T-CP/36/2/11453



# A VLSI Implementation of the Simplex Algorithm

ALAN A. BERTOSSI AND MAURIZIO A. BONUCCELLI

*Abstract*—**The use of a special-purpose VLSI chip for solving a linear programming problem is presented. The chip is structured as a mesh of trees and is designed to implement the well-known simplex algorithm. A high degree of parallelism is introduced in each pivot step, which can be carried out in $O$ (log $n$) time using an $m \times n$ mesh of trees having an $O(mn \log m \log^3 n)$ area where $m - 1$ and $n - 1$ are the number of constraints and variables, respectively. Two variants of the simplex algorithm are also considered: the two-phase method and the revised one. The proposed chip is intended as being a possible basic block for a VLSI operations research machine.**

*Index Terms*—**Linear programming, mesh of trees, simplex algorithm, time complexity, VLSI.**

## I. Introduction

The proposal for highly dedicated parallel systems, based on *very large scale integration* (VLSI) technology, are becoming more and more realistic [10]. In fact, the number of gates that can be integrated in a single VLSI chip is steadily increasing and the processing capability of the chip is superior to conventional integrated circuits, even in terms of higher reliability, parallelism and speed, smaller dimension, reduced cost for packaging different subsystems, and low power consumption. The main problem remains the high cost of designing a VLSI chip, which becomes competitive only for large production quantities. Therefore, only application areas whose use is clearly consolidated can be envisaged for VLSI realization. A very important one is that of *linear programming*, namely, find a real ($n - 1$)-vector $x$ (the *variable* vector) to

$$\text{minimize} \quad z = cx$$

$$\text{subject to} \quad Ax = d$$

$$x \geq 0 \tag{1.1}$$

where $A$ is an $(m - 1) \times (n - 1)$ integer matrix (the *constraint* matrix), $d$ is an integer $(m - 1)$-vector (the *right-hand* side), and $c$ is an integer (row)$(n - 1)$-vector (the *cost* vector).

The well-known Dantzig's *simplex algorithm* is widely used to find an optimal solution to a linear programming problem. This algorithm is based on the idea of repeatedly improving the cost of the *objective function z* by moving from a *basic feasible solution (bfs)* to another bfs which differ by one variable. Such moving is performed by means of a quite costly *pivot step* (for an explanation of these terms and a detailed description of the algorithm we refer to [12]).

Despite its practical relevance, linear programming has received very little attention so far in the VLSI literature. In this paper, our aim is to partly fill this gap, by presenting a design of a VLSI chip for this problem. To begin, we consider a VLSI implementation of the simplex algorithm, in which a high degree of parallelism is introduced to perform each pivot step. However, this is intended only as a first contribution in this area. Indeed, our hope is to initiate a new vain of research, which could lead to the realization of a number of specialized VLSI chips spanning a large range of operations research topics, including, e.g., branch-and-bound computations, algorithms for finding maximum flows, shortest paths (which can be solved via matrix multiplications [13]) and minimum spanning trees [11], etc.

Manuscript received September 30, 1985; revised February 9, 1986. This work was supported by a grant from the Ministry of Public Instruction, Italy.

The authors are with the Department of Computer Science, University of Pisa, 56100 Pisa, Italy.

IEEE Log Number 8611453.

Ideally, the goal should be the realization of an *operations research machine*, in which such specialized VLSI chips are attached to a general-purpose host computer. Thus far, machines of this kind have already been successfully realized, for instance, for database processing (e.g., see [1], [2], [4], [8]).

Briefly, this correspondence is organized in the following fashion. In Section II we give a detailed description of the system architecture we will deal with. This architecture is based on the well-known interconnection pattern called *mesh of trees* [11]. In Section III we define a set of *elementary operations* and *tree procedures* which constitute the building blocks of our VLSI design. In Section IV we consider the simplex algorithm in which an $m \times n$ mesh of trees is used to store and process the linear programming data. We assume there that an initial bfs to start with is already known and that data (i.e., $A$, $c$, $d$) have been accordingly shaped. We show how to perform each pivot step in $O$ (log $n$) time using an $O$ ($mn$ log $m$ log$^3$ $n$) area. It is well known that sequential implementations of the simplex algorithm require $O$ ($mn$) time per pivot step while the number of such steps grows linearly in $m$, on the average, and exponentially, in the worst case. Thus our VLSI implementation takes in practice a time proportional to $m$ log $n$ for solving a linear programming problem. In Section V we consider the *two-phase* method, which allows to get an initial bfs. In Section VI we also sketch a VLSI realization of the *revised* simplex method. Lastly, we present in Section VII some concluding remarks as well as directions for future research.

## II. System Architecture

The basic computational structure considered in the present paper consists of elementary processing units (also called *nodes)* whose interconnection pattern is an $m \times n$ *mesh of trees* (*MT*, for short). In other words, there are $mn$ processing units arranged as an $m \times n$ array. Usually, $m$ and $n$ are powers of 2 and $m \leq n$. Then, the nodes in the $i$th row, $i = 0, 1, \cdots, m - 1$, of the array are the *leaves* of a complete binary tree ($rt_i$, for short). Similarly, the nodes in the $j$th column, $j = 0, 1, \cdots, n - 1$, are the leaves of a complete binary tree (denoted by $ct_j$). Nonleaf nodes are called *internal nodes.* For the sake of clarity, we shall denote with $\lambda_{ij}$ the leaf which lies on row $i$ and column $j$ of the mesh of trees, $i = 0, 1, \cdots, m - 1, j = 0, 1, \cdots, n - 1$. We shall also denote with $\rho_i$ the root of $rt_i$ and with $\gamma_j$ the root of $ct_j$. An example of a $4 \times 4$ mesh of trees is illustrated in Fig. 1.

A structure of this type was originally introduced by Leighton [9] and by Nath *et al.* [11]. We share the electronic assumptions of those papers. In particular, we assume that all processing elements operate in a *synchronous* fashion. This is accomplished by the use of a *main clock* broadcasting its pulses to every node in *MT*. We assume also that *MT* is connected to a general-purpose *host processor*, which supervises *MT*'s computations, and to its *main memory*.

### A. Why Mesh of Trees

The choice of the mesh of trees architecture is dictated by its peculiar interconnection pattern which allows fast ($O$ (log $n$)) communication among nodes. Indeed, under our fairly general assumptions, namely that each node stores one entry of (1.1)(or a constant number of them), the following results hold.

*Theorem 1:* Any $p \times q$ *array* ($p \times q = O(mn)$) requires at least $\Omega(\max\{p, q\})$ time to perform a pivot step.

