1989

Constructing Trees in Parallel

Mikhail J. Atallah
Purdue University, mja@cs.purdue.edu

S. R. Kosaraju

L. L. Larmore

G. L. Miller

Report Number:
89-883

CONSTRUCTING TREES IN PARALLEL

M. J. Atallah
S. R. Kosaraju
L. L. Larmore
G. L. Miller
S-H. Teng

CSD-TR-883
May 1989

Constructing Trees in Parallel

M. J. Atallah* S. R. Kosaraju† L. L. Larmore‡ G. L. Miller§ S-H. Teng§

Abstract

$O(\log^2 n)$ time, $n^2/\log n$ processor as well as $O(\log n)$ time, $n^3/\log n$ processor CREW deterministic parallel algorithms are presented for constructing Huffman codes from a given list of frequencies. The time can be reduced to $O(\log n(\log\log n)^2)$ on a CRCW model, using only $n^2/(\log\log n)^2$ processors. Also presented is an optimal $O(\log n)$ time, $O(n/\log n)$ processor EREW parallel algorithm for constructing a tree given a list of leaf depths when the depths are monotonic. An $O(\log^2 n)$ time, n processor parallel algorithm is given for the general tree construction problem. We also give an $O(\log^2 n)$ time $n^2/\log^2 n$ processor algorithm which finds a nearly optimal binary search tree. An $O(\log^2 n)$ time $n^{2.36}$ processor algorithm for recognizing linear context free languages is given. A crucial ingredient in achieving those bounds is a formulation of these problems as multiplications of special matrices which we call *concave* matrices. The structure of these matrices makes their parallel multiplication dramatically more efficient than that of arbitrary matrices.

*Department of Computer Science, Purdue University. Supported by the Office of Naval Research under Grants N00014-84-K-0502 and N00014-86-K-0689, and the National Science Foundation under Grant DCR-8451393, with matching funds from AT&T.

†Department of Computer Science, Johns Hopkins University. Supported by National Science Foundation through grant CCR-88-04284

‡ICS, UC Irvine.

§School of Computer Science, CMU and Department of Computer Science, USC. Supported by National Science Foundation through grant CCR-87-13489.

1 Introduction

In this paper we present several new parallel algorithms. Each algorithm uses substantially fewer processors than used in previously known algorithms. The four problems considered are: The Tree Construction Problem, The Huffman Code Problem, The Linear Context Free Language Recognition Problem, and The Optimal Binary Search Tree Problem. In each of these problems the computationally expensive part of the problem is finding the associated tree. We shall show that these trees are not arbitrary trees but are special. We take advantage of the special form of these trees to decrease the number of processors used.

All of the problems we consider in this paper, as well as many other problems, can be performed in sequential polynomial time using Dynamic Programming. $\mathcal{NC}$ algorithms for each of these problems can be obtained by parallelization of Dynamic Programming. Unfortunately, this approach produces parallel algorithms which use $O(n^6)$ or more processors. An algorithm which increases the work performed from $O(n)$ or $O(n^2)$ to $O(n^6)$ is not of much practical value. In this paper we present several new paradigms for improving the processor efficiency for dynamic programming problems. For all the problems considered a tree or class of trees is given implicitly and the algorithm must find one such tree.

The construction of optimal codes is a classical problem in communication. Let $\Sigma = \{0, 1, ..., \sigma - 1\}$ be an *alphabet.* A *code* $\mathcal{C} = \{c_1, \ldots, c_n\}$ over Σ is a finite nonempty set of distinct finite sequences over Σ. Each sequence c_i is called *code word.* A code $\mathcal{C}$ is a *prefix code* if no code-word in $\mathcal{C}$ is a prefix of another code-word. A *message* over $\mathcal{C}$ is a word resulting from the concatenation of code words from $\mathcal{C}$.

We assume the words over a source alphabet $a_1, \ldots, a_n$ are to be transmitted over a communication channel which can transfer one symbol of Σ per unit of time, and the probability of appearance of a_i is $p_i \in \mathcal{R}$. The **Huffman Coding Problem** is to construct a prefix code $\mathcal{C} = \{c_1, \ldots, c_n \in \Sigma^*\}$ such

that the average word length $\Sigma_{i=1}^{n} p_i \cdot |c_i|$ is minimum, where $|c_i|$ is the length of c_i.

It is easy to see that prefix codes have the nice property that a message can be decomposed in code word in only one way– they are uniquely decipherable. It is interesting to point out that Kraft and McMillan proved that for any code which is uniquely decipherable there is always a prefix code with the same average word length [13]. In 1952, Huffman [9] gave an elegant sequential algorithm which can generate an optimal prefix code in $O(n \log n)$ time. If the probabilities are presorted then his algorithm is actually linear time [11]. Using parallel dynamic programming, Kosaraju and Teng [18], independently, gave the first $\mathcal{NC}$ algorithm for the Huffman Coding Problem. However, both constructions use n^6 processors. In this paper, we first show how to reduce the processor count to n^3, while using $O(\log n)$ time, by showing that we may assume that the tree associated with the prefix code is left-justified (to be defined in Section 2).

The n^3 processor count arises from the fact that we are multiplying $n \times n$ matrices over a closed semiring. We reduce the processor count still further to $n^2/\log n$ by showing that, after suitable modification, the matrices which are multiplied are concave (to be defined later). The structure of these matrices makes their parallel multiplication dramatically more efficient than that of arbitrary matrices. An $O(\log n \log\log n)$ time $n^2/\log n$ processor CREW algorithm is presented for multiplying them. Also given is an $O((\log\log n)^2)$ time, $n^2/\log\log n$ processor CRCW algorithm for multiplying two concave matrices[1].

The algorithm for construction of a Huffman code still uses n^2 processors, which is probably too large for practical consideration since Huffman's algorithm only takes $O(n \log n)$ sequential time. Shannon and Fano gave a code, the Shannon-Fano Code, which is only one bit off from optimal. That is, the expected length of a Shannon-Fano code word is at most one bit longer than the Huffman code word.

The construction of the Shannon-Fano Code reduces to the following **Tree Construction Problem**,

Definition 1.1 (Tree Construction Problem) *Given n integer values $l_1, \ldots, l_n$, construct an ordered binary tree with n leaves whose levels when read form left to right are $l_1, \ldots, l_n$.*

[1] Independently, [1] and [2] improved the CREW algorithm results by showing that two concave matrices can be multiplied in $O(\log n)$ time, using $n^2/\log n$ CREW PRAM processors. Also, [2] improved the CRCW algorithm by reducing the number of CRCW PRAM processors required to $n^2/(\log\log n)^2$.

