step B.4: Discard every candidate source flagged with a *discard*. The remaining candidates represent all pattern appearances.

Our only remaining task is to show how to mark the text elements with the appropriate pattern coordinate pairs. The implementation of this step relies on the following lemma.

Lemma 4 *[FRA88] The first and last one in an array of length n containing zeros and ones can be determined in $O(1)$ time with $O(n)$ processors on a CRCW PRAM.*

Similarly, finding the first and last one in an array of length n can be accomplished optimally in $O(\log n)$ time on a CREW PRAM using a prefix sum [JáJ92].

For each block $B[i,j]$, we have an array $C_{i,j}[1 \ldots \frac{3m}{2}, 1 \ldots \frac{3m}{2}]$ in which we will do our computation. All ones in the array represent surviving candidates in $B[i,j]$ and all other positions contain zeros. The information in $C_{i,j}[r,c]$ applies to text element $T[i \cdot \frac{m}{2} + r, j \cdot \frac{m}{2} + c]$. For clarity, we now refer to this as text location $T[r,c]$ and drop the subscripts on C.

The goal is to assign to each position $C[x,y]$ a pair (a,b) such that $C[a,b] = 1$ and both $0 \leq x - a < m$ and $0 \leq y - b < m$. We refer to such a relevant candidate as the *ruler* of $C[x,y]$. We proceed first within columns and then within rows (figure 10).

Independently for each column c, if column c contains all zeros, do nothing. Otherwise, find the first and last occurrence of a "one" (candidate). Call the row position of these "ones" r_f and r_l respectively. Now we have several cases:

$r < r_f$: Do nothing. There is no "one" in column c relevant to $C[c,r]$.

$r_f \leq r < r_l$: Set the ruler of $C[c,r]$ to be (c, r_f). Since $r_l - r_f \leq \frac{m}{2}$, then the candidate at $C[c, r_f]$ is close enough to $C[c,r]$ to be relevant.

$r_l \leq r < r_l + m$: Set the ruler of $C[c,r]$ to be (c, r_l). Similarly, the candidate at $C[c, r_l]$ is close enough to $C[c,r]$ to be relevant.

$r_l + m \leq r$: Do nothing. $C[c,r]$ is too far from any candidate in column c.

Now we propagate along the rows with the following modifications. Instead of finding the first and last candidate of each row, we find the the first and last position which has been assigned a ruler in the previous phase.

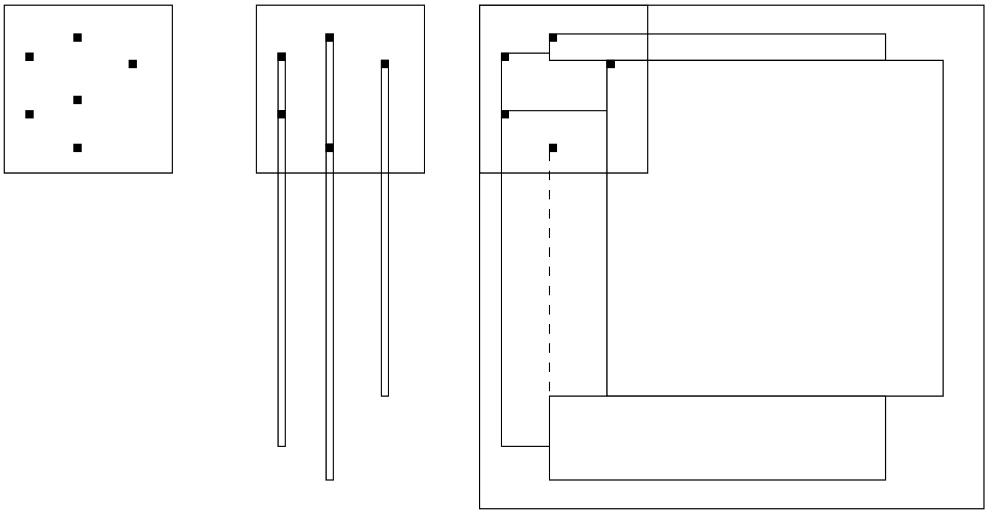


Figure 10:
a)Candidates in $B[i,j]$.
b)First and last rulers in the columns
c) First and last rulers propagated along the rows.