*Theorem 2:* Any *binary tree* requires at least $\Omega(m)$ time to perform a pivot step.

The proofs of the above theorems are given in the Appendix.

Theorem 1 allows us to derive lower bounds on the time needed by two very common VLSI arrays. In particular, a *linear array* takes at least $\Omega(mn)$ time per pivot step and thus its performance is no better than that of a sequential implementation, while a *square array* with side $O(\sqrt{(mn)})$ gets an $\Omega(\sqrt{(mn)})$ lower bound. All these architectures behave poorly compared to the mesh of trees one which, as

0018-9340/87/0200-0241$01.00 © 1987 IEEE

Reprinted from: IEEE TRANSACTIONS ON COMPUTERS, Vol. C-36, No. 2, FEB. 1987.

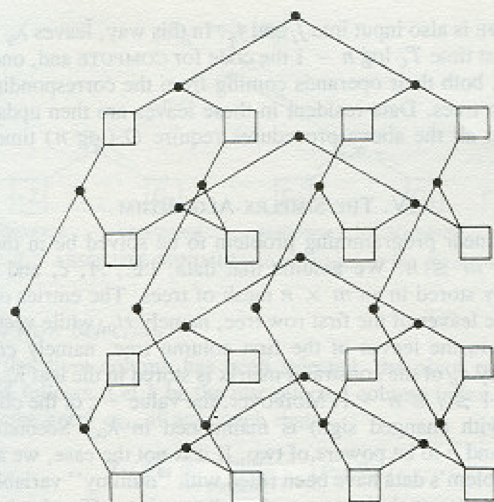

Fig. 1. A 4 × 4 mesh of trees.

shown in Section IV, can perform a pivot step in $O$ (log $n$) time still requiring only a slightly higher area.

### B. Node Architecture

Basically, only the communication pattern of the architecture would be important in order to achieve the desired time bounds for the simplex algorithm. Thus we could assume any kind of general processor organization at each node. However we sketch here a possible structure for the single nodes of $MT$, in order to show the attainment of a reasonable low area.

The architecture of a node can consist of a few registers, an *arithmetic and logic unit* and a *control unit*. Registers can be used to (temporarily or permanently) store data and column or row indexes. Thus their size equals either $b$, that of data to be processed, or log $m$ or log $n$ bits. Of course, each node has an *operation code* register, whose size $s$ amounts to a few bits.

The communication among nodes of each binary (row or column) tree is carried out through *bidirectional busses*, each capable of transmitting the content of a register in parallel. Thus busses have bandwidth equal to $b$, log $m$, log $n$ or $s$. We assume that data, indexes and operation codes are transferred through distinct busses. In this way, each node is connected to its parent and children in a row (column) tree via nine busses. Of course, each leaf is connected to both its parents in the corresponding row and column trees via six busses.

We assume that the host processor is the parent of each (row and column) root in $MT$ and that it can communicate with such a root by means of three appropriate busses.

By observing the elementary operations that will be defined in the next section, one can easily check that only a constant number of registers is required at each node. In particular, only three $b$-bit registers and one (log $m$ or log $n$)-bit register suffice for each internal node, while only two $b$-bit registers, one (log $m$)-bit register and one (log $n$)-bit register (for permanently storing its row and column indexes) are enough for each leaf.

## III. Node Operations and Tree Procedures

We now present a set of *elementary operations* that can be executed by the nodes in $MT$. The execution of most such operations is usually initiated by the host processor by issuing operation codes to some roots of $MT$.

Once a node has received a new operation code, it broadcasts such code to its children (the only exception for the SENDDOWN operation whose code, as described later, is sent to only one child). Afterwards, it starts the execution of an appropriate program corresponding to the operation to be performed. Such a program is stored in the memory of its own control unit. Operation codes are eventually followed by some parameters. We assume that the nodes not involved in performing an elementary operation always output a string of zeroes.

### A. Internal Node Elementary Operations

The basic set of elementary operations for internal nodes can consist of five operations: BROADCAST, MIN, MINPOS, SENDUP, and SENDDOWN.

BROADCAST is used to make two copies of a datum and an (row or column) index received from its parent and send them to its children.

MIN has the purpose of performing a comparison between two values coming from the children and returning the minimum one to the parent, along with an incoming index associated with it. MINPOS is the nonnegative analogue of MIN: the smaller nonnegative value coming from the children is returned to the parent along with the associated index. If a negative value (which always is $-1$) comes from both children, it is sent up to the parent along with an index value set to $-1$. When MIN or MINPOS is performed by a node in a column (row) tree, data coming from a leaf of row (column) zero are neglected.

SENDUP simply allows the execution of a logical OR on the data received by the children, returning the result to the parent.

SENDDOWN is used whenever a datum must be transferred from the root of a tree to a specific leaf. This is accomplished by associating the datum with the row index (if the involved tree is a column tree) or the column index (if the tree is a row one) of the destination leaf. This index is used to locate the leaf as follows. The operation code of SENDDOWN is sent to the left or right child, depending on the value, 0 or 1, respectively, of the leftmost bit of the index. Such an operation code is followed by the datum itself and by the index shifted by one bit to the left.

It is easy to verify that the above operations can all be carried out in constant time. For the sake of simplicity, we assume that each of them takes exactly the same time, say $T_C$ clock cycles, to be performed by a node. Besides it is clear that the program for all the above operations require constant memory space.

### B. Leaf Elementary Operations

The basic set of elementary operations for the leaves comprises six operations: STORECOL, SENDUPROW, SENDUPCOL, DIV, SIMPLEDIV, and COMPUTE.

STORECOL has the purpose of storing data coming from column trees. The datum to be stored is transmitted along with a row index. The incoming index is compared to the content of the appropriate row register of the leaf. If these values match, the datum is permanently stored into the leaf.

SENDUPROW and SENDUPCOL are used for transmitting up a row or column tree, respectively, a copy of the resident datum along with that of the column or row register.

A division operation has two versions: SIMPLEDIV and DIV. The former is used to divide the resident datum by the value coming from the row tree, provided that this value is not zero. The result is then permanently stored. The latter, instead, is used to divide the value coming from the row tree by the resident datum, provided that this latter is positive. If it is nonpositive, the result is set to $-1$.

The last elementary operation, namely COMPUTE, is used to update the resident datum by subtracting from it the result of the multiplication between the two values coming from the row and column trees.