We give an $O(\log^2 n)$ time, n processor EREW PRAM parallel algorithm for the tree construction problem. In the case when $l_1, \ldots, l_n$ are monotonic, we give an $O(\log n)$ time and $n/\log n$ processor EREW PRAM parallel algorithm. In fact, trees where the level of the leaves are monotone will be used for both constructing Huffman Codes and Shannon–Fano Codes.

Using our solution of the tree construction problem we get an $O(\log n)$ time $n/\log n$ processor EREW PRAM algorithm for constructing Shannon-Fano Codes.

We also consider the problem of parallel constructing optimal binary search trees as defined by Knuth [10]. The best known NC algorithm for this problem is the parallelization of dynamic programming which uses n^6 processors. In this paper, using the new concave matrix multiplication algorithm, we show how to compute nearly optimal binary search tree in $O(\log^2 n)$ time using $n^2/\log n$ processors. Our search trees are only off from optimal by an additive amount of $1/n^k$ for any fixed k.

Finally, we consider recognition of linear context free languages. A CFL is said to be linear if all productions are of the form $A \rightarrow bB$, $A \rightarrow Bb$ or $a \rightarrow A$ where A and B are nonterminal variables and a and b are terminal variables. It is well known from Ruzzo [17] that the *general CFL's recognition* problem can be performed on a CRCW PRAM in $O(\log n)$ time using n^6 processors again by parallelization of dynamic programming. By observing that the parse tree of the linear context free language is of very restricted form, we construct an $O(n^3)$ processor, $O(\log^2 n)$ time CREW PRAM algorithm for it. Using the fact that we are doing Boolean matrix multiplication, we can reduce the processor count to $n^{2.36}$.

2 Preliminaries

Throughout this paper a *tree* will be a rooted tree. It is *ordered* if the children of each node are ordered from left to right. The *level* of a node in a tree is its distance from the root. A binary tree $\mathcal{T}$ is *complete* at level l if there are 2^l nodes in $\mathcal{T}$ at level l. A binary tree is *empty* at level l if there is no vertex at level l.

A binary tree $\mathcal{T}$ is a *left-justified tree* if it satisfies the following property:

1. if a vertex has only one child, then it is a left child;
2. if u and v are sibling nodes of $\mathcal{T}$, where u is to the left of v, then if $\mathcal{T}_v$ is not empty at some level l,

then $\mathcal{T}_u$ is complete at level l, where T_u and $\mathcal{T}_v$ denote the subtrees rooted at u and v, respectively. *Right-justified* trees can be defined similarly.

Let *RAKE* be an operation that removes all leaves from a tree. We shall consider a restricted form of RAKE where leaves are removed only when its siblings are leaves.

Proposition 2.1 *The set of left-justified trees (right-justified trees) is closed under the* RAKE *operation.*

Lemma 2.1 *For any left-justified tree $\mathcal{T}$ of n vertices, $\lceil \log_2 n \rceil$ applications of* RAKE *will reduce $\mathcal{T}$ to a single chain. Moreover, the resulting chain comes from the left most path in $\mathcal{T}$.*

Proof: We need only show that any vertex not on the left most path of $\mathcal{T}$ is removed by $\lceil \log n \rceil$ iterations of RAKE.

Let v be a vertex of $\mathcal{T}$ not on the left most path, and let h be the *height* of v, the maximum length of a path from v to a leaf in $\mathcal{T}_v$. Since $\mathcal{T}$ is left-justified, there exists a vertex u of $\mathcal{T}$ at the same level as v and to the left of v, and since $\mathcal{T}_v$ is not empty at level h, $\mathcal{T}_u$ is complete at level h and hence has at least 2^h leaves. Since $\mathcal{T}$ has n leaves altogether, $h < \log n$. Each RAKE decreases the height of every non-empty subtree by 1, thus $\log n$ iterations of RAKE completely eliminate $\mathcal{T}_v$. □

Let the *height* of a tree $\mathcal{T}$ be the height of its root.

Corollary 2.1 *If $\mathcal{T}$ is a left-justified tree of n vertices, then for all v not on the left most path of $\mathcal{T}$, the height of $\mathcal{T}_v$ is bounded by $O(\lceil \log n \rceil)$.*

3 Parallel Tree Contraction and Dynamic Programming

In this section, we present a parallel algorithm for finding an optimal Huffman tree for a given monotonic frequency vector $(p_1, \ldots, p_n)$. The general Huffman Coding Problem is reducible to this special case after applying one sort (see Teng [18]).

For $1 \leq i \leq j \leq n$ let $H_{i,j}$ be defined to be the minimum average word length of a Huffman code over $(p_i, \ldots, p_j)$. Let $p_{i,j} = \sum_{s=i}^{j} p_s$. The values of H may be obtained recursively as follows: for all $1 \leq i \leq j \leq n$,

$$H_{i,j} = \begin{cases} 0 & i = j \\ \min_{k=i+1}^{j}(H_{i,k-1} + H_{k,j}) + p_{i,j} & i < j \end{cases} \quad (1)$$

The values of all $H_{i,j}$, including the desired output value $H_{1,n}$, may be obtained by the following algorithm which simulates the RAKE operation:

1. Estimate $H_{i,j}$ to be 0 if $i = j$, $+\infty$ otherwise.
2. Iterate this step until all $H_{i,j}$ are stable: Use relation (1) to re-estimate $H_{i,j}$ for all $i < j$, using the values of H obtained during the previous estimation step.
3. Output the value of $H_{1,n}$.

Each iteration of the second step can be done in $O(\log n)$ time using $n^3/\log n$ processors, if a CREW PRAM model of computation is used. Unfortunately, the best upper bound on the number of iterations needed is $O(n)$, since each iteration simulates just one RAKE operation.

The algorithm can be improved by introducing a step which simulates the COMPRESS operation, as well. The COMPRESS operation halves each chain in a tree by doubling. For any $1 \leq i < j \leq n$, define $F_{i,j}$ to be that quantity such that $H_{1,i} + F_{i,j}$ is the minimum average word length of a binary tree over $(p_1, \ldots, p_j)$, where the only trees considered are those which contain a subtree which is a binary tree over $(p_1, \ldots, p_i)$. If the value of all $H_{i,j}$ are already known, $F_{i,j}$ can be defined by the following:

$$F_{i,j} = \begin{cases} H_{i+1,j} + p_{1,j} & i+1 = j \\ \min_k(H_{i+1,j} + p_{1,j}, F_{i,k} + F_{k,j}) & i+1 < j \end{cases} \quad (2)$$

We now describe the modified algorithm, which makes use of relations (1) and (2), and which simulates $\log n$ iterations of RAKE followed by $\log n$ iterations of COMPRESS:

1. For $1 < i \leq j \leq n$, estimate $H_{i,j}$ to be 0 if $i = j$, $+\infty$ otherwise.
2. Iterate this step $\lceil \log n \rceil$ times: For all $1 < i < j \leq n$, re-estimate $H_{i,j}$ using relation (1) and the values of H computed during the previous estimation step.
3. For $1 \leq i < j \leq n$, estimate $F_{i,j}$ to be $H_{i+1,j} + p_{i,j}$, using the last estimate of $H_{i+1,j}$.
4. Iterate this step $\lceil \log n \rceil$ times: For all $1 \leq i < j \leq n$, re-estimate $F_{i,j}$ using relation (2) and the values of F computed during the previous estimation step.