Theorem 3 *Algorithm B is correct and runs in time $O(\log m)$ with $O(m^2/\log m)$ CREW processors per block and linear space, thus $O(\log m)$ time, $O(n^2/\log m)$ CREW processors and linear space for all blocks. Similarly, the algorithm runs optimally in constant time on a CRCW with linear space.*

Correctness: We need only show that if $C[a,b]$ is assigned ruler (x,y), then $T[x,y]$ is a candidate. But this follows directly from the case analysis above. The symmetric process of finding, for each candidate, if it has a relevant error can clearly be computed within the same bounds.

Time: The bottleneck in the compuation is finding the first and last "one" in each column and row. By lemma 4, this step takes exactly the stated bounds. ∎

Aknowledgements

The authors wish to thank S. Muthukrishnan and H. Ramesh for fruitful discussion.

References

[AB92] A. Amir and G. Benson. Two-dimensional periodicity and its application. *Proc. of the Third Ann. ACM-SIAM Symp. on Discrete Algorithms*, Jan 1992.

[ABF92a] A. Amir, G. Benson, and M. Farach. Alphabet independent two-dimensional matching. *Proc. of the 24th Ann. ACM Symp. on Theory of Computing*, pages 59–68, 1992.

[ABF92b] A. Amir, G. Benson, and M. Farach. The truth, the whole truth, and nothing but the truth: Alphabet independent two dimensional witness table construction. Technical Report GIT-CC-92/52, Georgia Institute of Technology, Aug 1992.

[ABF93] A. Amir, G. Benson, and M. Farach. Optimal parallel two dimensional pattern matching. *Proc. of the 5th ACM Symp. on Parallel Algorithms and Architectures*, pages 79–85, 1993.

[ABF94] A. Amir, G. Benson, and M. Farach. Optimal two dimensional compressed matching. *Proc. of 21st International Colloquium on Automata Languages and Programming*, 1994.

[AC75] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Comm. ACM*, 18:333–340, 1975.

[Bak78] T. J. Baker. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM J. Comp*, 7:533–541, 1978.

[BG90] D. Breslauer and Z. Galil. An optimal $o(\log\log n)$ time parallel string matching algorithm. *SIAM J. Comput.*, 19:1051–1058, 1990.

[Bir77] R. S. Bird. Two dimensional pattern matching. *Information Processing Letters*, 6:168–170, 1977.

[BM77] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Comm. ACM*, 20:762–772, 1977.

[CCG+93] R. Cole, M. Crochemore, Z. Galil, L. Gąsieniec, R. Harihan, S. Muthukrishnan, K. Park, and W. Rytter. Optimally fast parallel algorithms for preprocessing and pattern matching in one and two dimensions. *Proc. 34th IEEE FOCS*, pages 248–258, 1993.

[CDR86] S.A. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneus writes. *SIAM Journal on Computing*, 15(1):87–97, 1986.

[FRA88] F. Fich, R. Ragde, and A.Widgerson. Relations between concurrent-write models of parallel computation. *SIAM J. Comput.*, 17:606–627, 1988.

[Gal92] Z. Galil. A constant-time optimal parallel string-matching algorithm. *Proc. of the 24th Ann. ACM Symp. on Theory of Computing*, pages 69–76, 1992.

[GP92] Z. Galil and K. Park. Truly alphabet independent two-dimensional pattern matching. *Proc. of the 33rd IEEE Annual Symp. on Foundation of Computer Science*, 1992.

[JáJ92] J. JáJá. *An introduction to parallel algorithms.* Addison-Wesley, 1992.

[KLP89] Z. M. Kedem, G. M. Landau, and K. V. Palem. Optimal parallel suffix-prefix matching algorithm and application. *Proc. of the First Ann. ACM Symp. on Parallel Algorithms and Architectures*, pages 388–398, 1989.

[KMP77] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comp.*, 6:323–350, 1977.

[RK82] A. Rosenfeld and A.C. Kak. *Digital Picture Processing.* Academic Press, New York, 1982.

[Vis85] U. Vishkin. Optimal parallel pattern matching in strings. *Proc. 12th ICALP*, pages 91–113, 1985.

[Vis91] U. Vishkin. Deterministic sampling - a new technique for fast pattern matching. *SIAM J. Comp.*, 20:303–314, 1991.