These operations can all be executed in constant time, since the operands involved cannot be more than $b$ bits long and $b$ is a constant (see Section II). In particular, let $T_A$, $T_D$, and $T_M$ be the clock cycles taken by an addition, a division, and a multiplication, respectively. Then, DIV and SIMPLEDIV require $T_D$ clock cycles, while COMPUTE can be carried out in $T_A + T_M$ clock cycles. As before, we assume that all the remaining operations take exactly $T_C$ clock cycles. Besides, the programs for all the above operations require constant memory space.

### C. Tree Procedures

We now describe how the foregoing elementary operations can be combined in a few standard operation patterns which operate on the

row and column trees. We call such operation patterns *tree procedures*. The complete simplex algorithm will result from proper sequences of these procedures.

We distinguish seven procedures, namely, INPUT-AND-STORE, INPUT-AND-DIVIDE, BROADCAST-AND-COMPUTE, each of which (INPUT-AND-STORE excepted) can be performed either on a row or a column tree.

INPUT-AND-STORE is performed on a column tree to transfer a datum from a root to a specific leaf. To do this on the generic $ct_j$, the host processor puts the STORECOL operation code into $\gamma_j$ and, one clock cycle later, a SENDDOWN one. After one more clock cycle, the datum $\delta$ and the index $i$ of the destination leaf are fed into $\gamma_j$. After $T_C \log m$ clock cycles, $\delta$ reaches $\lambda_{ij}$, which permanently stores it by executing STORECOL. Of course, $T_C$ clock cycles after $i$ and $\delta$ are fed into $\gamma_j$, another index-datum pair can be sent down the tree in a pipelined fashion.

INPUT-AND-DIVIDE is analogous to INPUT-AND-STORE, the only difference being that DIV is used instead of STORECOL. In this way, it is possible to divide $\delta$ by the entry resident in $\lambda_{ij}$. This procedure takes $T_C \log n + T_D$ (or $T_C \log m + T_D$) time when performed on a row (or column) tree.

OUTPUT is used to transfer a copy of the resident datum and column (row) index from a leaf $\lambda_{ij}$ to the root of $rt_i(ct_j)$. To do this on $rt_i$, the host processor inputs a SENDUP operation code into $\rho_i$. This code is then broadcast to all nodes in $rt_i$. After $\log n - \log m$ clock cycles, the host processor inputs a SENDUPROW code into $\gamma_j$. Such a code is broadcast in $ct_j$ and after $\log m$ clock cycles is received by its leaves. In this way, at time $\log n$, each $\lambda_{kj}$, $0 \le k \le m - 1$, sends up a copy of the resident datum to its parent in $rt_k$. However, only the internal nodes in $rt_i$ do perform a SENDUP, and, at time $(T_C + 1) \log n + T_C$, $\rho_i$ receives the proper data. Of course, this procedure can be done in parallel on more than one tree. For instance, to accomplish this on $rt_i$ and $rt_k$, the host processor has only to input the SENDUP code into $\rho_i$ and $\rho_k$. As a result, $\rho_i$ and $\rho_k$ will receive at the same time data coming, respectively, from $\lambda_{ij}$ and $\lambda_{kj}$.

ABSOLUTE-MINIMUM has the purpose of finding the minimum entry among those stored in the leaves of a tree. To do this task on $rt_i$, the host processor issues the operation code for SENDUPROW and, one clock cycle later, that for MIN. Such operation codes are input into $\rho_i$, and broadcast to all the nodes in $rt_i$. After $\log n$ clock cycles, the leaves of $rt_i$ receive the code for SENDUPROW. Then, each leaf sends to its parent in $rt_i$ a copy of the resident datum and column index. Upon receiving indexes and data from its children, each internal node in $rt_i$ selects the smaller datum and the relative column index and sends them to its parent. Therefore, after additional $T_C \log n$ clock cycles, $\rho_i$ sends to the host processor the desired minimum.

POSITIVE-MINIMUM is the nonnegative analogue of the previous procedure, in which MINPOS substitutes MIN. If there is no nonnegative entry in the leaves, the root will receive data equal to $-1$.

BROADCAST-AND-DIVIDE is used to transfer a datum from the root of a tree to all its leaves. Moreover, entries resident in some selected leaves are divided by the incoming datum. For instance, to update the entries in $\lambda_{ij}$ and $\lambda_{ik}$, the operation code of BROADCAST is input by the host processor into $\rho_i$ along with, one clock cycle later, the appropriate datum $\delta$. After $T_C \log n - \log m$ clock cycles, the SIMPLEDIV code is input into $\gamma_j$ and $\gamma_k$. In this way, at time $\log n$, $\lambda_{ij}$ and $\lambda_{ik}$ receive the code for SIMPLEDIV (from $ct_j$ and $ct_k$) and the datum $\delta$ (from $rt_i$). Only data resident in $\lambda_{ij}$ and $\lambda_{ik}$ are updated, since all other leaves in $rt_i$ do not receive the SIMPLEDIV code and all the other leaves in $ct_j$ and $ct_k$ receive data equal to zero. The time needed is $T_C \log n + T_D$.

Lastly, BROADCAST-AND-COMPUTE is similar to the previous procedure, in which COMPUTE substitutes SIMPLEDIV. The main difference, however, is that the selected leaves, say $\lambda_{ij}$ and $\lambda_{ik}$, have to receive two data at the same time in order to perform COMPUTE. One datum has to come from the row tree and the other from the column tree. This is accomplished by putting at time $T_C$ ($\log n - \log m$) the BROADCAST operation code into $\gamma_j$ and $\gamma_k$, followed by the proper parameters. $(T_C - 1) \log m - 1$ clock cycles later, the code for COMPUTE is also input into $j_j$ and $j_k$. In this way, leaves $\lambda_{ij}$ and $\lambda_{ik}$ do receive at time $T_C \log n - 1$ the code for COMPUTE and, one clock cycle later, both their operands coming from the corresponding row and column trees. Data resident in these leaves are then updated.

Note that all the above procedures require $O(\log n)$ time to be performed.