5. Output the value $F_{1,n}$, which will be the minimum average word length of any Huffman code.

Intuitively, each re-estimation of the values of H simulates one RAKE step, while each re-estimation of the values of F simulates one COMPRESS step. Correctness of the algorithm follows from the fact that any left-justified binary tree can be reduced to the empty tree by $\lceil \log n \rceil$ iterations of RAKE followed by $\lceil \log n \rceil$ iterations of COMPRESS, and

Lemma 3.1 *For each monotonically increasing frequency vector $(p_1, \ldots, p_n)$, there is an optimal positional tree (Huffman tree) that is left-justified.*

[PROOF]: This lemma can be proven by a simple induction on n. In fact, the procedure given in the proof of Lemma 3.1 (Teng [18]) transforms any Huffman tree to a left-justified one. □

Theorem 3.1 *The Huffman Coding Problem can be solved in $O(\log n)$ time, using $O(n^3/\log n)$ processors on a CRCW PRAM.*

4 Multiplication of Concave Matrix

In this section we introduce a new subclass of matrices which we call concave matrices. A *concave* matrix is a rectangular matrix M which satisfies the *quadrangle condition* [19], that is $M_{ij} + M_{kl} \leq M_{il} + M_{kj}$ for all $i < k, j < l$ in the range of the indices of M.

Matrix multiplication shall be defined over the closed semiring $(\min, +)$, where the domain is the set of rational numbers extended with $+\infty$. For example, if M is the $n \times n$ matrix giving the weights of the edges of a complete digraph of size n, then M^k is the matrix giving the minimum weight of any path of length exactly k between any given pair of vertices.

We give a recursive concave matrix multiplication algorithm which takes $O(\log n \log\log n)$ time, using $n^2/\log n$ processors on a CREW machine and $O((\log\log n)^2)$ time, using $n^2/(\log\log n)$ processors on a CRCW machine. Our algorithm is very simple and has very small constant.

Theorem 4.1 *Two concave matrices can be multiplied in $O(\log n \log\log n)$ time, using $n^2/\log n$ processors on a CREW machine; and $O((\log\log n)^2)$ time, using $n^2/(\log\log n)$ processors on a CRCW machine.*

In the absence of the concavity assumption, the best known algorithm for computing AB requires $O(n^3)$ comparisons.

4.1 The Matrix $Cut(A, B)$

Let A be a concave matrix of size $p \times q$, B be a concave matrix of size $q \times r$. By the definition of matrix multiplication above, $(AB)_{ij} = \min\{A_{ik} + B_{kj} | 1 \leq k \leq q\}$. We can define a matrix $Cut(A, B)$ taking values in $[1, q]$ as follows: $Cut(A, B)_{ij}$ = that value of k such that $A_{ik} + B_{kj}$ is minimized. (If there is more than one value of k for which that sum is minimized, take the smallest.)

To compute AB it is clearly sufficient to compute $Cut(A, B)$, since we can construct AB from $Cut(A, B)$ in $O(1)$ time using pr processors. In the algorithm below, we just indicate how to compute $Cut(A, B)$.

Define A_{even} to be the submatrix of A consisting of all the entries of A whose row index is even, while (by an abuse of notation) we define B_{even} to be the submatrix of B consisting of all entries of B whose column index is even.

MULTIPLICATION ALGORITHM:
Procedure: $Cut(A, B)$
if A has just one row, or if B has just one column **then**

compute $Cut(A, B)$ by examining all possible choices

else

compute $Cut(A_{even}, B_{even})$ by recursion
compute $Cut(A_{even}, B)$ by interpolation
compute $Cut(A, B_{even})$ by interpolation
compute $Cut(A, B)$ by interpolation

fi

Interpolation:

The concavity property of A guarantees the following inequality:

$$Cut(A, B)_{ij} \leq Cut(A, B)_{i+1,j}$$

while concavity of B guarantees a similar inequality:

$$Cut(A, B)_{ij} \leq Cut(A, B)_{i,j+1}$$

The combination of these two properties we call the *monotonicity* property. By the monotonicity property, the total number of comparisons needed to compute $Cut(A, B)$, given $Cut(A_{even}, B)$, cannot exceed $(q - 1)r$. To see this, fix a particular column index j. For a particular odd row value of i, $q - 1$ comparisons could be needed to decide the value of $Cut(A, B)_{ij}$, since every k is a candidate. But monotonicity allows us to decide that value with only $Cut(A, B)_{i+1,j} - Cut(A, B)_{i-1,j}$ comparisons. Summed over all odd values of i, the total

number of comparisons needed (for the fixed value of j) is thus only $q-1$. For all j together, $(q-1)r$ are enough. Similarly, monotonicity allows us to compute $Cut(A_{even}, B)$ given $Cut(A_{even}, B_{even})$ using at most $p(q-1)/2$ comparisons and $Cut(A, B)$ given $Cut(A_{even}, B)$ and $Cut(A, B_{even})$ using at most $qr/2$ operations.

Time and Work Analysis:

Except for the recursion, the time to execute the multiplication algorithm is $O(\log q)$ or $O(\log\log q)$ on a CREW and CRCW machine, respectively. Since the depth of the recursion is $\min\{\log p, \log r\}$, the total time is $O(\log q(\min\{\log p, \log r\})$ on a CREW machine and $O(\log\log q(\min\{\log p, \log r\})$ on a CRCW machine.

4.2 More Efficient Concave Matrix Algorithm

In the MULTIPLICATION ALGORITHM given in the above subsection, the size of matrices is getting small during the recursion, while the number of processor available is still n^2. Hence at a certain stage, we do have enough processors to run the general matrix multiplication algorithm to compute the Cut matrix in one step. This implies that we can stop the recursion whenever the sizes of matrices are smaller enough to run the general matrix multiplication algorithm. Thus, the parallel (concave matrix) multiplication algorithm can be speeded up.

For each integer m, let $A_{\mathrm{mod}m}$ be the submatrix of A consisting of all the entries of A whose row index is a multiple of m, while (by an abuse of notation) let $B_{\mathrm{mod}m}$ submatrix of B consisting of all the entries of B whose column index is a multiple of m. Clearly, $Cut(A_{\mathrm{mod}\lfloor\sqrt{n}\rfloor}, B_{\mathrm{mod}\lfloor\sqrt{n}\rfloor})$ requires n^2 comparisons, and can be computed in $O(\log n)$ time and $(\log\log n)$ time on a CREW machine and a CRCW machine, respectively.