## IV. The Simplex Algorithm

Let the linear programming problem to be solved be in the form (1.1), with $m \le n$. We assume that data, i.e., $A$, $c$, and $d$, are permanently stored in an $m \times n$ mesh of trees. The entries of $c$ are stored in the leaves of the first row tree, namely $rt_0$, while vector $d$ is maintained in the leaves of the first column tree, namely $ct_0$. The generic entry $a_{ij}$ of the constraint matrix is stored in the leaf $\lambda_{ij}$, $1 \le i \le m - 1$, $1 \le j \le n - 1$. Moreover, the value $-z$ of the objective function (with changed sign) is maintained in $\lambda_{00}$. Secondly, we assume $m$ and $n$ to be powers of two. If it is not the case, we assume that the problem's data have been filled with "dummy" variables and constraints so to reach the appropriate dimensions. Clearly, in doing so the time and area bounds that will follow change only by a constant factor. Thirdly, we assume that data are represented as $b$ bits numbers (see Section II). It is well known that the total area occupied by an $m \times n$ mesh of trees having constant area nodes and constant width communication lines is $O(m\, n \log m \log n)$[9]. Since $b$ is constant, nodes and communication lines require at most $O(\log n)$ area and width, respectively, in our proposal. Therefore, it is easy to see by an argument similar to that of [9] that the mesh of trees proposed in this paper can be laid out on an $O(m\, n \log m \log^3 n)$ area silicon chip. Finally, we assume throughout this section that a bfs for the linear programming problem is already known and that the data have been accordingly shaped. In particular, this means that matrix $A$ contains a permutation of the $(m - 1) \times (m - 1)$ *identity* matrix and $c$ contains the *reduced costs* relative to the current basis [12]. We suppose that the basis is kept by the host processor, e.g., by using a vector whose $k$th entry is the index of the $k$th basic variable.

We now give a detailed description of how the single stages of the simplex algorithm can be realized by using the proposed mesh of trees.

### A. Data Loading

The first task to be faced consists in loading the linear programming problem's data into $MT$. Since $m \le n$, it is convenient to load data one row at a time, by using column trees. We assume that all data are initially resident in the main memory and that I/O between $MT$ and the main memory can be carried out at a rate of $bn$ bits per clock cycle, thus allowing simultaneous transfer of $n$ data.

Loading a row can be done by performing $n$ INPUT-AND-STORE procedures simultaneously, one for each column tree. The overall loading can be achieved by iterating this $m$ times in a pipelined fashion. The first time $\gamma_j$ will receive the index 0 and $c_j$, $1 \le j \le n - 1$, while $\gamma_0$ will receive 0 and $-z$. $T_C$ clock cycles later $\gamma_j$ will receive the index 1 and $a_{1j}$, $1 \le j \le n - 1$, while $\gamma_0$ will receive 1 and $d_1$, and so on.

It is clear that after $O(m + \log m)$ time data loading is over and the computation of the real simplex algorithm can start.

### B. Pivot Column Selection

Firstly, the column of the pivot has to be selected. A classical rule for finding such a column is to determine the minimum reduced cost. If such a cost is nonnegative, then the current bfs is optimal; otherwise, the desired column has been found. This rule is known as the *most negative pricing rule* and can be implemented simply by performing an ABSOLUTE-MINIMUM procedure on $rt_0$ (see Fig. 2). If the final result in $\rho_0$ is negative, then the associated index, say $k$, is taken by the host processor in order to keep track that $x_k$ will enter the basis. Otherwise, no further decrease in the objective function is possible, and the actual bfs is optimal. In this latter case, the host processor initiates the output stage (see Subsection IV-F).

It is easy to convince ourselves that the overall running time to select the smallest reduced cost is thus $O(\log n)$.

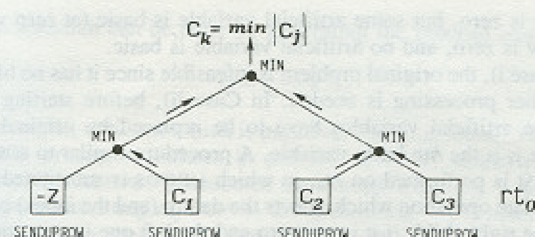


Fig. 2. ABSOLUTE-MINIMUM on $rt_0$ to select the pivot column.

### C. Pivot Row Selection

Once the pivot column has been selected, the pivot row can be located as follows. Let $k$ be the just selected column index. Then the *pivot* is the entry $a_{hk}$, if any, such that

$$d_h / a_{hk} = \min_{\substack{1 \le i \le m-1 \\ a_{ik} > 0}} \{d_i / a_{ik}\}. \tag{4.1}$$

The pivot row index can be found in the following way. Firstly, the entries of $d$ are sent to the row tree roots by means of $m - 1$ OUTPUT procedures done in parallel on row trees $rt_i$, $1 \le i \le m - 1$, in which the SENDUPROW operation code is put into $\gamma_0$. Afterwards, the host processor initiates $m - 1$ INPUT-AND-DIVIDE procedures simultaneously on the same row trees, but using $k$ as column index. Finally, the desired minimum is obtained in $\gamma_k$ after executing a POSITIVE-MINIMUM on $ct_k$. An example is given in Fig. 3. Therefore, it is clear that after $O$ (log $n$) clock cycles the pivot row index, say $h$, is available in $\gamma_k$. The host processor can thus keep track of this and determine the variable leaving the basis (i.e. the $h$th one). Of course, if the received index is $-1$, then the linear programming problem is unbounded and the host processor initiates the output stage.

### D. Greatest Decrement Rule

In the last two subsections, we have seen how to locate the pivot entry in $O$ (log $n$) time. In particular, we chose the pivot column by selecting the most negative reduced cost. In doing so there is no guarantee that after pivoting the improvement of the objective function will be the best possible. This goal, instead, can be achieved by choosing as pivot the entry $a_{hk}$ for which

$$c_k d_h / a_{hk} = \min_{c_j < 0} \{c_j \min_{\substack{1 \le i \le m-1 \\ a_{ij} > 0}} \{d_i / a_{ij}\}\}. \tag{4.2}$$

This rule is known as the *greatest decrement* rule. It is never used in sequential implementations of the simplex algorithm because requires $O(mn)$ time in the worst case. When properly implemented with our $MT$, however, it takes $O$ (log $n$) time only. We now sketch how this time bound can be achieved.

The greatest decrement rule can be implemented by means of a proper combination of the procedures developed in the last two subsections for finding the pivot column and row indexes. In particular, one can firstly perform this latter procedure in parallel, over all column indexes $j$, $1 \le j \le m - 1$. This solves the inner minima in (4.2) for all $j$. Then, the just found minimum ratios are sent in parallel through column trees to the leaves of $rt_0$. In this way, each leaf $\lambda_{0j}$, which stores the reduced cost $c_j$, receives the appropriate minimum ratio, $1 \le j \le m - 1$. Successively, each $\lambda_{0j}$ multiplies the resident datum by the incoming value, provided that the former is negative and the latter nonnegative, and transmits the result up to $rt_0$. If some operand has not the proper sign, a positive number is sent up $rt_0$. Finally, the outer minimum in (4.2) is performed in $rt_0$ as for the most negative rule.