The following is a bottom-up procedure for computing $Cut(A, B)$.

- for $m = 1$ to $\lceil\log\log n\rceil + 1$ do
  1. Compute $Cut(A_{\mathrm{mod}\lfloor n^{1/2^m}\rfloor}, B_{\mathrm{mod}\lfloor n^{1/2^m}\rfloor})$;;
  2. Compute $Cut(A_{\mathrm{mod}\lfloor n^{1/2^m}\rfloor}, B)$;;
  3. Compute $Cut(A, B_{\mathrm{mod}\lfloor n^{1/2^m}\rfloor})$;;

We now show that each step of the loop can be computed by n^2 comparisons.

Clearly, when $m = 1$, step (1) requires n^2 comparisons. And it follows from the monotonicity properties that in Step(2), each row requires $\sqrt{n}\cdot n$ comparisons. Since there is $\sqrt{n}$ rows, n^2 comparisons are sufficient. Similarly, Step (3) takes n^2 comparisons.

For $m > 1$, by monotonicity properties that each row (column) takes $n^{1/2^m}\cdot n$ comparisons. Since there are $n^{1-1/2^{m-1}}\cdot n^{1/2^m}$ rows (columns), n^2 comparisons are sufficient.

Therefore, the above algorithm takes $O(\log n\log\log n)$ time, using $n^2/\log n$ processors on a CREW machine or $O((\log\log n)^2)$ time, using $n^2/(\log\log n)$ processors on a CRCW machine.

5 The Parallel Generation of Huffman Code

Presented in this section is an efficient parallel algorithms for the Huffman Coding Problem. The algorithm runs in $O(\log^2 n)$ time, using $n^2/\log n$ processors on an CRCW PRAM. This algorithm improves upon the previous known $\mathcal{NC}$ algorithms significantly on the processor count. It hinges on the use of concave matrix multiplication.

By Lemma 3.1, for each nondecreasing vector $(p_1, \ldots, p_n)$, there exists an optimal ordered tree for $(p_1, \ldots, p_n)$ which is left-justified. From Corollary 2.1, it follows that there exists an optimal tree for $(p_1, \ldots, p_n)$ such that the heights of all subtrees induced by nodes not on the leftmost path are bounded by $\lceil\log n\rceil$.

This observation suggests the following paradigm for the Huffman Coding Problem.

1. **Constructing Height Bounded Subtrees**: for all $i \le j$, compute $\mathcal{T}_{i,j}$, an optimal tree for $(p_i, \ldots, p_j)$ whose height is bounded by $\lceil\log n\rceil$.
2. **Constructing the Optimal Tree**: using the information provided in the first step to construct an optimal Huffman tree for $(p_1, \ldots, p_n)$.

Assume that weights $p_1, \ldots, p_n$ are given in monotonically increasing order. Define a matrix S as $S[i, j] = \sum_{k=i+1}^{j} p_k$ for $i < j$, and $S[i, j] = +\infty$ for $i \ge j$. It follows easily that S is a concave matrix.

For each $h \ge 0$, then define a matrix A_h as follows. For $0 \le i < j \le n$, $A_h[i, j]$ be the average word length of the optimal Huffman tree for the weights $(p_{i+1} \ldots p_j)$, restricted to height h, i.e., minimum over all trees whose height does not exceed h. If no tree exists, *i.e.*, $i \ge j$ or $(j-i) > 2^h$, define $A_h[i, j] = +\infty$.

Note that A_0 is trivial to compute, while $A_h = \min(A_{h-1}, A_{h-1} * A_{h-1} + S)$, where $*$ stands for the matrix multiplication using the closed semiring $\{\min, +\}$.

Note also that $A_{\lceil \log n \rceil}[i,j]$ is equal to the average word length of the optimal tree for $(p_{i+1}, ..., p_j)$ of height bounded by $\lceil \log n \rceil$.

The following lemma was first proven by Garey [6] and is known as the Quadrangle Lemma. For a simplier proof see Larmore [11].

Lemma 5.1 *For each h, A_h is a concave matrix.*

Therefore, $A_{\lceil \log n \rceil}$ can be computed in $O(\log^2 n)$ time, using $n^2/\log n$ processors. This is the first step in the paradigm proposed above. We now show that the second step in the paradigm can be also reduced to multiplying concave matrices.

A square matrix can be identified with a weighted directed graph. It is well known [3] that if M is the matrix for a weighted digraph with $(n+1)$ vertices, that $\min(M, I)^n$ contains the solutions to the all-pairs minimum path problem for that digraph, where I is the identity matrix over the closed semiring $\{\min, +\}$.

We now show how to reduce the Huffman problem to a minimum weight path problem for a directed graph. The matrix M, defined below, will be the weight matrix for a directed graph,which is also called M, whose vertices are $\{0, 1, ..., n\}$.

$$M[i,j] = \begin{cases} +\infty & \text{if } i = j = 0 \\ 0 & \text{if } i = 0 \text{ and } j = 1 \\ A_{\lceil \log n \rceil}[i,j] + S[0,j] & \text{if } 0 < i < j \leq n \\ +\infty & \text{otherwise.} \end{cases}$$

It is easy to verify that M is a concave matrix.

The entries of M^k contain minimum weights of paths in the digraph M of length exactly k. For $i > 0$, $M^k[i,j]$ has no simple meaning in terms of Huffman trees. But $M^k[0,j]$ contains the minimum average word length over all Huffman trees on the weights $(p_1, ...p_j)$ which satisfies the following two properties:

1. There are exactly $k-1$ internal nodes on the left edge of the tree, i.e., the leftmost leaf is at depth k.

2. The tree is left-justified.

By Lemma 3.1, there is an optimal Huffman tree that satisfies the above two conditions for some k. Thus computing M^k for all k up to n will give us the optimal Huffman tree. Unfortunately, the amount of computation involved is too great.

This problem can be overcome by a very slight modification of the graph of the matrix M.

Define a matrix M' as follows: $M'[0,0] = 0$ and $M'[i,j] = M[i,j]$ otherwise.

Think of M' as a digraph derived from M by adding a self-loop of weight 0 at 0. It is easy to verify that M' satisfies the quadrangle condition [19]. Hence, M' is a concave matrix.

Note that any path of length k or less from 0 to j in M corresponds to a path of length exactly k from 0 to j in M'.

The left edge of the optimal Huffman tree has length less than n, therefore $(M')^k[0,n]$ equals the weighted path length of the optimal Huffman tree for any $k > n$.

Note that M' is a concave matrix. Moreover $(M')^k$ is also a concave matrix. Hence $(M')^{2^{\lceil \log n \rceil}}$ can be computed by starting with M', then squaring $\lceil \log n \rceil$ times. Using the parallel concave matrix multiplication algorithm, each squaring can be performed in $O(\log n)$ time, using $n^2/\log n$ processors.