### E. Pivoting

Once the pivot has been located, all problem's data have to be changed to reflect the fact that we are moving to another bfs. This can be done by means of *pivoting*. Let $a_{hk}$ be the selected pivot. Data are changed in two stages. Firstly, each entry in row $h$ is divided by the pivot. Successively, every other entry $a_{ij}$, $1 \le i \le m - 1$, $i \ne h$, $1 \le j \le n - 1$, is set to $a_{ij} - a_{ik} a_{hj}$. Entries of $d$, $c$, and $-z$ are similarly updated [12]. These two stages can be implemented by $MT$ as follows.

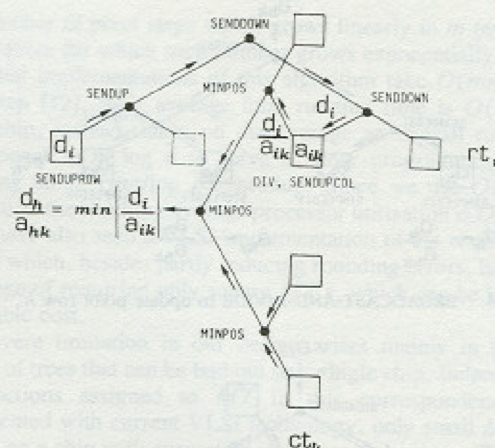


Fig. 3. Sequence of OUTPUT, INPUT-AND-DIVIDE (on all $rt_i$, $i \ne 0$) and POSITIVE-MINIMUM (on $ct_k$) to select the pivot row.

The host processor initiates an OUTPUT procedure on $rt_h$ with $k$ as column index, so to transfer the pivot $a_{hk}$ to $\rho_h$. Then a BROADCAST-AND-DIVIDE is performed: $a_{hk}$ is the datum to be broadcast to every leaf in $rt_h$, while the code for SIMPLEDIV and index $h$ are fed into each $ct_j$, $0 \le j \le n - 1$. (See Fig. 4 for an example).

This completes the execution of the first stage. The second stage can now be carried out as follows. The host processor issues $m - 1$ OUTPUT procedures in parallel in every row tree other than $rt_h$. The selected leaves are, of course, those lying on column $k$. At time $(T_C + 1) \log n + T_C$, the datum transmitted by $\lambda_{ik}$, $0 \le i \le m - 1$, $i \ne h$, is available in $\rho_i$. Then the host processor initiates a BROADCAST-AND-COMPUTE procedure in the same row trees, the selected leaves being simply all the leaves. In the meanwhile, the host processor starts, at time $(2T_C + 1)$ (log $n$ − log $m$), $n$ OUTPUT procedures in parallel on every column tree. As a result, each leaf $\lambda_{ij}$, $0 \le i \le m - 1$, $i \ne h$, $0 \le j \le n - 1$, does receive at time $(2T_C + 1) \log n - 1$ the operation code for COMPUTE and, one clock cycle later, both the data which reside in $\lambda_{ik}$ (namely $a_{ik}$, if $i \ne 0$, or $c_k$ if $i = 0$) and in $\lambda_{hj}$ (namely $a_{hj}$, if $j \ne 0$, or $d_h$ if $j = 0$) coming, respectively, from $rt_i$ and $ct_j$. The datum resident in $\lambda_{ij}$ is then updated. An example can be seen in Fig. 5.

A complete pivot step is thus over. A new pivot step can be carried out by repeating the procedures seen in Subsections IV-B through E until either an optimal bfs is found or the problem is discovered to be unbounded.

It is easy to see that pivoting requires $O$ (log $n$) time. Hence, the overall time taken by a complete pivot step is also $O$ (log $n$).

### F. Output of the Optimal Solution

When the simplex algorithm is over, the host processor receives proper data either from $\rho_0$ or from a column tree root (see Subsections IV-B and C). Then, the current bfs can be output from $MT$. Since nonbasic variables have to be set to zero, only the values to be assigned to the basic variables are needed. If $x_j$ is the $h$th basic variable, then its value is available in $\lambda_{0h}$, while the value of the objective function (with sign changed) is available in $\lambda_{00}$. Since the host processor took note of all changes in the successive bases, the output of all entries in the leaves of $ct_0$ is enough. This can be done by performing $m$ OUTPUT procedures in parallel, one for each row, selecting only those entries having column index equal to 0.

The output step is thus performed in $O$ (log $n$) time and $MT$ is now ready for processing a new linear programming problem.

## V. Two-Phase Method

In the previous section, we assumed that a bfs to start with was already known. When this is not the case, one bfs can be found by means of the so called *two-phase* method [12].

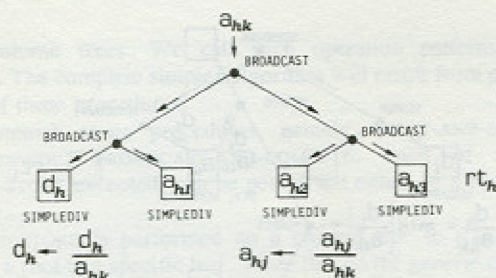


Fig. 4. BROADCAST-AND-DIVIDE to update pivot row $h$.

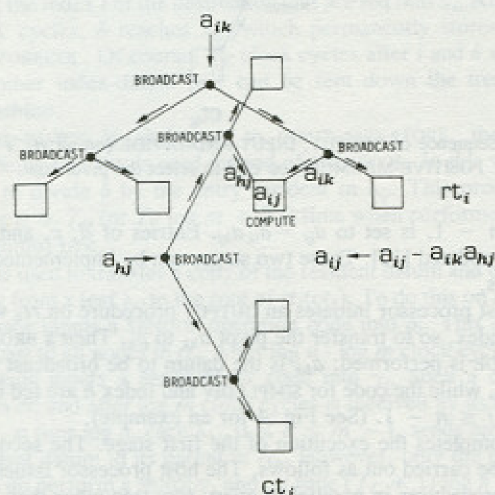


Fig. 5. BROADCAST-AND-COMPUTE to update all rows but row $h$.

The first phase consists in solving (by the ordinary simplex algorithm) the following *auxiliary* problem: find an $m - 1$ real vector $y$ (the *artificial* variable vector) to

$$\text{minimize } w = \sum_{i=1}^{m-1} y_i$$

$$\text{subject to } Ax + Iy = d$$

$$x,\ y \geq 0 \qquad (5.1)$$

where $I$ is the $(m - 1) \times (m - 1)$ identity matrix, and $A$ and $d$ are the original data of (1.1).

An initial basis for the auxiliary problem consists of exactly all the artificial variables. Then (5.1) can be solved by means of an $m \times (n + m - 1)$ mesh of trees. Of course, some additional procedures, very similar to those of Subsection III-C, are required. We leave their detailed implementation to the reader as an exercise.