Theorem 5.1 *The Huffman Coding Problem can be solved in $O(\log^2 n)$ time, using $n^2/\log n$ processors.*

6 Constructing Almost Optimal Binary Searching Trees in Parallel

In this section, the parallel construction of optimal binary search trees, an important data structure for data maintenance and information retrieval [10], is considered. An $O(\log^2 n)$ time, $n^2/\log^2 n$ processor parallel algorithm is given for constructing an approximate binary search tree whose weighted path length is within ϵ off the optimal, where $\epsilon = n^{-k}$. Note that the best known sequential optimal search tree construction algorithm, due to Knuth, takes $O(n^2)$ time. Hence, our algorithm is optimal up to approximation. This algorithm too hinges on the judicious use of concave matrix multiplication.

The sequential version of the optimal search tree problem was first studied by Knuth [10], who used monotonicity to give an $O(n^2)$ time algorithm.

Suppose we are given n names $A_1, \ldots, A_n$ and $2n+1$ frequencies $p_0, p_1, \ldots, p_n, q_1, \ldots, q_n$, where q_i is the probability of accessing A_i and p_i is the probability of accessing a word which is not in the dictionary and is between A_i and A_{i+1}.

A labeled proper binary tree $\mathcal{T}$ of n internal nodes and $(n+1)$ leaves is a *binary search tree* for $A_1, \ldots, A_n$ iff there is a one-one onto mapping from $A_1, \ldots, A_n$ to the internal nodes of $\mathcal{T}$ such that the inorder traversal of $\mathcal{T}$ gives the vector $(A_1, \ldots, A_n)$.

Let b_i be the depth of the i^{th} internal node and a_i be the depth of the i^{th} leaf. Then $P(T)$, the weighted path length of T, is defined to be $P(T) = \sum_i q_i(b_i + 1) + \sum_i p_i a_i$.

T is an *optimal* binary search tree for $(A_1, \ldots, A_n)$ if $P(T)$ is minimized over all possible search trees.

The optimal binary search tree problem is reducible to a dynamic programming problem over the closed semiring $\{\min, +\}$. Hence, it lies in $\mathcal{NC}$. However, the best known $\mathcal{NC}$ parallel algorithm requires n^6 processors.

Let the *weight* of a subtree be the sum of the p_i and q_i for the nodes and leaves in that subtree. Let the *depth* of the subtree to be the depth of its root in the whole tree.

Our parallel algorithm utilizes the following approximating lemma due to Gu:ttler, Mehlhorn, and Schneider [7].

Lemma 6.1 *If S is a subtree of an optimal tree, and if w and d are the weight of S and the depth of the root of S, then $d \leq C + \log(1/w)/\log(\phi)$, where $\phi \simeq 1.618...$ is the Golden ratio, and C is some small constant.*

The following is the outline of our parallel approximate optimal binary search tree construction algorithm.

1. Let $0 < \epsilon < n^{-1}$ and let $\delta = \epsilon/2n \log n$.

2. Define a p_i or q_i to be *small* if it is less than δ; Define a *run of small frequencies* to be a sublist starting and ending with a p value, where every p value and every q value in that sublist is small. Collapse every maximal run of small frequencies to a single frequency, which will then still be less than ϵ.

3. Let $H = O(\log(1/\epsilon))$ be the maximum height, given in the sense of Gu:ttler, Mehlhorn, and Schneider [7], of any optimal tree which has no subtrees (other than a single leaf) of weight less than δ.

4. Let T' be an optimal tree for the collapsed list of frequencies. Note that $height(T') \leq H$.

5. Let T be the tree of $n+1$ leaves obtained from T' as follows. If L is a leaf of T' whose frequency is one of the "collapsed" values obtained in step 2, replace L by an arbitrary binary tree of height no more than $\log n$, which contains all the low frequency nodes involved in the collapse.

The correctness of the algorithm is guaranteed by the following lemma due to Larmore [12].

Lemma 6.2 *The weighted path length of T will differ from that of the optimal tree by at most ϵ.*

Clearly, steps (1)–(3) and (5) can be performed optimally in $O(\log n)$ time. The bottleneck of the algorithm is step (4) which computes optimal binary search trees of height bounded by $H = O(\log n)$ for all pairs. Like the problem of constructing optimal Huffman tree of height bounded by $O(\log n)$, this problem can be also reduced to multiplication of concave matrices. Moreover, the number of concave matrix multiplications is bounded by $O(\log n)$. The formal description of the method is given in the full paper.

Theorem 6.1 *For any $0 < \epsilon < 1/n$, a binary search tree T can be found whose weighted path length is within ϵ of that of the optimal tree, in $O(\log(1/\epsilon)(\log n))$ time, using $n^2/\log^2 n$ processors.*

7 Constructing Trees from Given Leaf-Patterns

In this section, an optimal $O(\log n)$ time, $n/\log n$ processor EREW parallel algorithm is given for the tree construction problem when the leaf pattern is monotone or bitonic. Also presented is an $O(\log^2 n)$ time, $n/\log n$ processor parallel EREW PRAM algorithm to the tree construction problem with general leaf patterns. This involves an $\mathcal{NC}$ reduction for the general tree construction problem to the tree construction problem with bitonic leaf patterns. Consequently, an optimal, $O(\log n)$ time EREW parallel algorithm is obtained for constructing Shannon-Fano code.

7.1 Monotonic Leaf Patterns and Bitonic Leaf Patterns

There is an elegant characteristic function, due to Kraft [5], to determine whether there is a solution to the tree construction problem with a monotone leaf pattern.

Lemma 7.1 (Kraft [5]) *There is a solution to the tree construction problem for a monotone leaf pattern $(l_1, \ldots, l_n)$ iff $\sum_{i=1}^{n} 1/2^{l_i} \leq 1$.*

In using the Kraft sum one has to be careful that the numbers added have only $O(\log n)$ bits in their representations and not $O(n)$ as they naively appear to have in the Kraft sum.

Suppose $(l_1, \ldots, l_n)$ is a monotone leaf pattern. Since it is sorted we can construct a vector $a = (a_1, \ldots, a_m)$ such that $a_i =$ the number leaves at level i and $m = l_1$ in $O(\log n)$ time optimaly. In the case when $l_1 > n$ we must store a as linked-list of nonzero entries. For simplicity of the exposition assume that $m \leq n$. We first show how to reduce a to a vector such that $a_i \leq 2$ for $1 \leq i \leq n$. We compute a vector a' from a by setting $a'_{i-1} = \lfloor a_i/2 \rfloor + (a_{i-1} \bmod 2)$. It follows by the Kraft sum that the tree for a exists iff it does for a'. Further, from the tree for a' we can in unit time construct one for a. This reduction from a to a' is very analogous the the RAKE in the Huffman code algorithm for left-justified trees. We shall apply this reduction until the $a_i \leq 2$, at most $O(\log n)$ times. To see that we only need $n/\log n$ processors for the $\log n$ reductions, observe that the total work is $O(\sum_{a_i \geq 2} \log a_i) \leq n$. To balance the work, any processor that computes a'_i at a given stage will be required to compute a'_{i-1} at the next stage. Thus we distribute the $a_i \geq 2$ based on the work of a_i which is $\log a_i$. To construct a tree for a where $a_i \leq 2$ reduces to computing the sum of two n-bit numbers and their intermediate carries. This can all be done optimaly using prefix sums.

This gives the following theorem:

Theorem 7.1 *Trees with monotone leaf patterns can be constructed in $O(\log n)$ time, using $n/\log n$ processors on an EREW PRAM.*

A pattern $(l_1, \ldots, l_n)$ is a bitonic pattern if there exists i such that $(l_1, \ldots, l_i)$ is monotone increasing and $(l_i, \ldots, l_n)$ is monotone decreasing.

Lemma 7.2 *The tree construction problem for a bitonic leaf pattern $(l_1, \ldots, l_n)$ has a solution iff $\sum_{j=1}^{n} 2^{-l_j} \leq 1$.*

Using the methods presented for monotone leaf patterns and the above lemma we get the following theorem:

Theorem 7.2 *A Tree from a bitonic leaf pattern can be constructed in $O(\log n)$ time, using $n/\log n$ processors on an EREW PRAM if it exists. In general, the minimum number of trees (in order) will be generated with the prescribed leaf pattern.*

7.2 General Leaf-Patterns

Presented in this subsection is an $O(\log n)$ time reduction from the tree construction problem with a general leaf pattern to the one with bitonic leaf patterns. Moreover, the reduction can be performed with $n/\log n$ processors. Therefore, an $O(\log^2 n)$ time, $n/\log n$ processor EREW PRAM parallel algorithm results.

A *segment-representation* of a pattern $(l_1, \ldots, l_n)$ is $((l'_1, n_1), \ldots, (l'_m, n_m))$ where $\sum_{j=}^{m} n_j = n$, $l'_j \neq l'_{j+1}$, and

$$(l_1, \ldots, l_n) = (\underbrace{l'_1, \ldots, l'_1}_{n_1}, \ldots, \underbrace{l'_m, \ldots, l'_m}_{n_m})$$

For simplicity, $((l'_1, n_1), \ldots, (l'_m, n_m))$ is also called a *pattern*. In a pattern $((l_1, n_1), \ldots, (l_m, n_m))$, l_i is a *min-point* if $l_{i-1} > l_i < l_{i+1}$; l_i is a *max-point* if $l_{i-1} < l_i > l_{i+1}$.

In a pattern $((l_1, n_1), \ldots, (l_m, n_m))$, $(l_i, \ldots, l_j)$ is a *right-finger* if (1) l_{i-1} is a min-point and for no $i \leq k \leq j$ is l_k a min-point (2) $l_{j+1} \leq l_{i-1} < l_j$. A left-finger is defined similarly except that l_{j+1} is a min-point. Note that a finger may be both a left and a right finger. We next show how to "remove" every finger from a leaf pattern using the tree construction for bitonic patterns. Finally, we observe that the new pattern will have at most half as many fingers as before. Thus we need only remove fingers $O(\log n)$ times.

Finger-Reduction applied to one finger $(l_i, \ldots, l_j)$ in a pattern $((l_1, n_1), \ldots, (l_m, n_m))$ is defined as follows: Without loss of generality assume that it is a right-finger and $l_{j+1} < l_{i-1}$. Set

$$K = \lceil \sum_{k=i}^{j} \frac{n_k}{2^{l_k - l_{i-1}}} \rceil.$$

Finger-Reduction returns the pattern:

$$((l_1, n_1), \ldots, (l_{i-1}, n_{i-1} + K), (l_{j+1}, n_{j+1}), \ldots, (l_m, n_m)).$$

We have just relaced a finger with the number (from Lemma 7.2) of leaves at level l_{i-1} that are needed to generate it. In general Finger-Reduction will simultaneously remove all fingers, both left and right fingers. It will return with a pattern.

To see that Finger-Reduction reduces the number of finger by at least one half, observe that Finger-Reduction removes all max-points. It is not hard to see that the only candidates for max-points are l_i which were previously min-points and also adjacent to a left and right finger. Thus the worst case for reducing the number of fingers of a pattern is when the pattern consists of consecutive pairs of left and right fingers that share a min-point. The next Lemma summerizes this:

Lemma 7.3 (Finger Cut Lemma) *If a pattern $(l'_1, \ldots, l'_n)$ is obtained by Finger-Reduction from*

a pattern $(l_1, \ldots, l_n)$, *then the tree construction problem with pattern* $(l_1, \ldots, l_n)$ *has a solution iff there is a solution to the tree construction problem with pattern* $(l'_1, \ldots, l'_n)$.

To obtain the tree for a pattern we apply Finger-Reduction until the pattern is reduced to a single finger. We then construct the *root* tree for the finger. In an expansion phase we attach the trees constructed while removing the finger during Finger-Reduction to the root tree.

Theorem 7.3 *A tree can be constructed for a pattern* $(l_1, \ldots, l_n)$ *with* m *fingers* $O(\log n \log m)$ *time, using* $n/\log n$ *processors.*

7.3 Constructing Approximate Optimal Trees

The Shannon-Fano coding method can be specified as: upon input $(p_1, \ldots, p_n)$, *compute* $(l_1, \ldots, l_n)$ such that $\log \frac{1}{p_i} \leq l_i \leq \log \frac{1}{p_i} + 1$, then construct a prefix code $C = (c_1, \ldots, c_n)$ such that $| c_i | = l_i$.

The proof to the following claim can be found in [8]:

Claim 7.1 *Let* $SF(A)$ *be the average word-length of the Shannon-Fano code of* $A = \{a_1, \ldots, a_n\}$ *and* $HUFF(A)$ *be the one of the Huffman code, then*

$$HUFF(A) \leq SF(A) \leq HUFF(A) + 1$$

The second part of the Shannon-Fano method can be implemented by the parallel tree construction algorithm presented in Section 7.1. Therefore,

Theorem 7.4 *In* $O(\log n)$ *time, using* $n/\log n$ *processors, a prefix code can be constructed with average word length bounded by that of the corresponding Huffman code plus one.*

8 Parallel Linear Context-free Language Recognition

In this section, the parallel complexity of *linear Context-free Language* recognition is considered. The linear CFLs' recognition problem is reduced to a *path problem* in a graph which has a family of small separators. An $O(\log^2 n)$ time, $M(n)$ processor parallel algorithm is obtained for linear CFLs' recognition by using the *parallel nested disection* of Pan and Reif [16]. Here $M(n)$ is the number of processors needed to multiply two $n \times n$ boolean matrices in $O(\log n)$ time in the CRCW PRAM model.