As before, let us assume that both $m$ and $n + m - 1$ are powers of two. We can load the data for the first phase as usual. As a result, each leaf $\lambda_{0j}$ will contain zero, if $0 \leq j \leq n - 1$, or 1, if $n \leq j \leq n + m - 1$. To apply the simplex algorithm, however, the reduced costs relative to the (basic) artificial variables have to become zero. This can be accomplished in $O$ (log $n$) time by subtracting to the zeroth row of $MT$ the sum of the remaining rows as follows.

Firstly, data resident in the leaves of each column tree $ct_j$ are summed up, by means of $n$ properly defined SUMMATION procedures similar to ABSOLUTE-MINIMUM (thus neglecting data coming from row 0). These procedures are performed in parallel on all $ct_j$'s. Secondly, $n$ properly defined INPUT-AND-SUBSTRACT procedures are performed on the same column trees.

In this way, $MT$ is now ready to perform the computation of the usual simplex algorithm as seen above. When this is over, the following three cases may arise [12].

i) The objective function $w$ is greater than zero.

ii) $w$ is zero, but some artificial variable is basic (at zero value).

iii) $w$ is zero, and no artificial variable is basic.

In Case i), the original problem is infeasible since it has no bfs, and no further processing is needed. In Case ii), before starting phase two, the artificial variables have to be replaced by original ones. Assume $y_i$ is the $h$th basic variable. A procedure similar to POSITIVE-MINIMUM is performed on $rt_h$, in which MINPOS is substituted by an appropriate operation which selects the datum (and the index) coming from the right child, if it is nonzero and the left one is zero, or from the left child, otherwise. As a result, the host processor will receive a column index, say $k$. If $k \geq m$, then row $h$ is a linear combination of the others (since $a_{hj} = b_h = 0$ for all $j$) and can thus be deleted. Otherwise, pivoting can be performed as described in Subsection IV-E, the (possibly negative) pivot being $a_{hk}$ (in such a case, $y_i$ will be replaced in the basis by $x_k$, while the objective function will continue to be zero). This procedure is repeated for every basic artificial variable. Since the above replacing takes $O$ (log $n$) time, a bfs for the original problem (1.1) is obtained in $O(s \log n)$ time, if there are $s$ basic artificial variables. In this way, Case ii) is reduced to Case iii).

Finally, in Case iii) we have to restore the original objective function, compute the appropriate reduced costs, and reapply the simplex algorithm to perform phase two. In this case, however, all data resident in the leaves previously associated to artificial variables have to be filled with zeroes to guarantee correctness.

The transition between phase one and phase two is begun by loading the original cost vector $c$ in the leaves of $rt_0$. This can be done, as usual, in $O$ (log $n$) time. Like at the beginning of phase one, however, the reduced costs relative to basic variables have to become zero. This can be done in $O$ (log $n$) time by first multiplying each row $i$ by the cost of the $i$th basic variable, say $c_{k(i)}$(e.g. by means of an appropriately defined BROADCAST-AND-MULTIPLY), and then substracting to the zeroth row the sum of the so updated rows (as we already saw at the beginning of phase one).

It is easily seen that the transition between phase one and phase two is carried out in $O$ (log $n$) time. Now, phase two can be performed as shown in Section IV.

## VI. Revised Simplex Algorithm

In this section, we shall sketch a VLSI implementation of the so called *revised method* (for a description of this method we refer to [12]). The motivation for this is two-fold. On one hand, the revised method tries to use the original linear programming data whenever possible. As a result, rounding errors due to arithmetical operations are partly reduced. On the other hand, the revised method needs only to maintain in a fast memory an $(m - 1) \times (m - 1)$ submatrix instead of the whole $(m - 1) \times (n - 1)$ matrix $A$. This saving in space is particularly relevant from a VLSI viewpoint. Indeed, whenever $m - 1$ is much smaller than $n - 1$, the meshes of trees proposed in the previous sections must be laid out on chips with very small height-to-width ratio and their cost can be exceedingly high [10]. As we shall see in this section, however, the revised method can be implemented in VLSI by using one $(m + 1) \times (m + 1)$ $MT$, thus overcoming the above drawback. Of course, the saving in chip area is got by trading time for it, and a single pivot step can take at most $O((n - m) \log m)$ and at least $O$ ( log $m$) time.

Assume an initial basis is already known and let $B$ be the $(m - 1) \times (m - 1)$ submatrix of $A$ containing only basic columns. Similarly, denote with $c_B$ the $m - 1$ (row) subvector of c having only basic components. Finally, denote with $\pi$ the (row) vector $c_B B^{-1}$, where $B^{-1}$ is the inverse of $B$.

Since the revised method updates only $d$, $B^{-1}$, $-\pi$, and $-z$, we store them in one $(m + 1) \times (m + 1)$ $MT$. Specifically, $-\pi$ and $d$ are stored in the leaves of row 0 and column 0, respectively, $-z$ is stored as usual in $\lambda_{00}$, and the element which lies in row $i$ and column $j$ of $B^{-1}$ is stored in $\lambda_{ij}$, $1 \leq i \leq m - 1$, $1 \leq j \leq m - 1$. Row and column $m$ are devoted to bookkeeping processing. All other problem data are stored in the main memory.

Of course, data loading into $MT$ can be carried out in $O(m)$ time as shown in Subsection IV-A. To perform a pivot step, we firstly need to select the pivot column. Let $A_j$ be the $j$th column of $A$. The

above selection can be done by generating the reduced costs

$$r_j = c_j - \pi A_j \tag{6.1}$$

one at a time, until one which is negative, if any, is found. To do this, the host processor initiates in parallel $m-1$, properly defined INPUT-AND-MULTIPLY procedures on column trees $ct_j$, $1 \le j \le m-1$, by transmitting $a_{hj}$ from $\gamma_h$ to $\lambda_{0h}$ (because $-\pi$ is maintained in row 0). Then a SUMMATION procedure can be performed on $rt_0$ and $\rho_0$ can thus compute $r_j$. The overall time spent is $O(\log m)$.

The above procedure is repeated for each nonbasic variable until a negative reduced cost is found, if any. In the wrost case, at most $n-m-2$ iterations are needed, and the time spent for finding the pivot column is $O((n-m)\log m)$. If no negative reduced cost is found, the optimum bfs has been reached, and the output stage follows as described in Subsection IV-F.