Definition 8.1 (Context-free Language) *A context-free grammar is a 4-tuple* $G = \{V, \Sigma, P, S\}$ *where:*

- V *is a finite nonempty set called the total vocabulary;*
- $\Sigma \subseteq V$ *is a finite nonempty set called the terminal alphabet;*
- $S \in V - \Sigma = N$ *is called the start symbol;*
- P *is a finite set of rules of the form:*

$$A \rightarrow \alpha, \text{ where } A \in N, \ \alpha \in V^*$$

A context-free grammar $G = \{V, \Sigma, P, S\}$ *is* linear *if each rule is of the form:* $A \rightarrow uBv$, *where* $A, B \in N$, $u, v \in \Sigma^*$

Let $G = \{V, \Sigma, P, S\}$ *be a context-free grammar, and let* $w, w' \in V^*$, w *is said to* directly generate w', *written* $w \Rightarrow w'$ *if there exist* $\alpha, \beta, u, v \in V^*$ *such that* $w = \alpha A \beta, w' = \alpha u B v \beta \in V^*$, *and* $A \rightarrow uBv \in P$. $\stackrel{*}{\Rightarrow}$ *stands for the reflexive-transitive closure of* $\Rightarrow$.

The language generated by G, *written* $L(G)$ *is the set*

$$L(G) = \{w \in \Sigma^* \mid S \stackrel{*}{\Rightarrow} w\}$$

The CFL-recognition problem is defined as: given a context-free grammar, G *and a finite sequence* $w = w_1 \ldots w_n \in \Sigma^*$, *decide whether* $w \in L(G)$ *(and generate a parse tree).*

Each linear context-free grammar $G' = \{V', \Sigma, P', S\}$ can be *normalized* by constructing another linear context-free grammar $G = \{V, \Sigma, P, s\}$ such that (i) $L(G) = L(G')$, (ii) P is a finite set of rules of the form

$$A \rightarrow bB, \text{ or } A \rightarrow a, \text{ or } A \rightarrow Cc,$$

where $a, b, c \in \Sigma$, $A, B, C \in N$.

A linear context-free grammar G' can be easily normalized by finding a G such that (i), (ii) are satisfied and moreover, the size of G is within a constant factor of that of G'. Throughout this section, it is assumed that the input linear context-free grammar is *normal* and its size is a constant with respect to n, the length of the input finite sequence.

Given a (normal) linear context-free grammar G and a finite sequence $w = w_1 \ldots w_n \in \Sigma^*$, a graph can be defined, $IG(G, w) = \{IV, IE\}$, called *induced graph* of G and w which has $|IV| = O(n^2)$ nodes. More specifically,

$$
\begin{aligned}
IV &= \{v_{i,j,p} \mid 1 \le i \le j \le n, p \in N\} \\
IE &= IE_l \cup IE_r \qquad \textit{where} \\
IE_l &= \{(v_{i,j,p}, v_{i,j-1,q}) \mid i < j,\ p \to qw_j \in P\} \\
IE_r &= \{(v_{i,j,p}, v_{i+1,j,q}) \mid i < j,\ p \to w_i q \in P\}
\end{aligned}
$$

We have the following observation.

Claim 8.1 *Let G be a linear context-free grammar and $w \in \Sigma^*$. Let $IG(G,w)$ be the induced graph of G and w. Then $w \in L(G)$ iff there exists a path in $IG(G,w)$ from $v_{1,n,s}$ to $v_{i,i,q}$ for some $i : 1 \le i \le n$, where $q \to w_i \in P$.*

The above observation reduces the linear context-free recognition problem to a *path problem* (reachability problem) in the induced graph $IG(G,w)$.

Let *cluster* i,j refer to the set of $|N|$ vertices of the form $v_{i,j,p}$ (see Figure 1). Note that if all vertices of each cluster i,j are "collapsed" into one vertex (call it $v_{i,j}$), then a planar grid graph is obtained (see Figure 2a) which we schematically draw as a triangle (see Figure 2b). Although $IG(G,w)$ itself is typically not planar, we shall take the liberty of talking about its *external face* to refer to the subset of its nodes that map into the external face of the collapsed version.

Figure 1: In $IG(G,w)$ the only edges leaving cluster i,j go to clusters $i,j-1$ and $i+1,j$.

Figure 2: A grid graph (a) and its schematic representation (b).

Let $m = O(n^2)$ denote the number of vertices in $IG(G,w)$, and $p_{IG} = O(n)$ be the *edge size* of $IG(G,w)$, i.e. the *perimeter* of the external face. The subset C, shown in Figure 3, is a separator of size $O(\sqrt{m}) = O(n)$, which partitions $IG(G,w)$ into four approximately equal components (in that figure the triangle is meant to depict $IG(G,w)$ itself rather than its collapsed version). Moreover, such a small separator can be uniformly found in each component recursively.

Figure 3: Illustrating the small separator C in $IG(G,w)$

The outline of the parallel algorithm becomes clear. Let U, M, L, R be the four pieces in $IG(G,w)$ induced by the separator C (see Figure 3). First, the reachability matrix $Reach_U$ between all pairs of vertices on the external face of U is recursively computed. The same is done for each of M, L, R, resulting in matrices $Reach_M, Reach_L, Reach_R$, respectively. Using the four boolean matrices returned by these four recursive calls, the reachability matrix $Reach_G$ between all pairs of vertices on the external face of $IG(G,w)$ is computed. This can be done simply by boolean matrix multiplication (actually three such multiplications), taking time $O(\log n)$ with $M(n)$ processors (it is known that $M(n) = O(n^{2+\epsilon})$ where $0 < \epsilon < 1$). Hence the time complexity of the algorithm is $O(\log^2 n)$. The processor count can be obtained by the following recurrence:

$$P(n) = \max(4P(n/2), M(n))$$

which implies $P(n) = O(M(n))$.

Theorem 8.1 *Linear context-free languages can be recognized in $O(\log^2 n)$ time, using $M(n)$ processors, where $M(n)$ is the number of processors needed to do boolean matrix multiply in $O(\log n)$ time.*

9 Open Questions

- Can the Huffman Coding Problem be solved in polylogarithmic time, using $o(n^{2-\epsilon})$ processors?
- Given a position tree, can we test whether it is a Huffman tree in polylogarithmic time, using $O(n^{2-\epsilon})$ processors?
- Can general context-free languages be recognized in polylogarithmic time, using $O(n^{6-\epsilon})$, or $o(M(n^2))$ processors?

- Can a linear context-free language be recognized in polylogarithmic time, using n^2 or $o(M(n))$ processors?
- Can the Optimal Binary Search Tree Construction be solved in polylogarithmic time, using fewer than $O(n^{6-\epsilon})$ processors?

Acknowledgements We would like to thank Manuela Veloso of CMU for carefully reading drafts of the paper and many helpful comments. We would also like to thank Alok Aggarwal for helpful discussions.

References

[1] A. Apostolico, M. J. Atallah, L. L. Larmore and H. S. McFaddin. Efficient parallel algorithms for string editing and related problems. In *Proc. 26th Annual Allerton Conf. on Communication, Control, and Computing*, Monticello, Illinois, September 1988, pp 253-263, 1988.

[2] A. Aggarwal and J. Park. Notes on searching in multidimensional monotone arrays. In *29th Annual Symposium on Foundations of Computer Science*, IEEE, 1988.

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

[4] R. Cole. Parallel merge sort. In *FOCS27*, pages 511–516, IEEE, Toronto, October 1987.

[5] S. Even. *Graph Algorithms.* Computer Science Press, Potomac, Maryland, 1979.

[6] M. R. Garey Optimal binary search tree with restricted maximal depth. *SIAM Journal of Computing*, 3:101–110, 1974.

[7] R. Guttler K. Mehlhorn and W. Schneider. Binary search trees: average and worst case behavior. *Elektron. Informationsverarb Kybernet*, 16:579–591, 1980.

[8] R. W. Hamming. *Coding and Information Theory.* Prentice-Hall, Inc., 1980.

[9] D. A. Huffman. A method for the construction of minimum redundancy codes. *Proc. IRE*, 40:1098–1101, 1952.

[10] D. E. Knuth. Optimal binary search trees. *Acta Informatica*, 1:14–25, 1971.

[11] L. L. Larmore. Height restricted optimal binary trees. *SIAM Journal of Computing*, 16:1115–1123, 1987.

[12] L. L. Larmore. A subquadratic algorithm for constructing approximately optimal binary search trees. *J. of Algorithms*, 8:579–591, 1987.

[13] B. McMillan. Two inequalities implied by unique decipherability. *IRE, Transaction on Information Theory*, ():185–189, 1956.

[14] G. L. Miller and J. H. Reif. Parallel tree contraction and its applications. In *26th Symposium on Foundations of Computer Science*, pages 478–489, IEEE, Portland, Oregon, 1985.

[15] G. L. Miller and S-H. Teng. Systematic methods for tree based parallel algorithm development. In *Second International Conference on Supercomputing*, pages 392–403, Santa Clara, May 1987.

[16] V. Pan and J. H. Reif. Fast and efficient parallel solution of linear systems. *SIAM Journal of Computing*, to appear, 1988.

[17] W. L. Ruzzo. On uniform circuit complexity. *Journal of Computer and System Sciences*, 22(3):, June 1981.

[18] S-H. Teng. The construction of Huffman-equivalent prefix code in *NC*. *ACM SIGACT*, 18(4):54–61, 1987.

[19] F. F. Yao. Efficient dynamic programming using efficient dynamic programming using quadrangle inequalities. In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing*, pages 429–435, ACM, 1980.

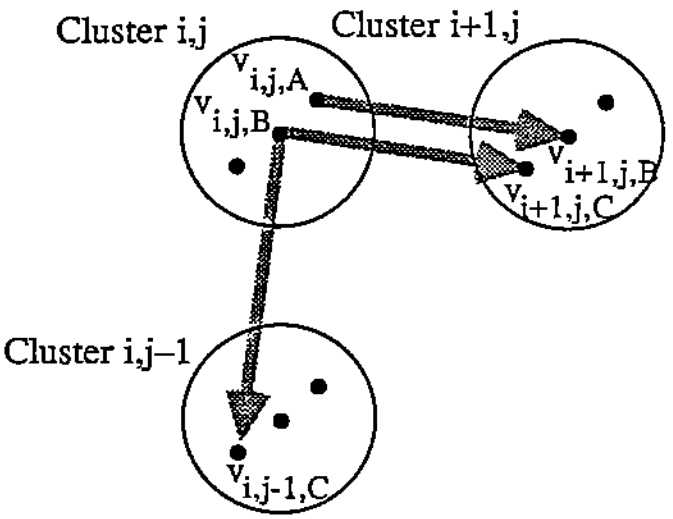

Figure 1

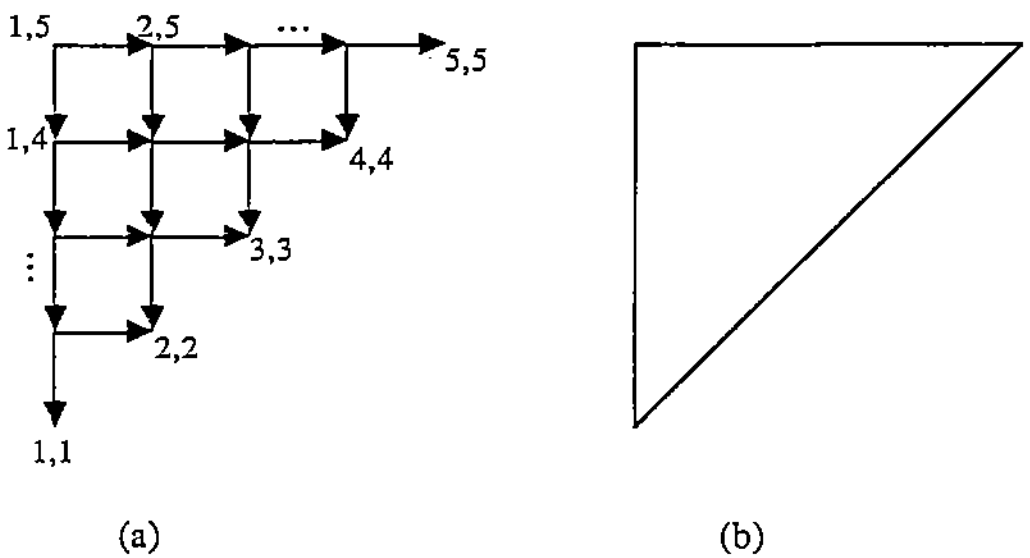

Figure 2

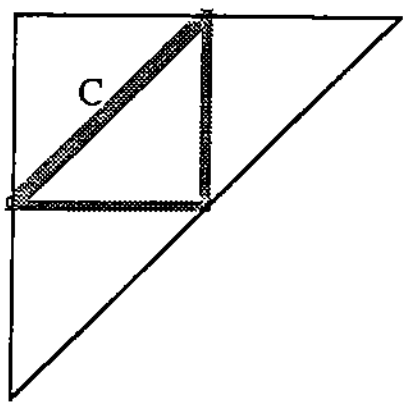

Figure 3