Now, assume that a negative reduced cost, say $r_k$, has been found and that it is stored in $\lambda_{0m+1}$. Then $A_k$ has to be loaded into $MT$ from the main memory so to evaluate $B^{-1}A_k$. This can be done with $m-1$ *BROADCAST-AND-MULTIPLY* procedures performed in parallel on $ct_j$, $1 \le j \le m-1$, followed by $m-1$ *SUMMATION* procedures on $rt_i$, $1 \le i \le m-1$. As a result, $\rho_i$ will send to the host processor the $i$th entry of $B^{-1}A_k$. The just computed entries can then be stored into $ct_m$. It is easy to realize that the generation and storing of $B^{-1}A_k$ takes $O(\log m)$ time.

The pivot row can be selected as described in Subsection IV-C remembering that the pivot column corresponds to the (nonbasic) variable $x_k$ but is stored in the leaves of $ct_m$. This can be done in $O(\log m)$ time.

Assume that the selected pivot is stored in $\lambda_{hm}$. Then pivoting is performed as described in Subsection IV-E. As a result, the $O$th row and column of $MT$ will contain the updated $-\pi$ and $d$, respectively, while the leaves in rows and columns 1 to $m-1$ will store the updated $B^{-1}$. Of course, the host processor will keep track of the new basis, by replacing the $h$th basic variable with $x_k$. In order to reduce rounding errors, a VLSI chip for matrix inversion [7] could be used to periodically recompute $B^{-1}$, using the original matrix $A$, and store it into $MT$.

To get an initial $B^{-1}$ to start with, a two-phase method can be used. The revised algorithm then starts with an initial basis comprising exactly all the artificial variables (see Section V.) In this case, $B^{-1} = B = I$, the $(m-1)\times(m-1)$ identity matrix. To use phase one, however, artifical variables have to be at zero cost. This is accomplished by substracting all the rows of $A$ from the cost vector, so that in phase one the reduced costs $r_j$ are calculated by

$$r_j = -\sum_{i=1}^{m-1} a_{ij} - \pi A_j. \tag{6.2}$$

When using our $MT$, it is not necessary to compute the summation in (6.2) beforehand. In fact, it can be computed whenever needed by slightly changing the procedure presented above. The overall running time remains $O(\log m)$ per nonbasic variable.

When we enter phase two, the rule to compute the reduced costs becomes (6.1), instead of (6.2). Besides, the vector $-\pi = -c_B B^{-1}$ must be generated and placed into the leaves of $rt_0$. This last computation is similar to that of $B^{-1}A_k$ and can be carried out in $O(\log m)$ time.

## VII. Conclusions

In this correspondence, we have seen how a VLSI chip composed as a mesh of trees can be effectively used to implement the well-known simplex algorithm for linear programming. In particular, the concept of using the chip itself to store data, besides processing them, has permitted the attainment of a low $O(\log n)$ execution time per pivot step, while still maintaining a reasonable low $O(mn \log m \log^3 n)$ area.

It is well known that the simplex algorithm requires in practice a total number of pivot steps which grows linearly in $m$ (even if there are instances for which such number grows exponentially) [6]. Since sequential implementations of this algorithm take $O(mn)$ time per pivot step [12], their average total running time is $O(m^2 n)$. Our VLSI chip, instead, takes on the average an overall running time proportional to $m \log n$ to solve a linear programming problem, achieving an $O(mn/\log n)$ speedup. Since we use $O(mn)$ node processors, the resulting average processor utilization is $O(1/\log n)$.

We have also seen a VLSI implementation of the revised simplex method which, besides partly reducing rounding errors, has the great advantage of requiring only square $MT$'s, which can be laid out at a reasonable cost.

A severe limitation in our design arises mainly in the size of meshes of trees that can be laid out on a single chip. Indeed, while all the functions assigned to $MT$ in this correspondence can be implemented with current VLSI technology, only small $MT$'s could be built on a chip with current or imminent technology. The fact that small $MT$'s are becoming feasible, however, demonstrates that our proposal could become an attractive one, if VLSI technology will keep increasing performance and space properties.

Anyhow, assume that only small, say $q \times q$, with $q^2 < n$, meshes of trees can be laid out with present technology. Then a partitioning approach can be addressed. Since no more than $q^2$ entries can be updated simultaneously, data can be no more resident in $MT$ and each pivot step then requires several successive data inputs and outputs. It is easy to see that segmented versions of the procedures presented in Section IV can be effectively used. For instance, pivot column selection can be performed as follows. The cost vector $c$ is input into $MT$'s rows, $q$ entries at a time, and then sent to the leaves. Each leaf will receive $n/q^2$ successive entries and computes the minimum among them. Then, after $O(n/q + \log q)$ time, such minima are sent up the row trees, where procedures are performed which in $O(\log q)$ time find the minimum entry in each row tree. Finally, the smallest among such minima can be computed by letting each row tree root to send the just received minimum to the leaf which lies in column 0. Successively, $ct_0$ will compute in additional $O(\log q)$ time the required minimum. Thus the pivot column is selected in $O(n/q + \log q)$ time. Similarly, the pivot row can be located in $O(m/q + \log q)$ time, while pivoting takes $O((q + \log q)mn/q^2) = O(mn/q)$ time.

As already mentioned, the $MT$ proposed in this correspondence is intended as an initial contribution for the realization of a VLSI operations research machine. In particular, we are currently planning to continue our research in the following main directions. On one hand it should be very interesting to give VLSI implementations of further specializations of the simplex algorithm such as those using LU factoring or Cholesky's decomposition (e.g., see [6]). On the other hand, it should be even more interesting to consider an algorithm for linear programming which is completely different from the simplex one. Karmarkar's algorithm [5] seems a good candidate for VLSI realization, since it includes a method for self-correcting rounding errors and it modifies the current feasible solution by introducing several variables at once.

## Appendix

*Proof of Theorem 1:* The theorem will be proved by lower bounding the time needed to update the problem's data $A$, $c$, $d$. Let us assume first that a single copy of each entry is stored in the array. In the following, we will call *distance* between two nodes of the array, the minimum number of intermediate nodes met in going from one node to the other. Such distance is a lower bound on the time needed to transmit data between those nodes.

Let $a_{hk}$ be the pivot, $1 \le h \le m-1$, $1 \le k \le n-1$. The generic entry $a_{ij}$, $i \ne h$, $j \ne k$, needs both $a_{ik}$ and $a_{hj}$ to be updated. Clearly, if these entries are placed in nodes with distance $\Omega(\max\{p, q\})$ then the theorem is proved. Otherwise, assume that for any pair $i, j$, $1 \le i \le m-1$, $1 \le j \le n-1$, entries $a_{ik}$ and $a_{hj}$ are placed in nodes with distance at most $\delta = o(\max\{p, q\})$. Let $Q$ be the $\delta \times \delta$ square subarray containing all the nodes storing the entries of row h

and column $k$. Besides, let $C_1$, $C_2$, $C_3$, and $C_4$ be the square subarrays, each containing $pq/16$ nodes in one corner of the array. Thus, node $n_{uv}$ is in $C_1$ if $1 \leq u \leq p/4$ and $1 \leq v \leq q/4$ is in $C_2$ if $1 \leq u \leq p/4$; $3q/4 \leq v \leq q$ is in $C_3$ if $3p/4 \leq u \leq p$ and $1 \leq v \leq q/4$ is in $C_4$ if $3p/4 \leq u \leq p$, and $3q/4 \leq v \leq q$.

At most one among the above four corner subarrays can have nodes in common with $Q$, since otherwise $\delta = \Omega(\max\{p, q\})$. Let $C_1$ be such subarray. Each entry $a_{ij}$ stored in the other three subarrays needs two entries in $Q$ to be updated. But the distance between the node storing $a_{ij}$ and those in $Q$ is $\Omega(\max\{p, q\})$. Hence, the updating of $a_{ij}$ requires $\Omega(\max\{p, q\})$ time.

Finally, if several copies of each entry are stored in the array, all of them must be updated. Thus they can be seen as different entries, and the above proof can be used again.

*Proof of Theorem 2:* This proof is based on the minimum amount of data that must flow through the root during a pivot step. Since data flow through the root is sequential, the above minimum is a lower bound on the time needed to perform a pivot step.

A row (or column) of (1.1) will be called *covered* by the left (or right) subtree whenever all the entries of that row (or column) are stored in that subtree. If a row or a column is not covered by any subtree, pivoting will require that at least one of its entries be transmitted from a subtree to the other.

Let us assume that the left subtree stores $mn/k$ entries, with $k$ being a constant number. If this is not the case, i.e., if one subtree stores $o(mn)$ entries, we substitute the other subtree to the whole tree in the proof. Besides, let $R_1$ and $C_1(R_2$ and $C_2)$ be the number of rows and columns, respectively, covered by the left (right) subtree. We claim that either $R_1 \neq 0$ and $C_2 = 0$, or $R_1 = 0$ and $C_2 \neq 0$. Similarly, either $R_2 \neq 0$ and $C_1 = 0$, or $R_2 = 0$ and $C_1 \neq 0$. In fact, let $R_1 \neq 0$, and let row $i$ of (1.1) be covered by the left subtree. Then $a_{ij}$, $1 \leq j \leq n - 1$, and $d_i$ are stored in the left subtree. Thus no column can be covered in the right subtree. Similar reasonings can be done for the other cases. Therefore, only four further cases can arise.

*Case 1):* $R_1 \neq 0$ and $C_1 \neq 0$ (thus, $R_2 = C_2 = 0$). Since the left subtree contains $mn/k$ entries, we have

$$R_1 n + C_1 m - R_1 C_1 \leq mn/k.$$

Let us assume that $R_1 + C_1 = \alpha + (m + n)/k$, with $\alpha \geq 0$. Then $C_1 = \alpha - R_1 + (m + n)/k$ and substituting in the above inequality we get

$$R_1 n + m^2/k + mn/k + \alpha m - R_1 m < mn/k.$$

Thus we obtain

$$R_1(n-m) < -m^2/k - \alpha m < 0$$

which is a contradiction, since $n \geq m$. Thus $R_1 + C_1 < (m + n)/k$, and at least $(m + n)(k - 1)/k = O\ (m + n)$ rows and columns are not covered by any subtree.

*Case 2):* $C_1 \neq 0$ and $C_2 \neq 0$ (thus, $R_1 = R_2 = 0$). Since $C_1 + C_2 \leq n$, at least $m$ rows are not covered by any subtree.

The remaining two cases are analogous to the previous ones, and similar conclusions can be derived.

It follows that at least $\Omega(m)$ entries must be transferred between the two subtrees during a pivot step.

Finally, we observe again that whenever the entries are stored in multiple copies, each of them must be updated, and so they can be treated as different entries. Besides, it is easy to see that this proof holds also for non-binary trees, provided that each node has constant degree.

## References


[1] M. A. Bonuccelli, E. Lodi, F. Luccio, P. Maestrini, and L. Pagli, "A VLSI tree machine for relational data bases," in *Proc. 10th Ann. IEEE Symp. Comput. Architect.*, 1983, pp. 67–73.

[2] ——, "A VLSI mesh of trees for database processing," *Lect. Notes Comput. Sci.*, no. 159, pp. 155–166, 1983.

[3] D. Dobkin, R. J. Lipton, and S. Reiss, "Linear programming is log-space hard for P," *Inform. Processing Lett.*, vol. 8, pp. 96–97, 1979.

[4] D. K. Hsiao, "Database computers," in *Advances in Computers*, vol. 19, M. C. Yovits Ed. New York: Academic, 1982.

[5] N. Karmarkar, "A new polynomial time algorithm for linear programming," in *Proc. 16th Ann. ACM Symp. Theory of Comput.*, 1984, pp. 302–311.

[6] K. Murty, *Linear and Combinatorial Programming*. New York: Wiley, 1976.

[7] M. R. Kramer and J. Van Leeuwen, "Systolic computation and VLSI," in *Foundations of Computer Science, IV*, J. de Bakker and J. Van Leeuwen Eds. Amsterdam: Mathematische Centrum Tracts, 1983, pt. 1.

[8] H. T. Kung and P. L. Lehman, "Systolic (VLSI) arrays for relational database operations," in *Proc. ACM—SIGMOD Int. Conf. Manage. Data*, 1980, pp. 105–116.

[9] F. T. Leighton, "New lower bound techniques for VLSI," in *Proc. 22nd Ann. IEEE Symp. Found. Comput. Sci.*, 1981, pp. 1–12.

[10] C. Mead and L. Conway, *Introduction to VLSI Systems*. Reading, MA: Addison Wesley, 1980.

[11] D. D. Nath, S. N. Maheshwari, and P. C. Bhatt, "Efficient VLSI networks for parallel processing based on orthogonal trees," *IEEE Trans. Comput.*, vol. C-32, pp. 569–581, 1983.

[12] C. H. Papadimitriou and K. S. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*. Englewood Cliffs, NJ: Prentice-Hall, 1983.

[13] F. P. Preparata and J. E. Vuillemin, "Area-time optimal VLSI networks for matrix multiplication," in *Proc. 14th Princeton Conf. Inform. Sci. Syst.*, 1980.